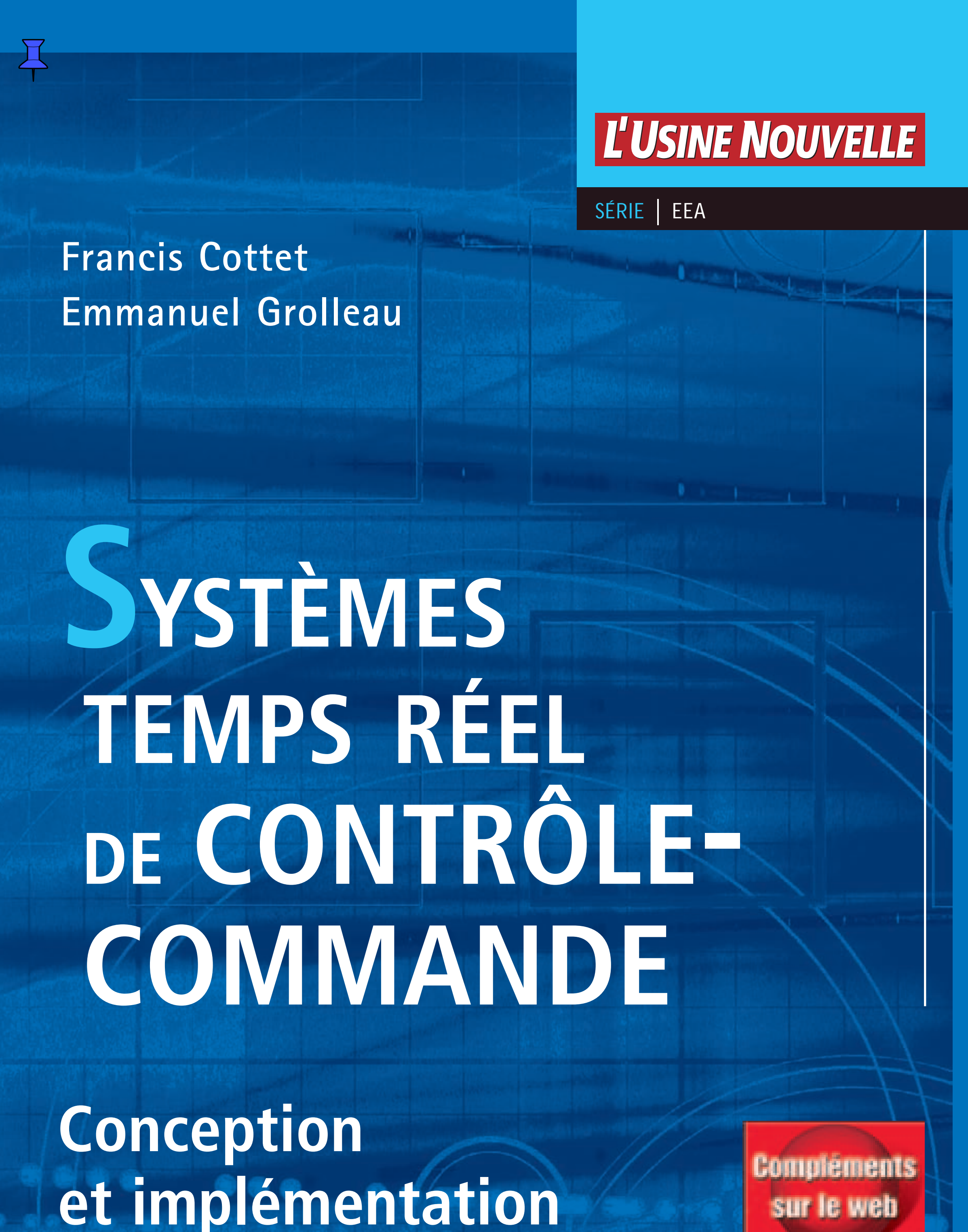
L'USINE NOUVELLE

SÉRIE | EEA

Francis Cottet
Emmanuel Grolleau



SYSTÈMES TEMPS RÉEL DE CONTRÔLE-COMMANDE

Conception et implémentation

Compléments sur le web

DUNOD

Francis Cottet
Emmanuel Grolleau

SYSTÈMES TEMPS RÉEL DE CONTRÔLE-COMMANDE

Conception et implémentation

DUNOD

Du même auteur : Francis Cottet

LabVIEW — Programmation et applications, Dunod, 2001

www.dunod.com

Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.

Le Code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée. Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).

© Dunod, Paris, 2005

ISBN 2 10 007893 3

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

TABLE DES MATIÈRES

Avant-Propos V

1 • Développement des systèmes de contrôle-commande 1

1.1 Introduction 1
1.2 Architecture des applications de contrôle-commande 7
1.3 Développement des applications de contrôle-commande 18

2 • Spécification selon la méthode SA-RT 27

2.1 Introduction générale à la méthode SA-RT 27
2.2 Présentation de la syntaxe graphique de la méthode SA-RT 30
2.3 Les diagrammes flot de données 36
2.4 L'aspect contrôle de la méthode SA-RT 41
2.5 Spécification des processus primitifs 49
2.6 Spécification des données 51
2.7 Organisation générale de la méthode SA-RT 54
2.8 Exemples 56
2.9 Extensions de la méthode SA-RT 70

3 • Conception selon la méthode DARTS 81

3.1 Introduction 81
3.2 Présentation de la méthode DARTS 85
3.3 Exemples de conception avec la méthode DARTS 102

4 • Architectures systèmes 109

4.1 Architecture matérielle 109
4.2 Architecture logicielle 138
4.3 Réseaux et bus de terrain 160

5 • Exécutifs temps réel 181

5.1 Introduction 181
5.2 Concepts des exécutifs temps réel 184
5.3 Principales normes temps réel 209
5.4 Exemples d'exécutifs temps réel 230

© Dunod – La photocopie non autorisée est un délit.

6 • Programmation des systèmes multitâches 245
6.1 Programmation C, Ada et LabVIEW 245
6.2 Programmation multitâche en langage C 285
6.3 Programmation multitâche en langage Ada 314
6.4 Programmation multitâche en langage LabVIEW 331

7 • Traitement complet d'une application industrielle 341
7.1 Cahier des charges 341
7.2 Spécification 342
7.3 Conception 350
7.4 Implémentation sur simulateur 354
7.5 Spécification et conception adaptées 388
7.6 Implémentation de la commande réelle 394
7.7 Conclusion 405

8 • Étude avancée des systèmes temps réel 407
8.1 Introduction 407
8.2 Modélisation des tâches 409
8.3 Ordonnancement des tâches indépendantes périodiques 431
8.4 Ordonnancement des tâches indépendantes apériodiques 447
8.5 Ordonnancement des tâches périodiques dépendantes 463
8.6 Analyse d'ordonnançabilité en environnement monoprocesseur 481
8.7 Ordonnancement en environnement multiprocesseur 491

Annexes

A • Représentation de l'information 513

B • Standards POSIX 519

C • Module de boîtes aux lettres POSIX 523

D • Module de communication Ada 533

Bibliographie 539

Lexique anglais – français 541

Sigles 546

Index 551

AVANT-PROPOS

Les applications informatiques dites de contrôle-commande ont envahi l'environnement industriel et notre vie quotidienne. Depuis quelques décennies, les besoins de plus en plus accrus en termes de technicité ont conduit à intégrer une très forte automatisation dans tous les produits industriels ou destinés à l'usage « grand public ». La liste infiniment longue des exemples contient des produits aussi divers qu'un téléphone mobile, un véhicule automobile, un four à micro-onde, une console de jeu, un satellite d'exploration, etc. Le dénominateur commun à toutes ces applications est la fourniture de fonctionnalités toujours plus sophistiquées : interface homme-machine (écran couleur de haute définition, écran tactile, commande vocale…), nombre élevé de fonctions débordant largement l'utilisation de base du produit (visualisation des commandes, liaison Internet…), sûreté de fonctionnement (robustesse, tolérance aux fautes, répartition, maintenance rapide et aisée…).
Pour ces raisons, les trois grands domaines permettant ces développements : l'électronique, l'automatique et l'informatique, ont dû progresser et s'adapter.

– Électronique : processeur multifonctions (microcontrôleur), processeur à faible consommation, réalisation de circuits électroniques dédiés (FPGA, FPLA), etc.
– Automatique : lois de régulations adaptées, régulation numérique, etc.
– Informatique : méthodes et méthodologies de développement, systèmes d'exploitation ou exécutifs embarqués, langages applicatifs, méthodes de tests et de validations, etc.

Les domaines de l'électronique et de l'automatique ne sont pas le propos de cet ouvrage. Toutefois, étant donné le développement lié entre les deux parties « matériel et logiciel », une présentation succincte du matériel est faite. En revanche, nous allons nous intéresser particulièrement à l'aspect informatique ou plus exactement à l'aspect génie logiciel, c'est-à-dire la ou les méthodes permettant de développer correctement ces applications. La signification du terme « correctement » est précisée dans le chapitre suivant, mais nous pouvons déjà annoncer que ces applications doivent être développées avec un grand souci de rigueur étant donné que leurs utilisations peuvent avoir un impact financier important, un effet nuisible sur l'environnement, ou plus gravement, mettre en jeu des vies humaines. Aussi la méthodologie de développement des applications de contrôle-commande doit assurer une qualité de réalisation en termes de fiabilité, d'efficacité, de maintenabilité, d'évolutivité, etc.
Il est important de noter qu'il n'est pas possible de parler des applications de contrôle-commande comme un ensemble homogène au sens de leur réalisation. En effet,

© Dunod – La photocopie non autorisée est un délit.

entre les développements de l'application informatique gérant un four à micro-onde et celle pilotant une navette spatiale, la distance est immense : criticité de l'application, taille du logiciel, évolution de l'application… Le seul lien en commun est que toutes ces applications mettent en relation un programme informatique avec un procédé externe.

D'autre part, le développement des applications de petite taille ou de taille moyenne a souvent été conduit par des professionnels du monde de l'automatique ou de l'électronique. Les régulations de type électromécanique ou électronique analogique ont rapidement fait place à des systèmes purement numériques ; seuls les capteurs et les actionneurs, faisant le lien vers le monde réel, sont toujours analogiques. Les concepteurs de ces applications ont des méthodes basées sur l'aspect fonctionnel : schémas blocs, grafcets, etc. En effet, la spécification et la conception de telles applications sont plus aisées lorsqu'elles sont pensées en termes de fonctions ou de traitements des données. Aussi les langages dédiés à ce type d'applications sont tout naturellement des langages fonctionnels à exécution séquentielle comme le langage C, les assembleurs ou le langage flot de données LabVIEW.

Dans le domaine informatique, en parallèle, ces applications ont conduit à la mise en place de méthodes de spécification ou de conception répondant à ces besoins, c'est-à-dire des méthodes basées sur la description de l'aspect fonctionnel, soit l'analyse structurée (SA), l'analyse structurée temps réel (SA-RT), la conception structurée (SD), etc. Ces méthodes correspondent parfaitement à ces besoins à condition que les applications ne deviennent pas de grande taille et n'impliquent des nombreuses données complexes (données structurées). D'autre part, il est important de noter que ces méthodes ont une dérivation très directe vers une implémentation multitâche qui permet de répondre à la fois à la logique du parallélisme du monde réel en termes de conception et aux contraintes temporelles exigées.

En ce qui concerne les applications informatiques dites classiques (applications bureautiques, bases de données…), des méthodes basées sur les données ont été élaborées afin de répondre à un besoin fort de modélisation des données et de leurs évolutions. Ces méthodes, dites orientées objets, permettent une spécification et une conception des applications en pensant avant tout en termes de données et d'actions sur ces données, mais en négligeant dans les premières étapes l'aspect fonctionnel, c'est-à-dire l'enchaînement, le séquencement ou l'exécution des différentes parties de l'application. Ces méthodes orientées objets, unifiées sous le nom UML (*Unified Modeling Language*), sont devenues incontournables pour le développement des applications informatiques.

Actuellement, ces méthodes orientées objets sont de plus en plus utilisées dans le domaine des applications de contrôle-commande, domaine non concerné initialement par ce type de modélisation objet. Il est concevable et acceptable de suivre une telle démarche pour des applications informatiques, même de contrôle-commande, de grande taille possédant souvent des données conséquentes de type complexe. Mais il ne semble pas naturel de vouloir imposer ce type d'outils pour des applications de contrôle de procédés de petite ou moyenne taille ayant des données en nombre réduit et de type simple. D'autant plus que, comme nous l'avons noté précédemment, les utilisateurs de ces outils ont une culture de conception de type fonctionnel qui a une très grande efficacité. Ensuite les méthodes orientées objets vont

conduire à des difficultés pour passer à l'étape de l'implémentation multitâche ; cette rupture de la chaîne de développement diminue fortement l'intérêt de ces méthodes de spécification et de conception.

Cet ouvrage a donc pris le parti de présenter une méthodologie complète de développement d'applications de contrôle-commande basé sur un aspect fonctionnel conduisant naturellement vers une implémentation multitâche. Sans rejeter les méthodes orientées objets largement répandues aujourd'hui, il semble contraire aux règles du génie logiciel d'utiliser une méthodologie non cohérente avec le domaine des applications ciblées, non cohérente à la fois en termes de logique d'analyse de bout en bout et d'objectifs applicatifs. **Notre méthodologie, basée sur une approche fonctionnelle au niveau de l'analyse et une approche multitâche au niveau de la conception, s'adapte parfaitement aux applications de contrôle-commande de petite taille ou de taille moyenne mettant en jeu des données simples.**

Le premier chapitre présente l'environnement de développement des systèmes de contrôle-commande en décrivant la spécificité de ces applications en termes d'architectures logicielles et matérielles. Le second chapitre traite de la méthode de spécification fonctionnelle choisie SA-RT (*Structured Analysis for Real Time systems*). La méthode de conception DARTS (*Design Approach for Real-Time Systems*) qui est la suite logique de SA-RT est décrite dans le chapitre 3. Les environnements matériels et logiciels (exécutifs temps réel) très particuliers de ces applications sont présentés dans les chapitres 4 et 5 afin de mieux comprendre la partie implémentation de ces systèmes de contrôle-commande. Le chapitre 6 est dédié à l'implémentation des applications de contrôle-commande en déclinant trois environnements : noyau temps réel et langage de type langage C, langage Ada et enfin un environnement spécifique basé sur LabVIEW. Les précédents chapitres sont illustrés par des exemples simples mais réalistes ; en revanche, le chapitre 7 propose le développement complet d'une application réelle industrielle. Enfin, le chapitre 8 ouvre le développement de ces applications vers des aspects avancés concernant l'ordonnancement.

Téléchargement sur Internet

Vous trouverez en téléchargement sur le site www.dunod.com les codes sources de tous les programmes présentés dans cette ouvrage.

1 • DÉVELOPPEMENT DES SYSTÈMES DE CONTRÔLE-COMMANDE

1.1 Introduction

1.1.1 Définitions

Nous pouvons définir un système de contrôle-commande comme un système informatique en relation avec l'environnement physique réel externe par l'intermédiaire de capteurs et/ou d'actionneurs, contrairement aux systèmes d'informatiques scientifiques (gestion de base de données, CAO, bureautique…) qui ont des entrées constituées de données fournies par des fichiers ou éventuellement un opérateur. Les grandeurs physiques acquises permettent au système de contrôle-commande de piloter un procédé physique quelconque. Donnons ainsi une définition générale d'un système de contrôle-commande (figure 1.1) :

> « Un système de contrôle-commande reçoit des informations sur l'état du procédé externe, traite ces données et, en fonction du résultat, évalue une décision qui agit sur cet environnement extérieur afin d'assurer un **état stable** ».

Cette notion d'état stable peut être différente selon les applications ou procédés. Il dépend du cahier des charges de l'application (maintien d'une température de consigne, régime moteur, qualité de service…). Deux caractéristiques font qu'un système de contrôle-commande ne possède pas les propriétés classiques des systèmes d'informatiques scientifiques :

– indépendance du résultat produit par rapport à la vitesse d'exécution. Le résultat d'un calcul effectué à partir de données d'entrée similaires est indépendant de la vitesse du calculateur. En revanche, l'état stable d'un procédé dépend de la dynamique du procédé par rapport à la vitesse d'exécution du système de contrôle-commande ;
– comportement reproductible. Un calcul effectué à partir de données d'entrée identiques donne toujours le même résultat. En revanche, dans le cas de données d'entrée (grandeurs physiques) obtenues par des capteurs, le système de contrôle-commande travaille sur un domaine de données réelles approximées qui sont très rarement identiques.

© Dunod – La photocopie non autorisée est un délit.

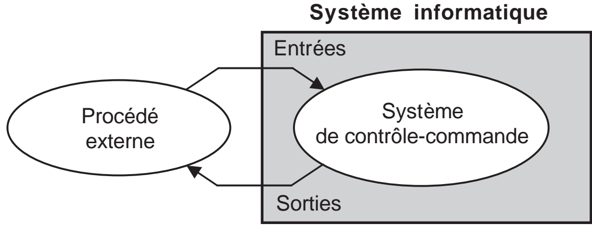


Figure 1.1 – Représentation schématique d'un système de contrôle-commande.

L'interaction du système de contrôle-commande avec le procédé extérieur à piloter se décompose en deux parties (figure 1.2) :

– observations par l'intermédiaire de capteurs (*sensors*) qui permettent d'obtenir des informations sous la forme des **interruptions** (information tout ou rien) ou des **mesures** (information continue) en provenance du procédé physique ;
– actions réalisées par l'intermédiaire d'actionneurs (*actuators*) qui permettent d'agir sur le procédé physique sous la forme de **commandes** (modification d'état physique du système) ou simplement sous la forme d'un **affichage** (diodes, lampes, afficheurs, écrans, etc.).

Cette définition des systèmes de contrôle-commande ayant été faite, nous pouvons replacer ces systèmes par rapport aux autres systèmes informatiques en faisant trois catégories :

– les **systèmes transformationnels** qui utilisent des données fournies à l'initialisation par l'utilisateur. Ces données, leurs traitements et l'obtention du résultat n'ont aucune contrainte de temps ;
– les **systèmes interactifs** dans le sens où les données sont produites par interaction avec l'environnement sous différentes formes (clavier, fichier, réseaux, souris, etc.). Mais le temps n'intervient pas en tant que tel si ce n'est avec un aspect confort de travail ou qualité de service ;
– les **systèmes de contrôle-commande** ou réactifs qui sont aussi en relation avec l'environnement physique réel pour les données en entrée ; mais, dans ce cas,

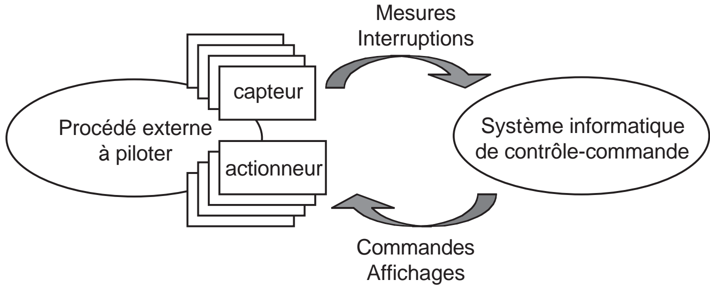


Figure 1.2 – Représentation schématique de l'interaction du procédé physique piloté et du système de contrôle-commande.

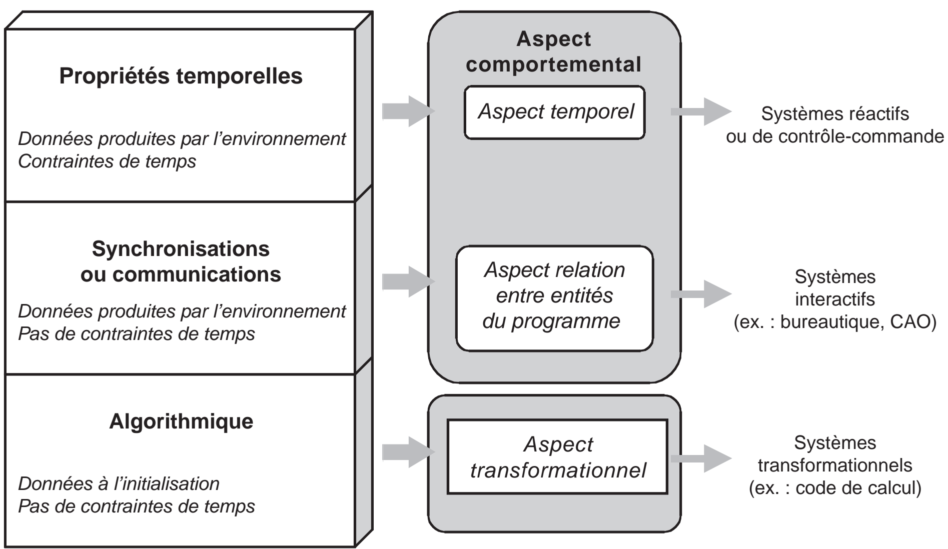


Figure 1.3 – Comparaison des systèmes de contrôle-commande par rapport aux autres applications informatiques.

l'aspect « temps » a une place importante sous la forme d'un temps de réaction, d'une échéance à respecter, etc.

Nous terminons cette section par des définitions qualifiant des systèmes de contrôle-commande ayant des spécifications particulières. La première de ces catégories concerne les systèmes temps réel (*real-time system*) dont la définition est : « un système de contrôle-commande dans lequel l'exactitude des applications ne dépend pas seulement du résultat mais aussi du temps auquel ce résultat est produit. Si les **contraintes temporelles** de l'application ne sont pas respectées, on parle de **défaillance** du système ». Ces contraintes temporelles peuvent être de deux types :

– **contraintes temporelles relatives** ou lâches (temps réel mou : *soft real-time*) : les fautes temporelles sont tolérables (ex. : jeux vidéo, applications multimédia, téléphonie mobile…) ;

– **contraintes temporelles strictes** ou dures (temps réel dur : *hard real-time*) : les fautes temporelles ne sont pas tolérables (ex. : avionique, véhicules spatiaux, automobile, transport ferroviaire…).

Dans le cas des systèmes temps réel à contraintes temporelles relatives, nous pouvons parler de systèmes de contrôle-commande classiques.
Nous pouvons aussi trouver les qualificatifs suivants :

– système de contrôle-commande **embarqué** (*embedded real-time system*) : pas d'intervention humaine directe (pas de modification du programme ou des paramètres du programme) ;

– système de contrôle-commande **dédié** (*dedicated real-time system*) : les architectures matérielles ou logicielles sont spécifiques à l'application (noyau, processeur…) ;

© Dunod – La photocopie non autorisée est un délit.

– système de contrôle-commande **réparti ou distribué** (*distributed real-time system*) : l'architecture matérielle est constituée de plusieurs processeurs reliés entre eux par un bus ou un réseau.

Il est évident que ces différentes spécifications d'un système de contrôle-commande peuvent se combiner comme par exemple un système de contrôle-commande dédié, distribué et à contraintes temporelles strictes (application pour un véhicule automobile).

1.1.2 Principales caractéristiques des systèmes de contrôle-commande

Considérons un exemple représentatif d'une application de contrôle-commande représenté sur la figure 1.4. Cet exemple de contrôle-commande d'un moteur à combustion est repris de façon détaillée dans le chapitre suivant. Le contrôle-commande de cette application est fait par l'intermédiaire d'un ensemble de capteurs et d'actionneurs (pédale d'accélérateur, température air, pression air, température eau, rotation vilebrequin, capteurs de pollution, injection essence, allumage, admission air, etc.) et d'une connexion au réseau interne à l'automobile. L'analyse de cet exemple d'application permet de mettre en exergue les principales caractéristiques des systèmes de contrôle-commande :

– **grande diversité des dispositifs d'entrées/sorties** : les données à acquérir qui sont fournies par les capteurs et les données à fournir aux actionneurs sont de types très variés (continu, discret, tout ou rien ou analogique). Il est aussi nécessaire de piloter un bus de terrain pour les communications ;

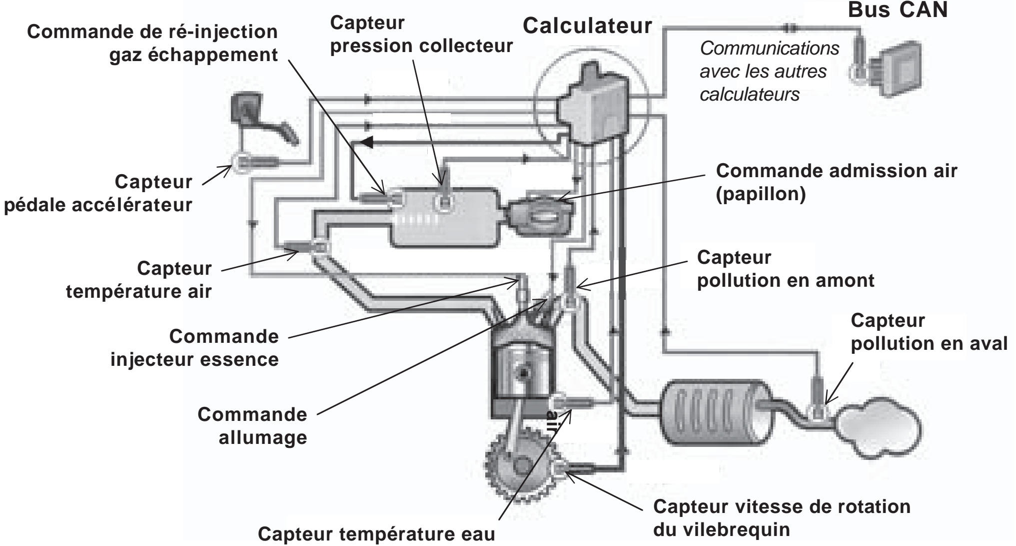


Figure 1.4 – Exemple d'une application de contrôle-commande d'un moteur à combustion.

– **prise en compte des comportements concurrents** : l'ensemble de ces donnés physiques qui arrivent de l'extérieur et le réseau qui permet de recevoir des messages ne sont pas synchronisés au niveau de leurs évolutions, par conséquent, le système informatique doit être capable d'accepter ces variations simultanées des paramètres ;
– **respect des contraintes temporelles** : la caractéristique précédente impose de la part du système informatique d'avoir une réactivité suffisante pour prendre en compte tous ces comportements concurrents et en réponse à ceux-ci, de faire une commande en respectant un délai compatible avec la dynamique du système ;
– **sûreté de fonctionnement** : les systèmes de type contrôle-commande mettent souvent en jeu des applications qui demandent un niveau important de sécurité pour raisons de coût ou de vies humaines. Pour répondre à cette demande, il est nécessaire de mettre en œuvre toutes les réponses de la sûreté de fonctionnement (développements sûrs, tests, méthodes formelles, prévisibilité, déterminisme, continuité de service, tolérance aux fautes, redondance, etc.).

1.1.3 Caractéristique temporelle des systèmes de contrôle-commande

Le respect des contraintes temporelles d'une application de contrôle-commande dépend essentiellement de la dynamique du procédé. Cette caractéristique temporelle peut être très différente suivant l'application (figure 1.5) :
– Milliseconde : systèmes radar, systèmes vocaux, systèmes de mesures…
– Seconde : systèmes de visualisation, robotique…
– Minute : chaîne de fabrication…
– Heure : contrôle de réactions chimiques…

Ce paramètre temporel correspond à l'ordre de grandeur de la capacité de réponse ou de traitement du système de contrôle-commande.

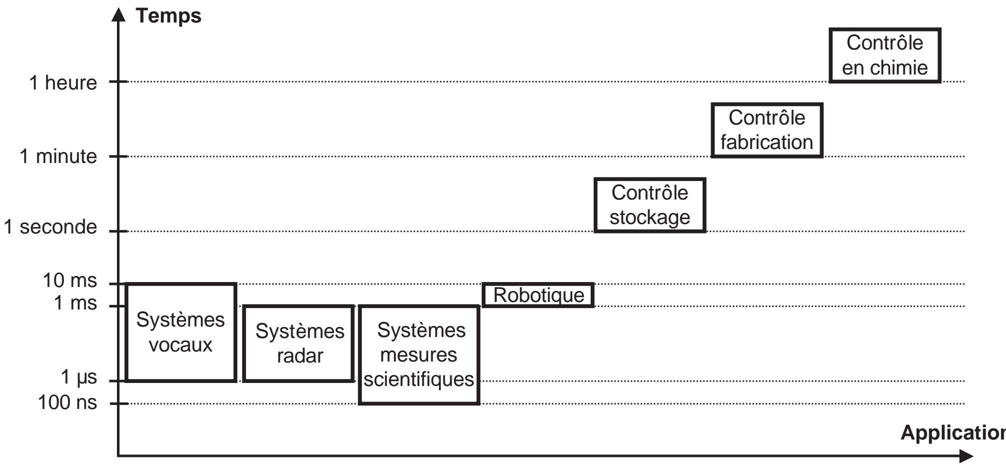


Figure 1.5 – Comparaison de la dynamique de différentes applications de contrôle-commande.

© Dunod – La photocopie non autorisée est un délit.

Mais, comme nous le verrons dans le chapitre 8, il est nécessaire de préciser et de formaliser cette caractéristique temporelle qui peut prendre de nombreuses formulations. Ainsi, nous pouvons définir de manière non exhaustive :

– **Durée d'exécution** d'une activité : l'activité d'une application, qui peut être l'enchaînement de plusieurs activités élémentaires (acquisition, traitement, commande, affichage…), possède une durée d'exécution qui peut être mesurée de diverses manières. Cette durée n'est pas constante à chaque occurrence de cette activité puisque les programmes et les enchaînements de programmes ne sont pas toujours identiques (branchement conditionnel, itération, synchronisation…).
– Cadence de répétition ou **périodicité** d'une activité : l'acquisition d'une donnée ou la commande d'un actionneur peuvent nécessiter une régularité liée par exemple à la fréquence d'échantillonnage.
– Date au plus tôt ou **date de réveil** : dans certains cas, un traitement doit être déclenché à une date précise relative par rapport au début de l'exécution de l'application ou absolue (plus rarement). Cette date de réveil n'implique pas obligatoirement l'exécution ; il peut y avoir un délai de latence dû à l'indisponibilité du processeur.
– **Date d'activation** : cet instant correspond à l'exécution effective de l'activité.
– Date au plus tard ou **échéance** : le traitement ou la commande d'un actionneur doivent être terminés à un instant fixé par rapport au début de l'exécution de l'application. Dans le cas d'applications à contraintes temporelles strictes, cette échéance doit être respectée de façon impérative, sinon il y a faute temporelle et l'application est déclarée non valide.
– **Temps de réponse** : cette caractéristique peut s'appliquer à une activité de régulation ou à un ensemble d'activités de régulation ; elle est directement liée à la dynamique du système. Ce paramètre correspond à la différence entre la date de réveil et la date de fin de l'activité.
– **Gigue temporelle** : ce paramètre caractérise la répétabilité d'une activité au fur et mesure de ses occurrences. En effet, entre deux exécutions successives d'une même activité, ses caractéristiques temporelles peuvent changer : date d'activation, durée d'exécution, temps de réponse, etc.

1.1.4 Quelques exemples d'applications

Nous pouvons citer quelques exemples d'applications de contrôle-commande :

– **Robot de production** : un robot, réalisant une activité spécifique (peinture, assemblage, tri) sur une chaîne de production, doit effectuer son travail en des temps fixés par la cadence de fabrication. S'il agit trop tôt ou trop tard, l'objet manufacturier traité sera détruit ou endommagé conduisant à des conséquences financières ou humaines graves (oubli d'un ou plusieurs rivets sur un avion).
– **Robot d'exploration** : ce robot doit se déplacer dans un environnement en principe non connu (zone radioactive après un accident, planète, épave sous la mer…). Il est important qu'il puisse réagir aux obstacles fixes ou mobiles afin de ne pas conduire à sa perte.

– **Téléphone mobile** : le système de contrôle-commande doit remplir plusieurs fonctions dont certaines ont des contraintes temporelles fortes pour avoir une bonne qualité de service (QoS : *Quality of Service*). Ainsi, la première fonction est de transmettre et de recevoir les signaux de la parole (577 μs de parole émises toutes les 4,6 ms et 577 μs de parole reçues toutes les 4,6 ms à des instants différents). En parallèle, il est nécessaire de localiser en permanence le relais le plus proche et donc de synchroniser les envois par rapport à cette distance (plus tôt si la distance augmente et plus tard si la distance diminue). Des messages de comptes rendus de la communication sont aussi émis avec une périodicité de plusieurs secondes. Les contraintes temporelles imposées au système doivent être imperceptibles à l'utilisateur.
– **Système de vidéoconférence** : ce système doit permettre l'émission et la réception d'images numérisées à une cadence de 20 à 25 images par seconde pour avoir une bonne qualité de service. Afin de minimiser le débit du réseau, une compression des images est effectuée. D'autre part la parole doit aussi être transmise. Bien que correspondant à un débit d'information moindre, la régularité de la transmission, qualifiée par une gigue temporelle, est nécessaire pour une reproduction correcte. De plus ce signal doit être synchronisé avec le flux d'images. L'ensemble de ces traitements (numérisations images et parole, transmission, réception, synchronisation…) sont réalisés en cascade, mais avec une cohérence précise.
– **Pilotage d'un procédé de fabrication** (fonderie, laminoir, four verrier…) : par exemple la fabrication d'une bobine d'aluminium (laminage à froid) exige un contrôle en temps réel de la qualité (épaisseur et planéité). Cette vérification en production de la planéité nécessite une analyse fréquentielle (FFT) qui induit un coût important de traitement. Le système doit donc réaliser l'acquisition d'un grand nombre de mesures (246 Ko/s) et traiter ces données (moyenne, FFT…) à la période de 4 ms. Ensuite, il affiche un compte rendu sur l'écran de l'opérateur toutes les 200 ms et enfin imprime ces résultats détaillés toutes les 2 s. Un fonctionnement non correct de ce système de contrôle de la qualité peut avoir des conséquences financières importantes : production non conforme à la spécification demandée.

1.2 Architecture des applications de contrôle-commande

1.2.1 Architecture logicielle des applications de contrôle-commande

■ Architecture multitâche

Le comportement concurrent des événements et grandeurs physiques externes amène à décrire l'environnement comme un système fortement parallèle. Cela conduit naturellement à adapter les méthodes de conception et de réalisation du système de contrôle-commande d'un tel environnement à ce parallélisme. Aussi, l'architecture la mieux adaptée pour répondre à ce comportement parallèle du procédé externe

© Dunod – La photocopie non autorisée est un délit.

est une **architecture multitâche**. Ainsi, au parallélisme de l'environnement, la réponse est le parallélisme de conception. Nous pouvons définir la tâche ou activité ou processus comme « une entité d'exécution et de structuration de l'application ». Cette architecture logicielle multitâche facilite la conception et la mise en œuvre et surtout augmente l'évolutivité de l'application réalisée.

D'une manière très générique, la figure 1.6 donne l'architecture logicielle d'une application de contrôle-commande multitâche. Nous pouvons ainsi découper cet ensemble de tâches ou activités selon les groupes suivants :

– Tâches d'entrées/sorties : ces tâches permettent d'accéder aux données externes par l'intermédiaire de cartes d'entrées/sorties et ensuite de capteurs et d'actionneurs directement liés au procédé géré. Ces tâches peuvent être activées de façon régulière ou par interruption.
– Tâches de traitement : ces tâches constituent le cœur de l'application. Elles intègrent des traitements de signaux (analyse spectrale, corrélation, traitement d'images, etc.) ou des lois de commande (régulation tout ou rien, régulation du premier ordre, régulation PID, etc.). Dans le cadre de cet ouvrage, nous considérerons ces tâches comme des boîtes noires, c'est-à-dire que le traitement effectué par ces tâches relève des domaines comme le traitement du signal, le traitement d'images ou l'automatique, disciplines qui débordent largement le contexte de ce livre.
– Tâches de gestion de l'interface utilisateur : ces tâches permettent de présenter l'état du procédé ou de sa gestion à l'utilisateur. En réponse, l'opérateur peut modifier les consignes données ou changer les commandes. Ces tâches peuvent être très complexes et coûteuses en temps de calcul si l'interface gérée est de taille importante (tableau de bord) ou de type graphique (représentation 3D).

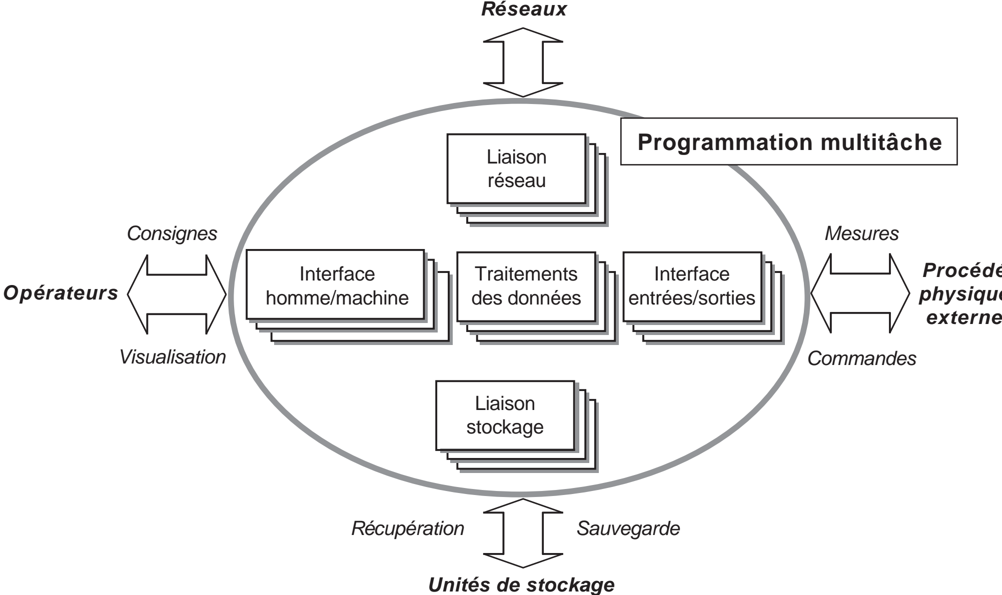


Figure 1.6 – Architecture logicielle d'une application de contrôle-commande multitâche.

– Tâches de communications : ces tâches sont destinées à gérer les messages envoyés ou reçus à travers un ou plusieurs réseaux ou bus de terrain. Si ce type de tâches existe, l'application est dite distribuée ou répartie.
– Tâches de sauvegarde : ces tâches permettent de stocker l'état du système à des instants fixés. Cette sauvegarde peut être utilisée *a posteriori* pour analyser le fonctionnement de l'application ou lors d'une reprise d'exécution à une étape précédente.

Après l'analyse et la conception de l'application, nous obtenons un ensemble de tâches ou activités qui coopèrent afin de réaliser le contrôle-commande du procédé géré. Ces tâches appartiennent aux différents groupes listés précédemment : tâches d'entrées/sorties, tâches de traitement, tâches de gestion de l'interface utilisateur, tâches de communications et tâches de sauvegarde. Ce découpage purement fonctionnel peut être modifié dans certains cas en utilisant une conception tournée vers les entités ou « objets » à contrôler. Cet aspect de la conception et de la mise en œuvre est présenté dans les chapitres suivants.
Les tâches obtenues, qui constituent l'application de contrôle-commande, ne sont pas des entités d'exécution indépendantes. En effet, certaines tâches sont connectées vers l'extérieur pour les entrées/sorties. De plus elles peuvent être liées par des relations de type (figure 1.7) :

– synchronisation : cela se traduit par une relation de précédence d'exécution entre les tâches ;
– communications : à la notion de précédence, traduite par la synchronisation, s'ajoute le transfert de données entre les tâches ;
– partage de ressources : les tâches utilisent des éléments mis en commun au niveau du système comme des zones mémoire, des cartes d'entrées/sorties, cartes réseau, etc. Certaines de ces ressources, comme par exemple les zones mémoire, ne sont pas ou ne doivent pas être accessibles, pour avoir un fonctionnement correct, par plus d'une tâche à la fois, elles sont dites **ressources critiques**.

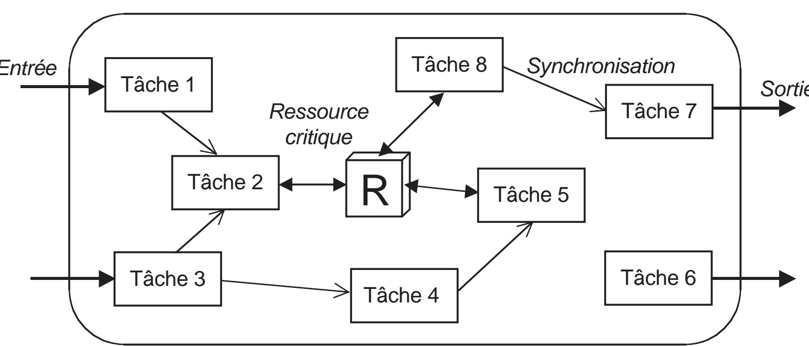


Figure 1.7 – Représentation schématique de l'architecture multitâche d'une application de contrôle-commande.

Ces différents concepts sont étudiés de façon détaillée dans le chapitre 4.

© Dunod – La photocopie non autorisée est un délit.

■ Modèles d'exécution et ordonnancement

Cette architecture logicielle peut être vue comme un ensemble de tâches synchronisées, communicantes et partageant des ressources critiques. Le rôle essentiel du système informatique est donc de gérer l'enchaînement et la concurrence des tâches en optimisant l'occupation du processeur, cette fonction est appelée l'**ordonnancement**. Ce principe d'ordonnancement est un point crucial des systèmes de contrôle-commande ; en effet l'ordonnancement va déterminer les caractéristiques temporelles et être le garant du respect des contraintes de temps imposées à l'exécution de l'application.

Nous pouvons distinguer deux modèles d'exécution de ces systèmes de contrôle-commande : l'exécution dite **synchrone** et l'exécution **asynchrone**. Nous allons présenter ces deux modèles d'exécution à l'aide d'un modèle d'application très simple. Cette application, constituée d'un ensemble de tâches pour gérer le procédé, intègre en particulier les deux tâches suivantes :

– Tâche de lecture des données entrées par l'opérateur à l'aide d'un clavier, appelée « Lecture_consigne ». L'intervention humaine fait que cette tâche peut être longue.

– Tâche d'alarme qui se déclenche sur un événement d'alerte correspondant au dépassement d'un paramètre critique, appelée « Alarme ». Celle-ci doit s'exécuter au plus vite pour éviter l'endommagement du procédé.

Pour mettre en avant les différences entre les deux modèles d'exécution, nous allons étudier la situation dans laquelle la tâche « Lecture_consigne » s'exécute et la tâche « Alarme » demande son exécution alors que la tâche « Lecture_consigne » n'est pas terminée.

Dans le modèle d'**exécution synchrone**, la perception de l'occurrence de tout événement par le système est différée du temps d'exécution de la tâche en cours. Dans l'exemple proposé, nous pouvons constater que la prise en compte d'un signal d'alerte n'est effective que lors de la fin de la tâche « Lecture_consigne » (figure 1.8). D'un point de vue du procédé, la réaction est perçue comme différée, alors que du point de vue du système informatique, elle est perçue comme immédiate. L'occurrence

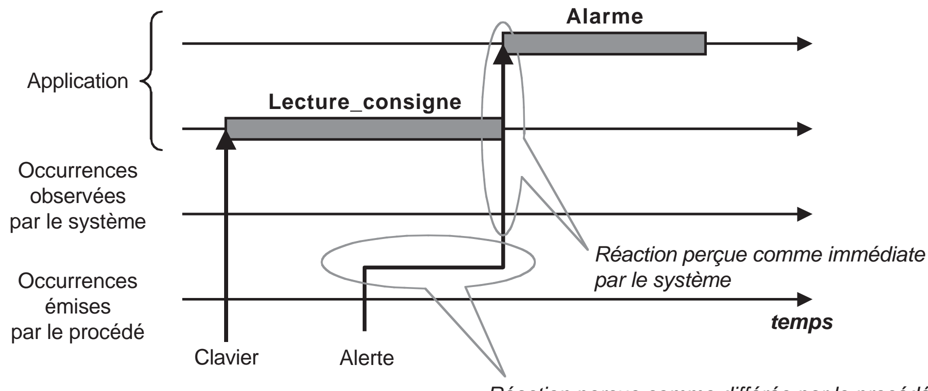


Figure 1.8 – Modèle d'exécution synchrone d'une application de contrôle-commande.

des événements externes a donc été artificiellement synchronisée avec le système informatique, d'où le nom d'exécution synchrone.

Ce retard peut affecter la prise en compte de n'importe quel événement, quelle qu'en soit la gravité pour l'application. Il faut donc vérifier que l'architecture opérationnelle choisie permettra de prendre en compte les contraintes temporelles : hypothèse de la fenêtre de visibilité des événements ou d'instantanéité des actions. La capacité du système à appréhender un événement externe est caractérisée par la durée de la tâche la plus longue puisque les tâches sont non interruptibles ou **non préemptibles**.

Dans le cas du modèle synchrone d'exécution, nous avons un système d'ordonnancement complètement prévisible et, en conséquence, il est possible en faisant une analyse exhaustive de l'exécution de produire une séquence d'exécution qui est jouée de façon répétitive. Cette étude de la séquence est appelée analyse de l'ordonnancement **hors ligne**. L'ordonnancement peut se réduire à un séquencement. Nous avons alors un environnement informatique très simple de l'application développée puisqu'il se réduit à une liste de tâches à exécuter. L'environnement informatique pour piloter cette liste de tâches se réduit à un système très simple : un **séquenceur**.

Dans le modèle d'**exécution asynchrone**, l'occurrence de tout événement est immédiatement prise en compte par le système pour tenir compte de l'urgence ou de l'importance. Dans l'exemple proposé, nous pouvons constater que la prise en compte d'un signal d'alerte est immédiate sans attendre la fin de la tâche « Lecture_consigne » (figure 1.9). La prise en compte de l'événement « alerte » est identique pour le procédé et le système informatique. L'occurrence des événements externes n'est pas synchronisée avec le système informatique, d'où le nom d'exécution asynchrone.

Dans ce contexte, nous avons des tâches qui sont interruptibles ou **préemptibles**. En conséquence, l'ordonnancement n'est pas totalement prévisible et l'analyse de l'exécution des tâches doit se faire **en ligne** par simulation ou par test. Cela nécessite l'utilisation d'un gestionnaire centralisé des événements et de la décision d'exécution : **exécutif ou noyau temps réel**.

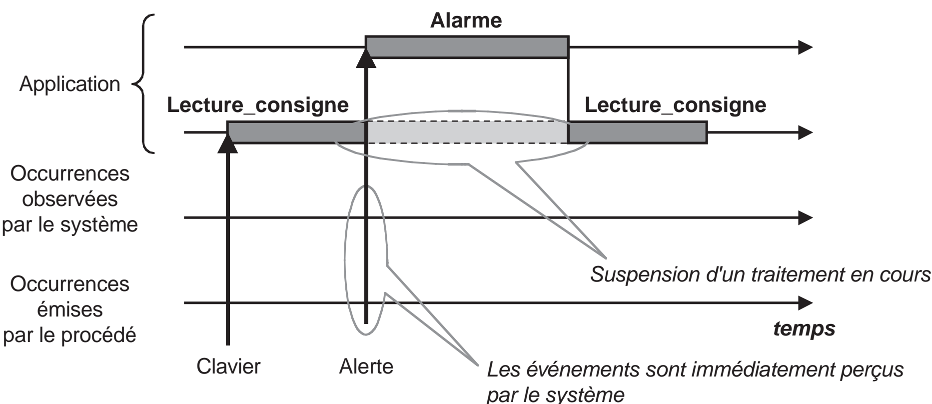


Figure 1.9 – Modèle d'exécution asynchrone d'une application de contrôle-commande.

© Dunod – La photocopie non autorisée est un délit.

Pour terminer cette section, nous allons rappeler trois définitions importantes que nous avons utilisées et fixer le contexte de cet ouvrage. Nous avons ainsi défini :

– Tâche non préemptible ou préemptible :
  • Une tâche non préemptible ne peut être interrompue qu'à des endroits spécifiques et à la demande de la tâche elle-même : fin_de_tâche, attente_signal… Dans ce cas, la programmation et plus simple et aucun mécanisme de partage de ressources critiques n'est à prévoir. En revanche, des temps de réponse longs peuvent se produire.
  • Une tâche préemptible peut être interrompue à n'importe quel instant et le processeur affecté à une autre tâche. Dans ce cas, les temps de réponse à un événement externe peuvent être très courts ; mais nous avons alors une programmation plus complexe avec un besoin de mécanisme de partage de ressources critiques.
– Analyse de l'ordonnancement hors ligne ou en ligne :
  • Une analyse de l'ordonnancement hors ligne correspond à la construction d'une séquence d'exécution complète sur la base des paramètres temporels des tâches en utilisant une modélisation (réseaux de Petri…) ou une simulation (animation ou énumération du modèle). L'ordonnanceur nécessaire est minimal puisque la séquence d'exécution est prédéfinie, il se réduit à un séquenceur. En revanche, l'application ainsi figée est peu flexible.
  • Une analyse de l'ordonnancement en ligne correspond à un choix dynamique de la prochaine tâche à exécuter en fonction des paramètres de la tâche en utilisant une modélisation de l'algorithme d'ordonnancement et une simulation de l'exécution. L'ordonnancement a un coût temporel non négligeable ; en revanche, l'application peut réagir à des événements ou des situations non prévus.
– Exécution synchrone ou asynchrone :
  • Une exécution est dite synchrone si les tâches sont non préemptibles et s'exécutent les unes après les autres dans un ordre qui peut être défini par une analyse hors ligne de l'ordonnancement.
  • Une exécution est dite asynchrone si les tâches sont préemptibles et s'exécutent selon l'ordonnancement. Une analyse de la séquence doit se faire obligatoirement en ligne.

Dans la suite de cet ouvrage, nous nous intéressons plus particulièrement aux systèmes asynchrones composés de tâches préemptibles avec un ordonnancement en ligne. Ainsi, l'architecture logicielle de l'application est composée de plusieurs tâches réalisées par le concepteur et d'un environnement spécifique, le noyau temps réel, que nous allons décrire. Le point essentiel de cet environnement est l'ordonnanceur qui permet d'affecter à tout instant le processeur à une tâche afin de respecter l'ensemble des contraintes temporelles attachées à la gestion du procédé.

■ Exécutif ou noyau temps réel

Cet environnement particulier d'exécution, exécutif ou noyau temps réel, peut être assimilé à un système d'exploitation de petite taille dédié aux applications de contrôle-commande. La caractéristique fondamentale est son déterminisme d'exé-

cution avec des paramètres temporels fixés (temps de prise en compte d'une interruption, changement de contexte entre deux tâches, etc.). Nous pouvons comparer les différences au niveau des objectifs fixés pour le noyau d'exécution d'un système informatique classique et d'un système informatique de contrôle-commande.

Un système classique n'a pas été conçu pour permettre de respecter des contraintes temporelles, mais il suit les règles suivantes :

– politiques d'ordonnancement des activités basées sur le partage équitable du processeur : affectation identique du temps processeur à tous les processus en cours ;
– gestion non optimisée des interruptions ;
– mécanismes de gestion mémoire (cache…) et de micro-exécution engendrant des fluctuations temporelles (difficulté pour déterminer précisément les durées des tâches) ;
– gestion des temporisateurs ou de l'horloge pas assez fine (plusieurs millisecondes) ;
– concurrence de l'application temps réel avec le système d'exploitation toujours actif ;
– gestion globale basée sur l'optimisation d'utilisation des ressources et du temps de réponse moyen des différents processus en cours.

Un système informatique de contrôle-commande s'attache aux caractéristiques suivantes :

– efficacité de l'algorithme d'ordonnancement avec une complexité limitée ;
– respect des contraintes de temps (échéances…). Ces contraintes temporelles se traduisent plus en termes de choix d'une activité à exécuter à un instant donné plutôt que de rapidité d'exécution de toutes les activités ;
– prédictibilité (répétitivité des exécutions dans des contextes identiques) ;
– capacité à supporter les surcharges ;
– possibilité de certification pour les applications de certains domaines comme l'avionique, l'automobile…

> En général, contrairement à un noyau temps réel, les contraintes temporelles ne sont pas garanties dans un système d'exploitation classique (Unix, Windows NT…).

Une application temps réel étant par définition un système multitâche, le rôle essentiel du noyau temps réel est donc de gérer l'enchaînement et la concurrence des tâches en optimisant l'occupation de l'unité centrale du système informatique. Les principales fonctions d'un noyau temps réel peuvent être scindées en trois groupes :

1. gestion des entrées/sorties (gestion des interruptions, gestion des interfaces d'entrées/sorties, gestion des réseaux de communications…) ;

2. ordonnancement des tâches (orchestration du fonctionnement normal, surveillance, changements de mode, traitement des surcharges…) ;

3. relations entre les tâches (synchronisation, communication, accès à une ressource critique en exclusion mutuelle, gestion du temps…).

© Dunod – La photocopie non autorisée est un délit.

Il important de noter que les tâches sont les unités actives du système ; le noyau temps réel n'est actif que lors de son appel. Une tâche activée peut appeler le noyau temps réel par une **requête**. Les différentes requêtes sont servies par des modules du noyau temps réel appelées **primitives**. Ensuite le noyau temps réel réactive une tâche de l'application selon l'algorithme d'ordonnancement utilisé (figure 1.10). Ainsi, le noyau temps réel centralise toutes les demandes d'activation des tâches et gère des tables lui permettant de comparer les priorités (ou les urgences) et l'état de ces diverses tâches, ainsi que l'état d'occupation des ressources. La décision d'activation d'une tâche étant prise, le noyau temps réel lance les modules de programmes correspondant à cette tâche et lui alloue les ressources disponibles. La tâche activée occupe le processeur jusqu'à la fin de son exécution sous le respect des conditions suivantes :

- Elle ne réalise pas d'opérations d'entrées-sorties.
- Les ressources utilisées sont disponibles.
- Aucun événement extérieur ne revendique le déroulement d'une tâche plus prioritaire.

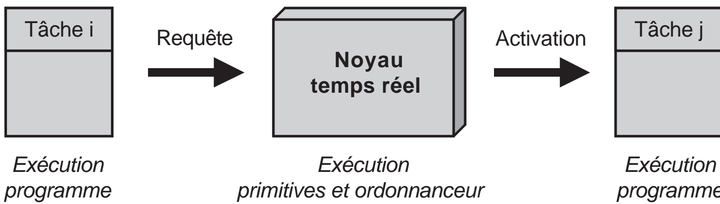


Figure 1.10 – Interaction entre les tâches et le noyau temps réel.

Nous pouvons donc décrire schématiquement le contexte complet d'exécution d'une application temps réel avec les deux parties : tâches et noyau temps réel (figure 1.11). En conclusion de cette section sur l'ordonnancement qui est étudié de façon plus complète dans le chapitre 8, l'ordonnancement dans le cas des systèmes temps réel à contraintes temporelles strictes a pour objectif principal de répondre aux deux cas suivants :

- Fautes temporelles : cela correspond à un non respect d'une contrainte temporelle associée à une tâche comme le dépassement de la date limite d'exécution ou échéance. Cela induit la notion d'**urgence** d'une tâche.
- Surcharge : lors de l'occurrence d'une ou plusieurs fautes temporelles, l'ordonnanceur peut réagir en supprimant une ou plusieurs tâches de l'application, ce qui amène à la notion d'**importance**, c'est-à-dire le choix d'une tâche à exécuter par rapport aux spécifications fonctionnelles de l'application.

■ Implémentation des applications de contrôle-commande

Comme nous le verrons au cours de cet ouvrage, les langages de développement des applications de contrôle-commande sont très divers. Mais, par rapport à l'environnement d'exécution que nous venons de décrire (noyau temps réel avec les trois

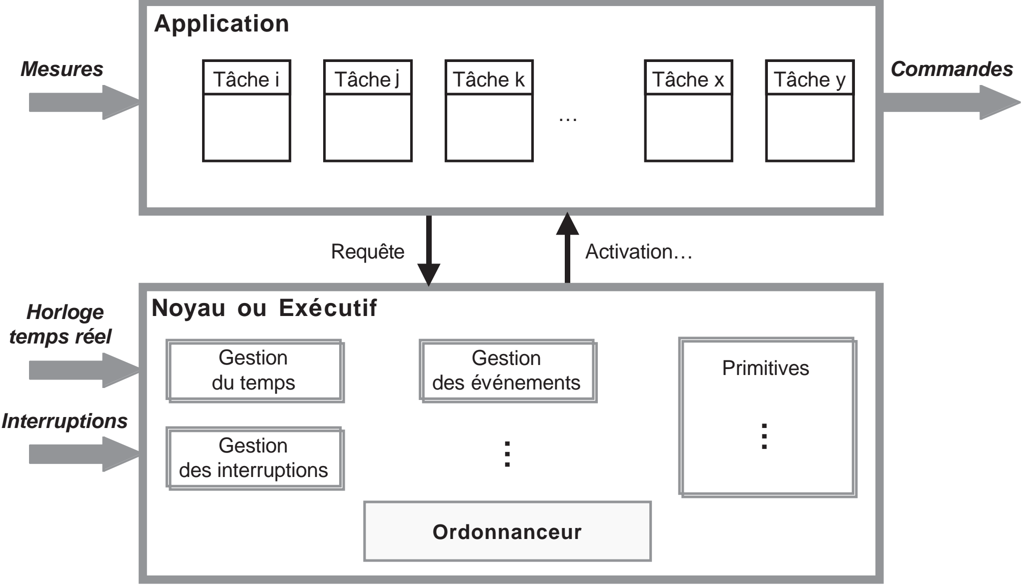


Figure 1.11 – Architecture de l'application : tâches et noyau temps réel.

fonctions décrites : 1. gestion des interruptions, 2. ordonnancement, 3. relations entre les tâches), il est possible de décliner les langages en trois groupes (figure 1.12) :

– langages standards (langage C…) : le noyau temps réel qui supporte ce type de langage doit être complet puisque le langage n'intègre aucune spécificité de ce domaine de contrôle-commande multitâche ;

– langages multitâches (langage Ada…) : ces langages permettent de décrire l'application en termes de tâches ; ainsi le noyau peut être plus réduit et ne comporter que les deux premières fonctions ;

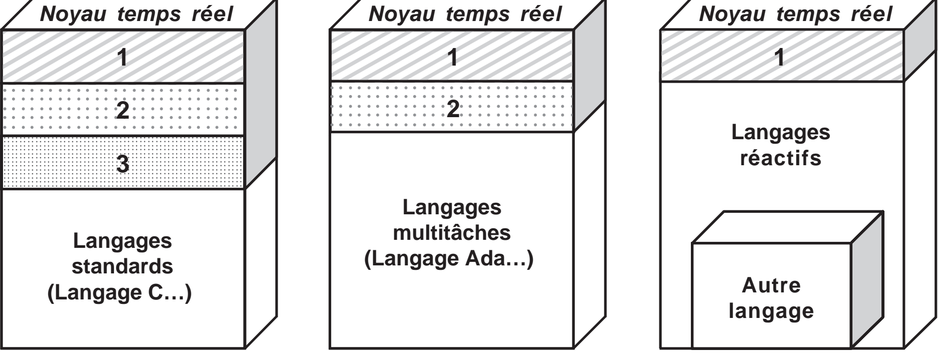


Figure 1.12 – Langages utilisés pour développer les applications de contrôle-commande avec un noyau temps réel (1. gestion des interruptions, 2. ordonnancement, 3. relations entre les tâches).

© Dunod – La photocopie non autorisée est un délit.

– langages réactifs (langages Lustre, Esterel, Signal…) : ces langages donnent non seulement la possibilité de décrire les fonctionnalités du programme, mais aussi l'enchaînement des différentes parties. Le noyau est donc limité à une couche proche du matériel lié notamment à la gestion des interruptions. En revanche, étant donné la possibilité très limitée d'expression de l'aspect fonctionnel, ils sont souvent associés à un langage standard pour palier ce manque.

1.2.2 Architecture matérielle des applications de contrôle-commande

Comme nous l'avons vu en introduction, l'aspect matériel a une très grande importance dans les applications de contrôle-commande. Cette implication est liée d'une part à la connexion directe avec le monde physique réel à l'aide d'une grande diversité de systèmes d'entrées/sorties et d'autre part au matériel informatique parfois spécifique et développé pour une application donnée. Ce dernier point concerne les applications dites dédiées et embarquées ; le matériel a été conçu et créé spécifiquement pour une application, comme un téléphone portable, une caméra vidéo, etc. À titre d'exemple, les différents calculateurs embarqués dans un véhicule automobile sont dédiés et spécialement développés pour cette application (figure 1.13). Dans ces matériels, nous trouvons un processeur de type microcontrôleur redondé pour avoir un haut niveau de sécurité, des composants spécifiques (ASIC : *Application Specific Integrated Circuit*), une alimentation électrique… Tout ce matériel est ensuite encapsulé dans un boîtier résistant à l'environnement de fonctionnement usuel (chaleur, vibrations, ambiance corrosive, etc.).
Beaucoup d'applications de contrôle-commande s'exécutent sur des plateformes PC classiques. En revanche, il est toujours nécessaire de disposer de cartes d'entrées/sorties qui doivent souvent être synchronisées (acquisition et commande effectuées à des instants précis). Ainsi, dans ce domaine d'applications informatiques, des matériels spécifiques existent et permettent ce fonctionnement temporel : c'est par exemple le cas de la plateforme PXI concaténation d'une plateforme classique PC et de cartes d'entrées/sorties synchronisées.
Enfin, dans de nombreux cas, les applications de contrôle-commande sont de type distribué, c'est-à-dire qu'elles sont composées de plusieurs sites ou processeurs, sur lesquels s'exécutent un environnement multitâche, reliés par un ou des réseaux informatiques (Ethernet, CAN, FIP…).
L'ensemble de ces aspects matériels est traité en détail dans le chapitre 4.

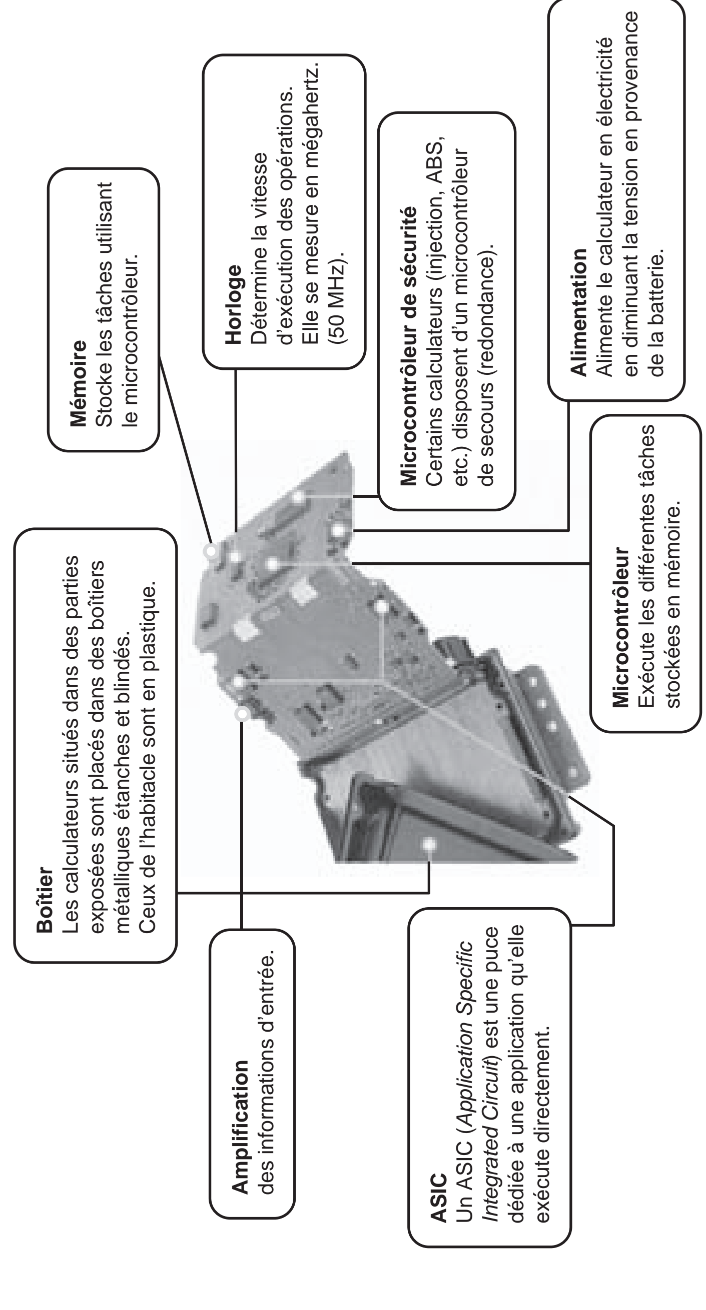


Figure 1.13 – Matériel dédié utilisé pour implémenter une application de contrôle-commande du domaine de l'automobile.

© Dunod – La photocopie non autorisée est un délit.

1.3 Développement des applications de contrôle-commande

Le développement des applications informatiques demande de plus en plus de rigueur dans le suivi des différentes étapes de spécification, de conception et de codage. Ce cycle de développement permet ainsi d'obtenir des applications de très bonne qualité d'un point de vue architecture logicielle, d'augmenter la maintenabilité et l'évolutivité. En particulier, cette rigueur de développement accroît de façon significative la correction des programmes en suivant une démarche de tests unitaires et d'intégration. Si ces tests, qui sont un point primordial dans l'obtention d'une qualité logicielle, sont aisés à réaliser dans le cas d'applications informatique classiques, en revanche, dans le cas des applications de contrôle-commande, les tests opérationnels en exécution réelle sont souvent difficiles à produire à cause de diverses particularités :

- exécution unique : satellite d'exploration ;
- coût très élevé : fusée… ;
- risques humains : avion…

Ainsi, malgré des phases de tests souvent coûteuses et conséquentes, de nombreuses applications de contrôle-commande n'ont pas rempli les objectifs fixés. Nous pouvons citer quelques exemples connus :

- Mission Vénus : le satellite d'exploration est passé à plus de 500 000 km de la planète Venus au lieu de 5 000 km, prévu initialement. Cet échec a été attribué à un simple remplacement d'une virgule par un point dans un programme Fortran (« DO 20 I = 1. 5 » au lieu de « DO 20 I = 1, 5 »).
- Avion militaire américain F16 : lors des premiers essais en vol, l'avion était déclaré sur le « dos » au passage de l'équateur à la très grande surprise du pilote. Cela était simplement dû à une erreur de signe dans le programme.
- Navette spatiale américaine : lors du premier lancement de la navette, le départ a été annulé et la mission reculée de trois jours (coût très important). Ce faux départ était dû à une erreur de synchronisation entre les deux ordinateurs de conduite de vol. Le fonctionnement en redondance de ces ordinateurs conduisait à un test de cohérence de certaines grandeurs physiques. Étant donné une désynchronisation des deux ordinateurs, ce test a été négatif simplement à cause de la mesure du même paramètre effectuée à des instants différents.
- Mission sur Mars : lors de la mission d'exploration de la planète Mars par le robot Pathfinder, une remise à zéro périodique des données acquises a fortement perturbé la mission. Ce problème était lié à un blocage d'une tâche très prioritaire par une tâche moins prioritaire mais détenant une ressource critique (réseau de communication vers la terre). En particulier les données météorologiques mesurées étaient très spécifiques d'un point de vue « durée et taille » du fait des caractéristiques martiennes.
- Fusée Ariane V : lors du premier lancement, la fusée a dû être détruite à cause d'une trajectoire non correcte. Cette erreur était liée à la réutilisation de certains

modules logiciels utilisés dans le contexte d'Ariane IV. Les spécifications, attachées à l'accélération, auraient dû être différentes en termes de limites afin d'éviter ce dysfonctionnement.

Cette liste d'exemples de problèmes au niveau de l'exécution d'applications de contrôle-commande montre la nécessité de mettre en place un cycle de développement encore plus rigoureux pour ces applications de gestion de procédé physique dont les tests en exécution réelle ne sont pas toujours facilement accessibles.

1.3.1 Cycle de développement des applications informatiques

Le cycle de développement des applications informatiques suit les trois étapes classiques que sont la spécification, la conception et la programmation. L'analyse des besoins permet d'écrire le cahier des charges de l'application. À partir de ce cahier des charges, le déroulement des trois étapes conduit successivement aux descriptions suivantes de l'application (figure 1.14) :

– **spécification globale** : description de « ce que l'on a à faire » ou le « quoi », c'est-à-dire une mise en forme plus ou moins formelle du « cahier des charges » ;
– **conceptions préliminaire et détaillée** : description du « comment », c'est-à-dire une modélisation de l'architecture logicielle de l'application (modules, tâches, objets…). Il peut être nécessaire d'employer différents niveaux de raffinement selon la taille de l'application ;
– **programmation** : traduction dans un langage exécutable de l'architecture logicielle de l'application décrite précédemment. Suivant la méthode de conception employée et le niveau de raffinement, la traduction dans un langage de programmation peut être plus ou moins automatisée.

Ces différentes étapes peuvent être plus ou moins formelles selon les méthodes employées. Aussi, à chaque étape, il est primordial de vérifier la cohérence de la description ou de la traduction réalisée à partir de l'étape précédente. Ce travail concerne la validation. Celle-ci constitue une preuve si la méthode est formelle ou un test si elle ne l'est pas.

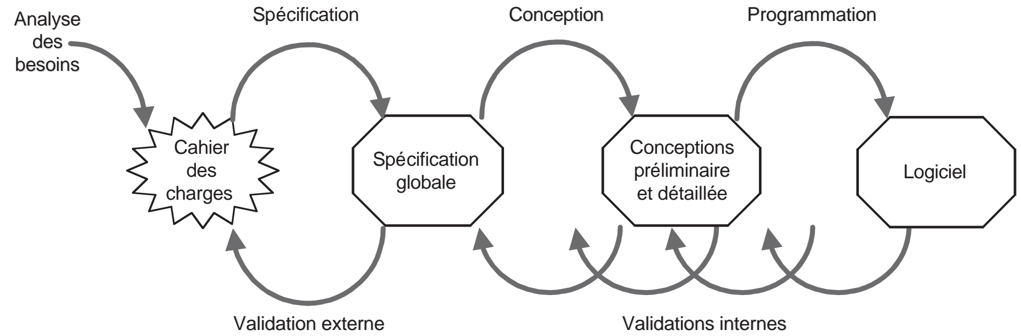


Figure 1.14 – Cycle de développement d'une application informatique classique.

© Dunod – La photocopie non autorisée est un délit.

Une présentation généralement plus adaptée est celle du cycle en « **V** » (figure 1.15). Dans cette représentation, à chaque niveau d'analyse ou de conception correspond un niveau de validation ; ainsi, nous avons :

– conception détaillée et tests unitaires ;
– conception préliminaire et tests d'intégration ;
– spécification et validation globale.

Il est évident que cette formalisation méthodologique du développement des applications informatiques a pour principaux objectifs : éviter les fautes logicielles, accroître la maintenabilité, faciliter l'évolutivité… À chaque étape ou ensemble d'étapes correspond une méthode qui est généralement supportée par un outil informatique pour aider à sa mise en œuvre plus ou moins automatisée (*CASE Tools : Computer Aided Software Engineering Tools*).

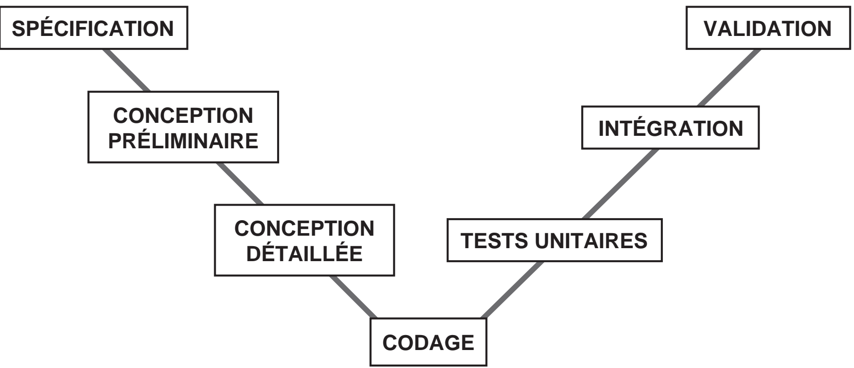


Figure 1.15 – Cycle de développement en « **V** » d'une application informatique classique.

L'expérience du développement de logiciels prouve que l'élaboration complète de l'application ne se fait pas en une seule fois : évolution du cahier des charges, modifications du découpage modulaire, correction de programmes, etc. Cela a induit de nouveaux schémas de développement, appelés itératifs ou en spirale, qui consistent à prendre en compte les passages successifs dans les différentes étapes du développement. L'idée forte à retenir est que, lors de toutes modifications apportées à l'application à quelque niveau que ce soit, il est nécessaire de décliner à nouveau toutes les étapes du développement.

1.3.2 Développement couplé matériel et logiciel

Avant de décrire les différentes méthodes utilisées dans le cadre des applications de contrôle-commande et leurs spécificités, il est important de remarquer la particularité du développement de ces applications quant au caractère du couplage fort entre les aspects matériel et logiciel.

En effet, le cahier des charges d'une application de contrôle-commande d'un procédé va intégrer dans la grande majorité des cas à la fois la description du matériel et

les fonctions à remplir par ce procédé (figure 1.16). Ainsi, la spécification de l'application commence par une spécification système (matériel et logiciel). Une partie de la conception préliminaire peut aussi intégrer les deux aspects puisque la réalisation d'un module fonctionnelle peut être réalisée soit de manière matérielle (circuit intégré spécifique – FPLA : *Field Programmable Logic Array* ou FPGA : *Field Programmable Gate Array*) soit d'une manière logicielle. Ensuite, le développement des deux parties peut continuer de façon différenciée et classique avec la conception détaillée, l'implémentation (réalisation ou codage) et les tests. À nouveau, les deux aspects de l'application doivent se rejoindre pour passer à la phase d'intégration et de validation de l'ensemble. La phase d'intégration est certainement l'étape la plus importante et la plus difficile. En effet, une partie logicielle va être insérée dans un environnement matériel très spécifique.

Bien que ce ne soit pas le propos de cet ouvrage, l'aspect matériel est abordé de façon succincte dans le chapitre 4. Au niveau de la conception, il existe aussi de nombreuses méthodes permettant d'avoir des développements de qualité. Ainsi, le langage VHDL (*VHSIC Hardware Description Language* – VHSIC : *Very High Speed Integrated Circuit*) de description des fonctions à réaliser d'un point de vue matériel conduit à une conception formelle des circuits, c'est-à-dire autorisant une preuve de la conception établie.

Prenons un exemple très répandu comme les consoles de jeux portables. Ces matériels possèdent une interface utilisateur très spécifique comportant un ensemble de

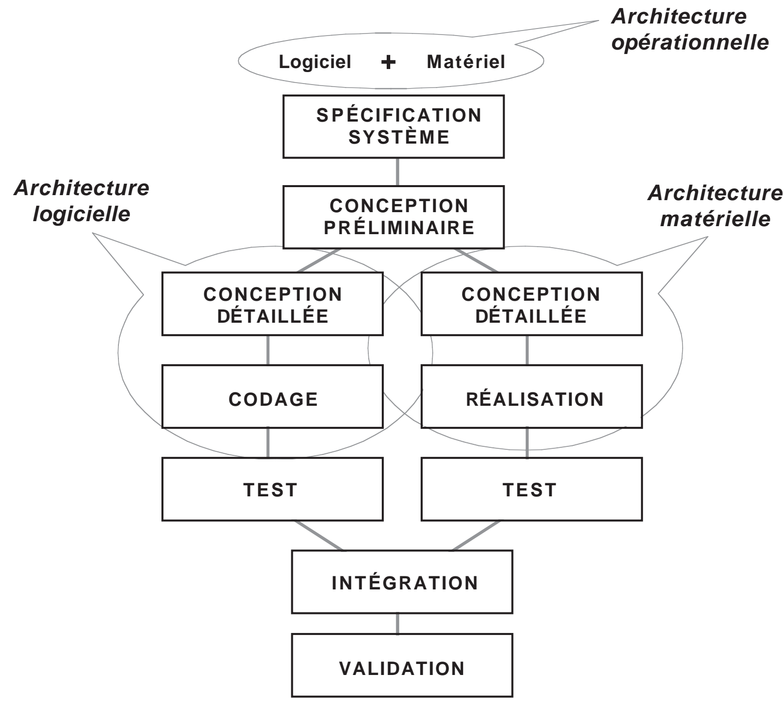


Figure 1.16 – Cycle de développement matériel et logiciel d'une application de contrôle-commande de procédé.

© Dunod – La photocopie non autorisée est un délit.

boutons, un haut-parleur et un écran couleur. De plus, ces consoles intègrent généralement une liaison vers l'extérieur de type infrarouge ou filaire pour une connexion avec une autre console de la même gamme. À ces entrées/sorties très dédiées, s'ajoutent des contraintes de réalisation liées à la caractéristique embarquée de l'application : alimentation sur batterie, autonomie importante, encombrement réduit, ergonomie, design du boîtier, réalisation en très grande quantité, coût le plus bas possible… Toutes ces contraintes vont avoir un impact important sur le développement aussi bien matériel que logiciel. Ainsi, le processeur est de type microcontrôleur à faible consommation intégrant des entrées/sorties multiples auquel s'ajoute un processeur graphique pour gérer l'écran couleur. Le noyau temps réel implanté doit posséder les caractéristiques suivantes : petite taille (taille mémoire limitée), rapidité d'exécution (ensembles de primitives simples), coût faible (réalisation en très grande quantité)… Ces différentes spécifications, qui lient la réalisation matérielle à l'implémentation logicielle, doivent être prises en compte au début de l'analyse de l'application.
Aussi, nous pouvons résumer l'environnement de développement d'une application de contrôle-commande par le schéma de la figure 1.17. Les spécifications du système sont donc élargies ; en plus des aspects fonctionnels et comportementaux classiques pour ce type d'application, nous devons ajouter les contraintes de développement évoquées précédemment, soit :

– contraintes matérielles : type de processeur, architecture (distribué, multiprocesseur…), taille mémoire, dimension physique, consommation, environnement (température, pression, corrosion…) ;
– noyau temps réel : primitives, taille (micronoyau…), certifié…

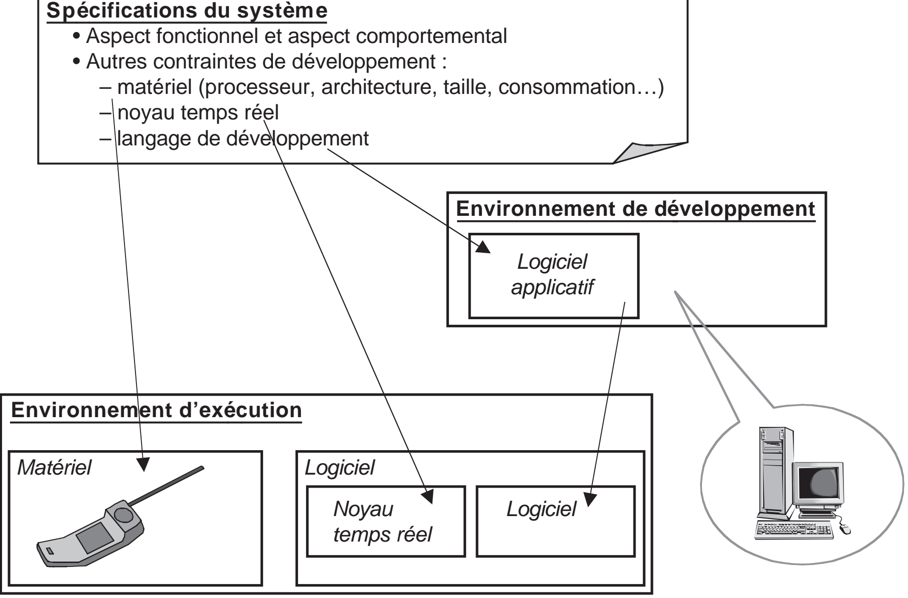


Figure 1.17 – Environnement spécifique du développement d'une application de contrôle-commande de procédé.

– langages de développement : assembleurs, langage de haut niveau, langage formel…

Ensuite, il est important de noter que le développement de la partie logicielle d'une application de contrôle-commande va être réalisé sur une plate-forme dite « **hôte** » qui n'a aucun rapport avec l'environnement d'exécution ou environnement « **cible** » en termes de processeur, mémoire, système d'exploitation, etc. Lorsque le logiciel applicatif est réalisé et testé autant que faire se peut sur cette plate-forme « hôte », le programme est compilé dans le code du processeur « cible » par un compilateur croisé ; puis il est téléchargé avec le noyau temps réel choisi vers l'architecture matérielle « cible ». De nouveau, des tests doivent être réalisés dans cet environnement d'exécution. En effet, le comportement du programme dans cette architecture « cible » de l'application peut être différent et amener à des modifications conséquentes du programme. Ce processus conduit à modifier le cycle en « V » de développement des applications informatiques classiques par un cycle en « W » où la deuxième partie du cycle correspond à la reprise de la première partie du cycle mais dans l'environnement « cible » (figure 1.18). Nous trouvons en particulier dans ce cycle en « W » un codage croisé et l'intégration avec le noyau.

Ce constat de la dualité de développement des applications de contrôle-commande de procédé amène à plusieurs remarques concernant les environnements permettant d'élaborer ce type d'application :

– le noyau temps réel choisi doit être adapté à l'architecture « cible » de l'application en termes de codage, de taille et de fonctionnalités (primitives, gestion des entrées, gestion du temps…) ;

– l'environnement de développement sur la plate-forme « hôte » doit posséder un émulateur du noyau temps réel afin de pouvoir faire les premiers tests du logiciel multitâche réalisé ;

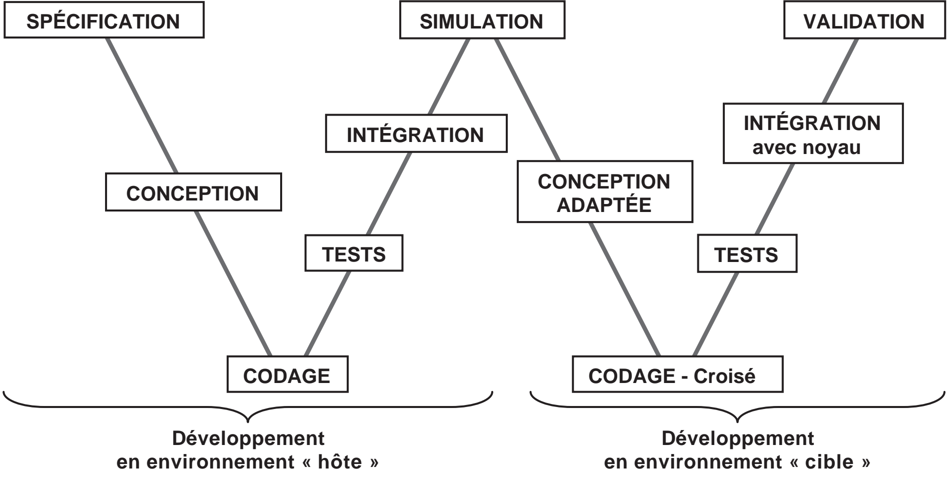


Figure 1.18 – Cycle de développement en « **W** » d'une application de contrôle-commande de procédé.

© Dunod – La photocopie non autorisée est un délit.

– l'environnement de développement sur la plate-forme « hôte » doit pouvoir faire une compilation croisée dans le code du processeur de l'architecture « cible » ;
– l'environnement de développement sur la plate-forme « hôte » doit permettre un « debug » de l'application lors de son exécution sur l'architecture « cible ». Cette observation de l'exécution se fait à distance par une liaison réseau quelconque (Ethernet…). De nombreux environnements proposent une représentation graphique de l'exécution des tâches. La plus grande difficulté réside dans le fait de ne pas modifier l'exécution de l'application par cette observation.

Toutes ces remarques impliquent dans le choix d'un environnement de développement de ces applications de prendre en compte l'ensemble des caractéristiques suivantes :

– environnement « cible » (microprocesseurs, architecture…) ;
– environnement « hôte » (type de système d'exploitation) ;
– conformité à une norme ou pseudo-norme (POSIX, projet Sceptre) ;
– compacité (pour les applications embarquées) ;
– outils d'aide au développement (« debug », analyse en ligne…) ;
– primitives temps réel (liste de tous les services fournis) ;
– caractéristiques de l'ordonnanceur (politiques d'ordonnancement) ;
– caractéristiques temporelles :
  • temps de masquage des interruptions (*interrupt latency*), temps pendant lequel les interruptions sont masquées et ne peuvent donc pas être prises en compte,
  • temps de réponse (*task response time*) : temps entre l'occurrence d'une interruption et l'exécution de la tâche réveillée.

L'ensemble de ces notions concernant le noyau temps réel et son choix pour une application donnée est abordé dans le chapitre 5.

1.3.3 Cycle de développement des applications de contrôle-commande

Il existe de nombreuses méthodes appliquées au développement logiciel des applications de contrôle-commande de procédé. Ces méthodes couvrent une ou plusieurs étapes du cycle de développement selon les niveaux de raffinement où elles sont utilisées. Sans vouloir faire un exposé exhaustif de l'existant, il est intéressant de citer quelques méthodes en les différenciant par leur concept de base. Nous trouvons ainsi des méthodes fonctionnelles et structurées, dites aussi à flots de données, qui sont fondées sur le principe du découpage fonctionnel de l'application. Ces éléments ou modules fonctionnels sont appliqués aux données qui se propagent de fonction en fonction. Le deuxième ensemble de méthodes est celui des méthodes basées sur des modèles de machines à états. Ces méthodes reposent sur des bases formelles et permettent en général des vérifications plus avancées que les précédentes. La troisième catégorie, plus récente, de méthodes est dite orientée objets ou objets. Nous pouvons résumer ces dernières en disant que ces méthodes mettent en avant les données et leur structuration. Nous pouvons donner les exemples suivants dans chacun de ces ensembles :

– Méthodes fonctionnelles structurées :
  - JSD : *Jackson System Design* (Michaël Jackson, 1981)
  - SA_RT : *Structured Analysis Real Time* (Ward-Mellor, 1984 ; Pirbhaï-Hatley, 1986)
  - DARTS : *Design Approach for Real-Time Systems* (Gomaa, 1984)
  - SDL : *Specification and Description Language* (CCITT, 1988)
  - MSMC : *Modélisation Simulation des Machines Cybernétiques* (Brenier, 2001)

– Méthodes basées sur les machines à états :
  - Réseaux de Petri (Petri, 1962)
  - GRAFCET : Graphe Fonctionnel de Commande Étape Transition (IEC 1988)
  - Statecharts : (D. Harel, 1987)
  - Langages réactifs synchrones : Lustre (Caspi, 1991), Esterel (Berry, 1991), Signal (le Guernic, 1991)

– Méthodes objets ou orientés objets :
  - UML : *Unified Modeling Language* (OMG, 1995)
  - HOOD : *Hiearchical Object Oriented Design* (CRI-Cisi Ingénierie-Matra, 1987)

La figure 1.19 reprend le cycle en « V » de développement avec le positionnement de quelques-unes de ces méthodes dans ce cycle.

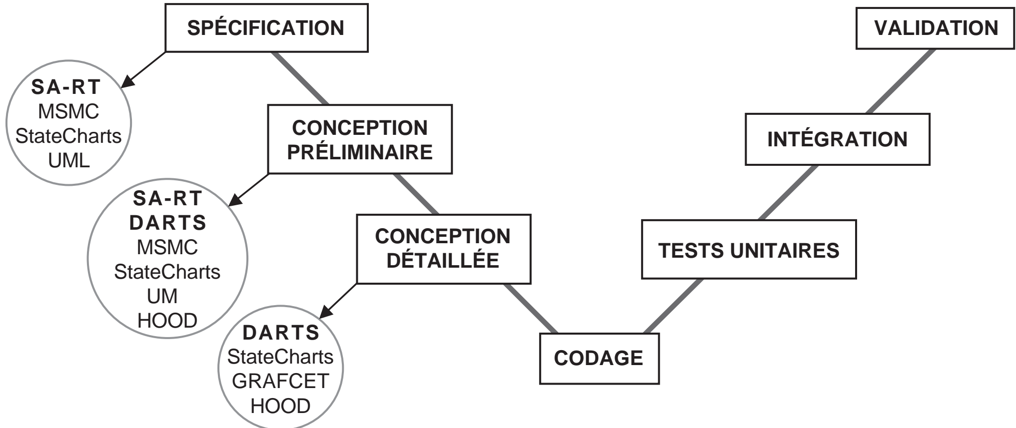


Figure 1.19 – Quelques méthodes de développement d'une application de contrôle-commande de procédé.

Comme cela a été justifié et expliqué dans l'avant-propos, les méthodes SA-RT et DARTS, qui permettent de décrire complètement le cycle dans ses phases de spécification et de conception, sont celles qui sont étudiées de façon détaillée dans cet ouvrage. En effet, dans le cas d'applications embarquées de taille petite ou moyenne qui sont implémentées dans une architecture multitâche, il semble plus pertinent d'utiliser une analyse et une conception de type fonctionnel et structuré.

© Dunod – La photocopie non autorisée est un délit.

1.3.4 Quelques exemples industriels d'applications de contrôle-commande

Afin d'illustrer les différents environnements de développements des applications de contrôle-commande, nous présentons des exemples industriels issus de programmes de taille importante des années 1990-2000. Ces exemples sont caractérisés par le nom du programme, la ou les sociétés en charge du programme et les méthodes et langages utilisés, soit :

– Programme **Spot 4** (Matra Marconi Space/CNES) : satellite destiné à une observation de la terre (météorologie, environnement, agriculture…)
  • Spécifications et conceptions : HOOD
  • Langages : Ada, Assembleur
– Programme **Ariane 5** (Aérospatiale/CNES) : lanceur
  • Spécifications et conceptions : HOOD
  • Langages : Ada, noyau temps réel ARTK, Assembleur (Motorola 68020)
– Programme **ISO** – *Infrared Space Observatory* (Aérospatiale/ESA) : ensemble de satellites destinés à une observation de l'espace dans un domaine infrarouge
  • Spécifications et conceptions : SART et HOOD
  • Langages : Ada (15 000 lignes), Assembleur (11 000 lignes)
– Programme **SENIT8** (Dassault Électronique & DCN-Ingénierie) : équipements de gestion et de contrôle-commande du porte-avions Charles de Gaulle
  • Spécifications et conceptions : SART et Ada-Buhr (proche de la méthode DARTS)
  • Langages : Ada (1 000 000 lignes), C (400 000 lignes)
– Programme **Rafale** (Dassault Électronique) : avion militaire
  • Spécifications et conceptions : SA-RT et OMT
  • Langages : Ada (800 000 lignes à 1 500 000 lignes selon les versions).

Nous pouvons remarquer que les environnements de développement intègrent une analyse fonctionnelle et structurée avec SA-RT et une conception orientée objet. Cela est dû essentiellement soit à des obligations du cahier des charges (applications spatiales) soit à la taille de l'application qui justifie une méthode orientée objet.

2 • SPÉCIFICATION SELON LA MÉTHODE SA-RT

2.1 Introduction générale à la méthode SA-RT

La méthode SA-RT est une méthode d'analyse fonctionnelle et opérationnelle des applications de contrôle-commande. Cette méthode permet de réaliser une description graphique et textuelle de l'application en termes de besoins, c'est-à-dire de « ce que l'on a à faire » ou le « quoi » (*What* ?). Cette mise en forme du « cahier des charges » de l'application est formelle dans le sens où la méthodologie (ensemble des documents à élaborer) et l'expression (syntaxe graphique) sont définies. En revanche, elle ne permet pas d'effectuer une vérification de propriétés de l'application à partir des seules descriptions SA-RT. Des études ont été menées pour associer à la méthode SA-RT des méthodes formelles afin d'apporter des possibilités de simulation et de vérification. Une de ces méthodes est présentée à la fin du chapitre. Aucune règle officielle ou normalisation n'a été mise en place pour la méthode SA-RT et son utilisation. Par conséquent, il existe de nombreuses mises en œuvre de la méthode SA-RT avec des différences plus ou moins importantes et aussi des extensions spécifiques de la méthode. Ceci fera l'objet du dernier point traité dans ce chapitre. Nous allons nous attacher à décrire la méthode SA-RT la plus générale et la plus usitée, et correspondant à la méthodologie de développement d'une application de contrôle-commande qui est l'objectif de cet ouvrage.

L'accroissement très important de la taille des logiciels développés dans les années 70 a conduit à mettre en place des méthodes d'analyse et de conception permettant une meilleure réalisation et aussi une maintenance plus efficace dans l'exploitation des logiciels. Le mot essentiel de ces méthodes est « structuration » dans le sens d'une décomposition en éléments ou blocs fonctionnels pour un niveau d'analyse donné et d'une décomposition hiérarchique cohérente entre les différents niveaux d'analyse. Ces méthodes d'analyse ou de conception structurées conduisent naturellement à la programmation structurée. La deuxième particularité commune à ces méthodes est la description sous forme de flux ou flots de données, de contrôle ou autres. L'aspect opérationnel de la description est alors visualisé par la propagation de ces flux.

Ainsi, nous trouvons la méthode SA-DT (*Structured Analysis Design Technics*) de spécification d'un système qui permet d'exprimer un bloc représentant soit les activités (fonctions) soit les données. Les flots entrants sont les données, un contrôle ou des mécanismes (méthodes) et les flots sortants correspondent aux sorties de

© Dunod – La photocopie non autorisée est un délit.

données. Cette méthode très générale de description d'un système a été adaptée à la spécification de logiciels avec la méthode très connue SA (*Structured Analysis*) (figure 2.1).

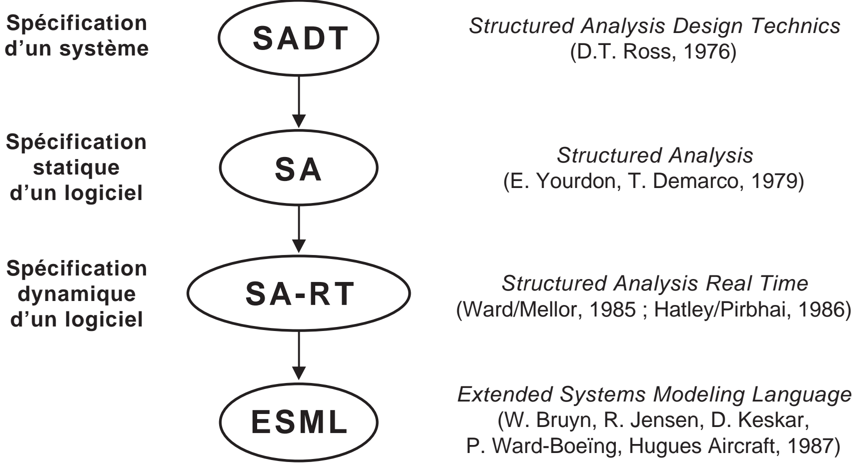


Figure 2.1 – Positionnement chronologique de la méthode SA-RT.

L'analyse structurée SA, définie par E. Yourdon et T. Demarco, est une méthode descendante par affinages successifs des traitements, appelés « process ». Les différents diagrammes sont donc ordonnés hiérarchiquement en faisant apparaître pour les derniers niveaux des fonctions élémentaires, appelées primitives élémentaires ou « process » primitifs. Les différents outils composant cette méthode sont :

– diagrammes de transformations de données ou diagramme de flots de données (DFD) ;
– dictionnaire de données ;
– spécifications des « process » primitifs.

Les diagrammes de flots de données sont construits à partir de quatre éléments graphiques : traitement (cercle), flot de données (flèche), unité de stockage (traits parallèles) et entité externe (rectangle) (tableau 2.1). À partir de ces éléments de base, il est possible de décrire l'aspect fonctionnel d'une application par un diagramme flots de données. Un exemple, présenté sur la figure 2.2, montre l'analyse d'une application très simple de régulation de température avec trois entités externes, deux process et une unité de stockage.

Remarque

Il est intéressant de noter que cette description graphique fonctionnelle d'une application à l'aide de la méthode SA sera presque entièrement reprise dans la méthode SA-RT, montrant bien ainsi sa dépendance chronologique.

Pour exprimer complètement le comportement de l'application, le diagramme flots de données de SA manquait d'un moyen permettant de spécifier l'aspect opérationnel,

Tableau 2.1 – Les différents éléments graphiques de la méthode SA.

Fonction	Signification	Représentation graphique	
Traitement ou process	Unité de travail qui réalise la transformation des données d'entrée en données de sortie	– Cercle ou bulle – Action décrite par : verbe + nom	Acquérir Mesures
Flot de données	Vecteur nommé reliant deux process, sur lequel circule un ensemble de données de même nature	– Flèche en trait plein – Donnée nommée	*Mesures*
Unité de stockage ou réservoir	Entité ou zone de rangement de données	– Deux traits parallèles – Entité nommée	*Mesures*
Entité externe ou terminateur	Provenance, source ou destination des données	– Rectangle – Entité nommée	Capteur

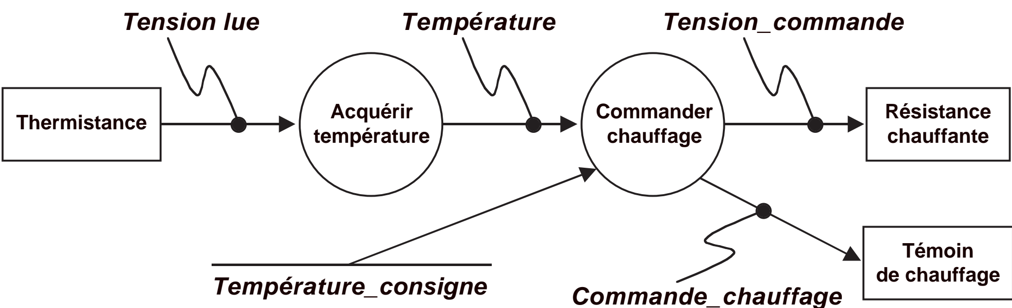


Figure 2.2 – Exemple simple du diagramme flot de données de la méthode SA correspondant à une application de régulation de température.

c'est-à-dire la description de l'enchaînement des différents process. Cette lacune fut comblée par la création de méthode SA-RT (*Structured Analysis-Real Time*). Deux groupes élaborèrent la méthode SA-RT avec des différences notables en termes de représentation : d'une part, la méthode établie par Ward et Mellor en 1985 qui associe le fonctionnel et le contrôle dans un même diagramme et, d'autre part, la méthode proposée par Hatley et Pirbhai en 1986 qui sépare le fonctionnel et le contrôle. Mais ces deux vues de la même méthode restent très similaires en termes de capacité d'expression de la spécification. Nous présentons dans cet ouvrage la méthode SA-RT établie par Ward et Mellor en 1985.

Comme le montre la figure 2.1, la méthode SA-RT a continué à évoluer au sein des entreprises en intégrant des besoins spécifiques à un domaine d'applications. Ainsi, nous trouvons une méthode SA-RT, appelée ESML et utilisée dans l'avionique, qui a été enrichie d'un point de vue flot de contrôle.

© Dunod – La photocopie non autorisée est un délit.

La méthode SA-RT intègre les trois aspects fondamentaux d'une méthode de spécification en mettant l'accent sur les deux premiers qui sont des points essentiels dans les applications de contrôle-commande :

– **aspect fonctionnel** (ou transformation de données) : représentation de la transformation que le système opère sur les données et spécification des processus qui transforment les données ;
– **aspect événementiel** (piloté par les événements) : représentation des événements qui conditionnent l'évolution d'un système et spécification de la logique de contrôle qui produit des actions et des événements en fonction d'événements en entrée et fait changer le système d'état ;
– **aspect informationnel** (données) : spécification des données sur les flots ou dans les stockages. Ce dernier aspect qui est en général assez négligé dans ce type d'application peut faire l'objet d'une description spécifique choisie au sein d'une entreprise.

2.2 Présentation de la syntaxe graphique de la méthode SA-RT

Nous allons présenter la syntaxe graphique complète de SA-RT permettant d'élaborer les différents diagrammes de la méthode. Cette syntaxe graphique très simple peut être scindée en deux parties : la syntaxe graphique afférente à l'aspect fonctionnel et la syntaxe dédiée à l'aspect contrôle ou événementiel.

2.2.1 Syntaxe graphique pour l'aspect fonctionnel

■ Syntaxe graphique du processus fonctionnel

En premier lieu, nous trouvons le **Processus fonctionnel** ou **Processus** qui représente une transformation de données. Un ou plusieurs flux de données en entrées sont traités pour donner un ou plusieurs flux de données en sortie (figure 2.3). Le Processus est représenté par un cercle avec une étiquette ou label explicite formé de :

Étiquette_Processus = verbe (+ un ou plusieurs compléments d'objets) + numéro

■ Syntaxe graphique du flot de données

Puis, nous avons le **Flot de Données** qui supporte ou transporte les valeurs d'une certaine information à différents instants. Ce concept représente le cheminement des données. Le flot de données est représenté par un arc orienté avec une étiquette ou label explicite formé de (figure 2.4) :

Étiquette_Flot_de_Données = nom (+ qualifiant)

Les valeurs de ce flot de données sont supposées disponibles pendant tout le temps où le processus producteur de ce flot est en mesure de les générer.
Le flot de données peut représenter aussi bien une donnée de type continu, codée par un entier ou un réel (`Température`) qu'une donnée discrète codée par un booléen,

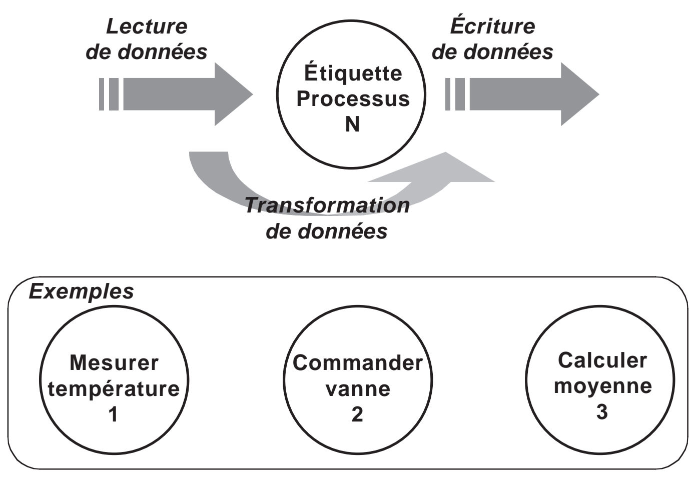


Figure 2.3 – Processus fonctionnel de la méthode SA-RT.

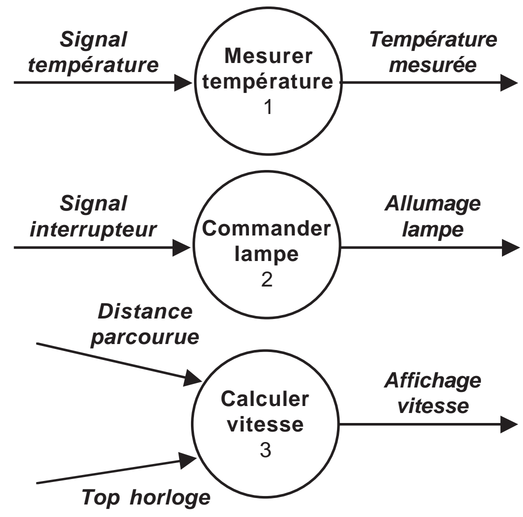


Figure 2.4 – Exemples simples de flots de données de la méthode SA-RT.

© Dunod – La photocopie non autorisée est un délit.

exemple « *Position_interrupteur* ». Un flot de données peut décrire aussi bien une donnée élémentaire ou unique, exemple « *Température* » qu'une donnée structurée intégrant plusieurs données élémentaires, exemple la donnée « *Pressions* » qui est composée de « *Pression_huile* » et « *Pression_air* ». Il est alors possible de faire apparaître l'étiquette du flot de données sous la forme suivante :

Étiquette_Donnée_Structurée = Étiquette_Donnée_1, Étiquette_Donnée_2

Une spécification détaillée de cette donnée, véhiculée par le flot de données, est faite dans le **Dictionnaire de données** (voir ci-après).

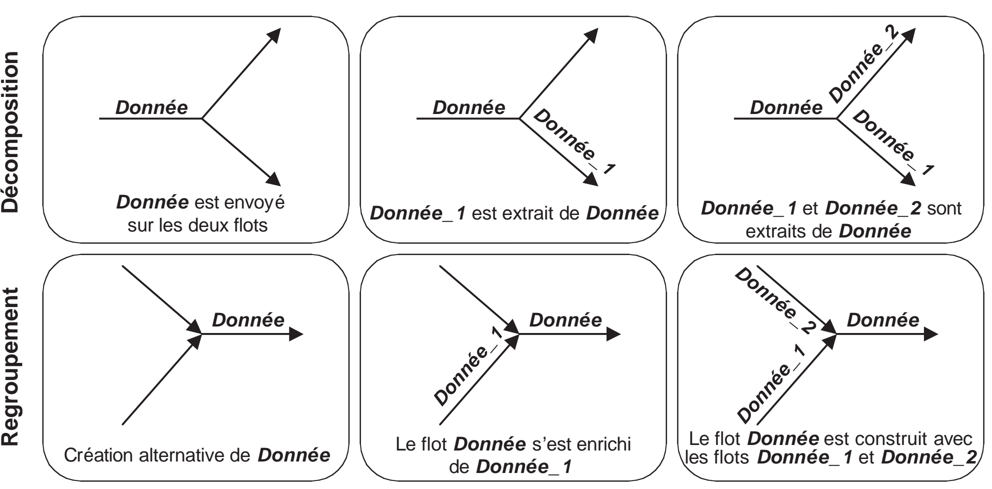


Figure 2.5 – Décomposition et regroupement des flots de données de la méthode SA-RT.

Ces flots de données peuvent se décomposer ou au contraire se regrouper lors des liaisons entre les processus fonctionnels dans le diagramme flot de données (figure 2.5).

■ Syntaxe graphique du stockage de données

Le troisième élément graphique est le **Stockage de Données** qui modélise le besoin de mémorisation d'une donnée de telle façon que sa valeur puisse être relue plusieurs fois. Comme le flot de données auquel il est étroitement associé, il est nommé par une étiquette ou label explicite formé de :

Étiquette_Stockage_de_Données = nom (+ qualifiant)

Le stockage de données est représenté par deux traits horizontaux encadrant l'étiquette définie ci-avant (figure 2.6). Les arcs « flots de données » arrivant ou partant de l'unité de stockage ne sont pas étiquetés s'ils transportent les données mémorisées complètes. Si une partie de la donnée est écrite ou lue, l'arc transportant de façon partielle la donnée doit être étiqueté avec le nom de cette donnée.

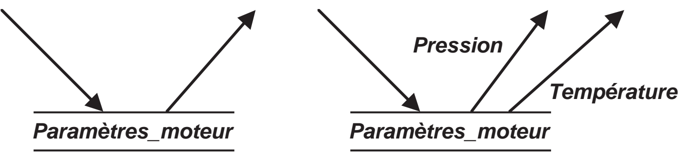


Figure 2.6 – Unité de stockage de la méthode SA-RT.

Un exemple de diagramme flot de données intégrant ces trois éléments graphiques de la méthode SA-RT (processus fonctionnel, flot de données et unité de stockage) est présenté sur la figure 2.7.

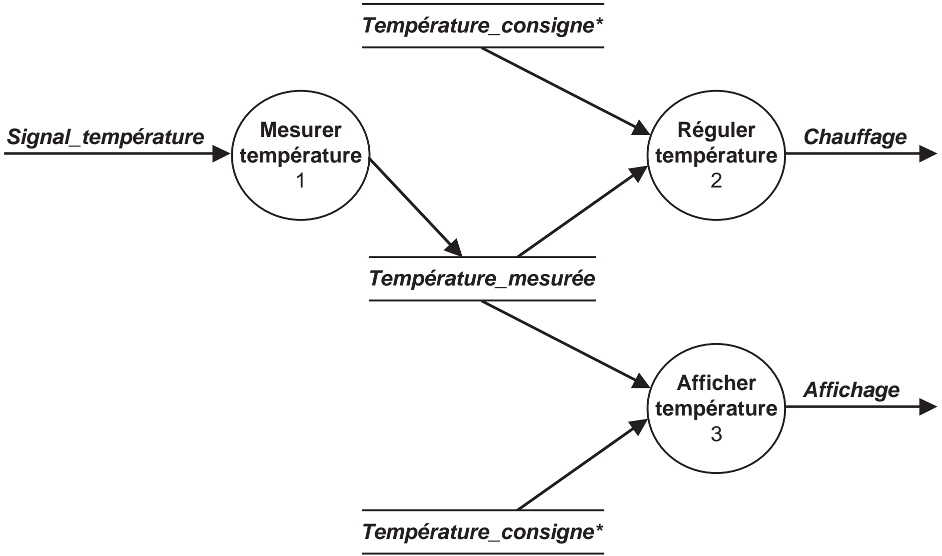


Figure 2.7 – Exemple d'un diagramme flot de données de la méthode SA-RT.

> **Remarque**
> Pour des besoins de clarté graphique, un stockage de données peut être visualisé plusieurs fois sur un diagramme flot de données en notant cette duplication par une « * ».

Il est important de noter que l'utilisation de l'élément « stockage de données » peut correspondre à deux cas :

– mémorisation ou représentation des constantes du système ;
– mémorisation de valeurs de données partagées entre deux ou plusieurs processus désynchronisés (consommation et production des données à des instants ou des rythmes différents).

■ Syntaxe graphique de la Terminaison

Enfin, le dernier élément graphique, utilisé dans cet aspect fonctionnel, est la **Terminaison**, ou encore appelée **« bord de modèle »**, qui représente une entité extérieure échangeant des données avec le système modélisé. Une terminaison peut donc être une entité logicielle (programme, base de données…) ou matérielle (capteurs, actionneurs, console opérateur…). Représentée par un rectangle, elle est nommée par une étiquette ou label explicite formé de (figure 2.8) :

Étiquette_Terminaison = nom (+ qualifiant)

© Dunod – La photocopie non autorisée est un délit.

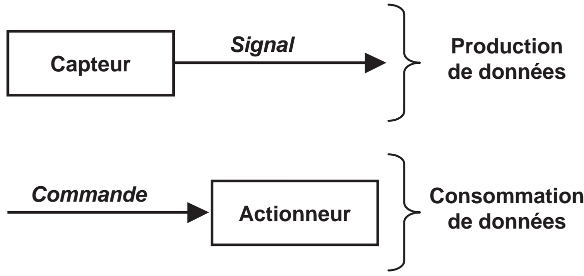


Figure 2.8 – Terminaison ou « bord de modèle » de la méthode SA-RT.

2.2.2 Syntaxe graphique pour l'aspect contrôle

Dans les diagrammes flot de données, le déclenchement de l'exécution des processus de transformation de données peut être lié au rythme d'apparition des données entrantes (diagramme piloté par les données : *data-driven*). Mais, dans le cas de la spécification des applications temps réel, il est préférable d'avoir un contrôle de ces transformations de données piloté par des conditions externes comme l'occurrence d'événements (*event-driven*) (figure 2.9). Cette vue dynamique du modèle impose la mise en place de la partie contrôle : processus de contrôle et flot de contrôle

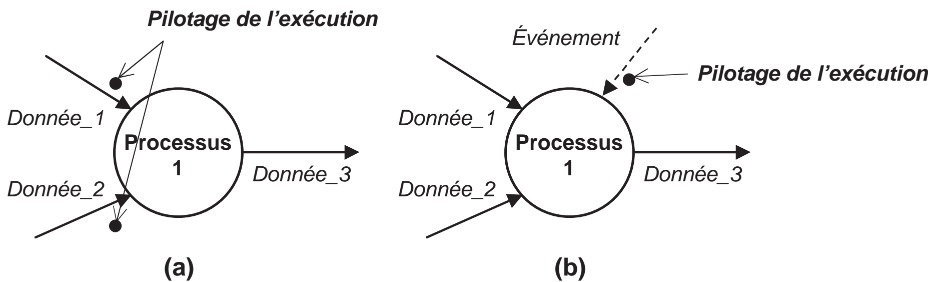


Figure 2.9 – Pilotage de l'exécution d'un processus fonctionnel : (a) piloté par les données et (b) piloté par les événements.

■ Syntaxe graphique du processus de contrôle

Le **Processus de contrôle** représente la logique du pilotage des processus fonctionnels. Il génère l'ensemble des événements qui vont activer ou désactiver les processus fonctionnels. En retour, les processus fonctionnels fournissent au processus de contrôle tous les événements nécessaires aux prises de décision. Le processus de contrôle ne peut en aucun cas gérer des données.

Le Processus de contrôle est représenté par un cercle en pointillé avec une étiquette ou label explicite formé de (figure 2.10) :

Étiquette_Processus_Contrôle = verbe (+ un ou plusieurs compléments) + numéro

Figure 2.10 – Processus de contrôle de la méthode SA-RT.

■ Syntaxe graphique du Flot de contrôle

Le **Flot de Contrôle** transporte les événements ou informations qui conditionnent directement ou indirectement l'exécution des processus de transformations de données. Le flot de contrôle est représenté par un arc orienté pointillé avec une étiquette ou label explicite formé de (figure 2.11) :

Étiquette_Flot_de_Contrôle = nom (+ qualifiant)

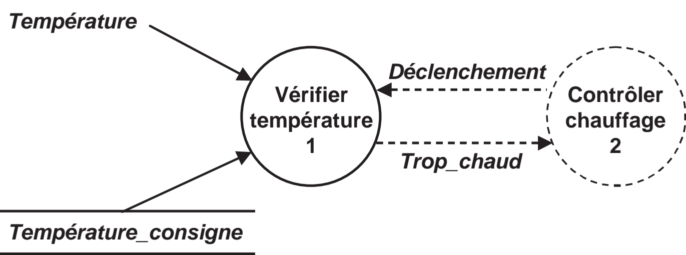


Figure 2.11 – Exemple simple d'une partie contrôle liée à une partie fonctionnelle de la méthode SA-RT.

Les événements, fournis par le **processus de contrôle**, sont généralement liés à l'activation ou à la désactivation des processus fonctionnels. Aussi, ces événements spécifiques ont été formalisés et prédéfinis :

– ***E*** pour *Enable* (activation) ;
– ***D*** pour *Disable* (désactivation) ;
– ***T*** pour *Trigger* (déclenchement).

Les deux premiers événements sont utilisés ensemble « *E/D* » pour piloter un processus fonctionnel de type « boucle sans fin » ou périodique, c'est-à-dire que le processus de contrôle doit lancer l'exécution de ce processus avec l'événement « *E* » et ensuite peut l'arrêter avec l'événement « *D* ». L'événement « *T* » est utilisé pour activer un processus fonctionnel de type « début-fin » ou sporadique, c'est-à-dire que le processus de contrôle doit lancer l'exécution de ce processus avec l'événement « *T* » et ensuite le processus s'arrête à la fin de son exécution sans intervention du contrôle.

© Dunod – La photocopie non autorisée est un délit.

2.3 Les diagrammes flot de données

2.3.1 Présentation d'un exemple simple d'application de contrôle-commande

Afin d'illustrer la méthodologie au fur et à mesure de la présentation, nous allons présenter un exemple simple qui va être décliné en détail au cours de cet ouvrage. Considérons un **système de freinage automobile** qui est constitué d'une part d'un ensemble classique composé d'une pédale de frein (demande de freinage) et d'un frein (actionneur de freinage) et d'autre part d'un système ABS (*Anti-blocking Brake System*). Un capteur de glissement de roues est associé à ce système ABS. Pour simplifier, le fonctionnement de l'ABS est basé sur un arrêt du freinage dès qu'un glissement est détecté sur les roues, et cela même si la demande du conducteur est toujours effective. Le conducteur a la possibilité d'activer ou non ce système ABS à l'aide d'un bouton spécifique (bouton à deux positions stables : interrupteur). Un voyant permet de lui indiquer l'activation du système ABS. En revanche, il n'est pas possible de désactiver le système ABS en cours de freinage, c'est-à-dire pendant l'appui sur la pédale de frein. La spécification fonctionnelle de cette application à l'aide de la méthode SA-RT va s'effectuer en plusieurs étapes :

- diagramme de contexte ;
- diagramme préliminaire ;
- diagrammes de décomposition.

2.3.2 Diagramme de contexte d'une application

Le diagramme de contexte est une première étape extrêmement importante puisqu'elle va définir le contexte et l'environnement extérieur du système piloté. Nous pouvons la considérer comme le contrat de réalisation entre le concepteur et son client. Les bords de modèle ou terminaisons vont apparaître uniquement dans ce diagramme. Les descriptions précises de ces terminaisons, ainsi que des données ou éventuellement des événements entrants ou sortants de ceux-ci, sont à la charge du donneur d'ordre. Un exemple générique d'un diagramme de contexte est donné sur la figure 2.12.

Nous trouvons un et un seul processus fonctionnel, numéroté « 0 », qui traduit l'application à réaliser effectivement par le concepteur. Autour de ce processus fonctionnel, un ensemble de bords de modèles fournit ou consomme les données ou événements de cette application. Ces bords de modèles peuvent donc être les éléments physiques suivants :

- capteurs (thermocouples, dynamomètres, etc.) ;
- actionneurs (résistances chauffantes, vannes, etc.) ;
- opérateur(s) lié(s) à des capteurs (interrupteurs, potentiomètres, etc.) ;
- systèmes d'affichage (lampes, diodes, écran d'ordinateur, etc.) ;
- système de stockage ou de sauvegarde externe (disque, bande magnétique, etc.) ;
- système d'impression (imprimante, dérouleur papier, etc.) ;
- …

Opérateur 1
Consigne_1
Capteur 1
Donnée_E_1
Donnée_S_1
Actionneur 1
Capteur 2
Donnée_E_2
Piloter application 0
Donnée_S_2
Actionneur 2
Donnée_S_3
Capteur 3
Donnée_E_3
Affichage_1
Actionneur 3
Écran

Figure 2.12 – Diagramme de contexte générique de la méthode de spécification SA-RT.

L'ensemble des données ou événements échangés avec l'extérieur du « processus fonctionnel », qui représente l'application, constitue les spécifications d'entrées et de sorties de l'application. La description de ces entrées/sorties sera faite dans le dictionnaire de données (§ 2.5).

Dans l'exemple simple d'un système de freinage automobile (§ 2.3.1), le diagramme de contexte est constitué du processus fonctionnel « Contrôler système freinage 0 » et de cinq bords de modèles (figure 2.13) :

– terminaison « Pédale de frein » fournissant la donnée « *Demande_freinage* » ;
– terminaison « Bouton d'activation de l'ABS » fournissant la donnée « *Activation_ABS* » ;
– terminaison « Capteur de glissement » fournissant la donnée « *Glissement_roue* » ;
– terminaison « Système de freinage » consommant la donnée « *Commande_freinage* » ;
– terminaison « Voyant ABS » consommant la donnée « *Affichage_ABS* ».

Ce diagramme de contexte définit parfaitement l'interface entre le concepteur et le client, c'est-à-dire les données à fournir ou à générer. La suite du travail d'analyse

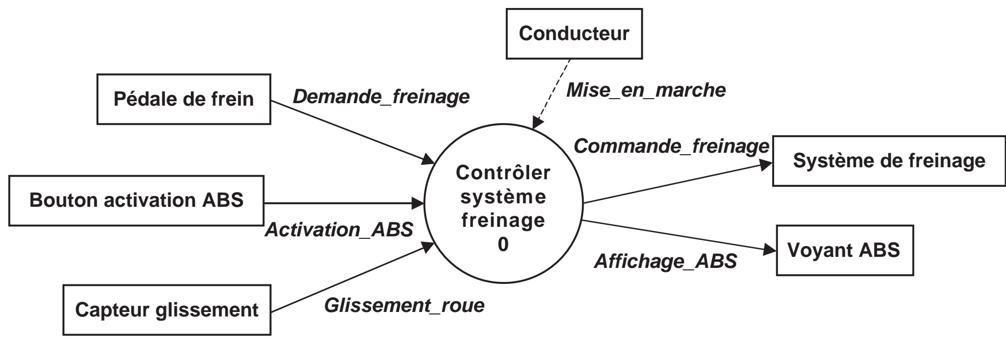


Figure 2.13 – Diagramme de contexte de l'application « système de freinage automobile ».

© Dunod – La photocopie non autorisée est un délit.

va donc se situer dans l'expression du processus fonctionnel à réaliser : « Contrôler système freinage 0 ».

2.3.3 Diagramme préliminaire et diagrammes de décomposition

Le premier niveau d'analyse est représenté par le **diagramme préliminaire**. Ce diagramme préliminaire est la première décomposition du processus à réaliser présenté dans le diagramme de contexte. À ce niveau, le diagramme représente la liste « graphique » des processus fonctionnels nécessaires à l'application sans se soucier de l'enchaînement (séquence d'exécution).

Le nombre de processus fonctionnels, composant ce diagramme préliminaire, doit être limité pour avoir une meilleure lisibilité : **5** à **9** maximum.

Dans ce cadre, nous pouvons aussi trouver un motif générique permettant de décrire une fonction simple ou complexe de contrôle-commande en la divisant selon les trois éléments de base, soit :

– un processus d'acquisition ;

– un processus de traitement (loi de régulation) ;

– un processus de commande.

Selon l'application, tout ou partie de ce diagramme flot de données générique peut être présent pour chaque chaîne de contrôle-commande impliquée dans le niveau d'analyse en cours d'élaboration. Ce diagramme flot de données générique peut être utilisé au niveau du diagramme préliminaire ainsi que dans les niveaux d'analyse suivant des diagrammes de décomposition.

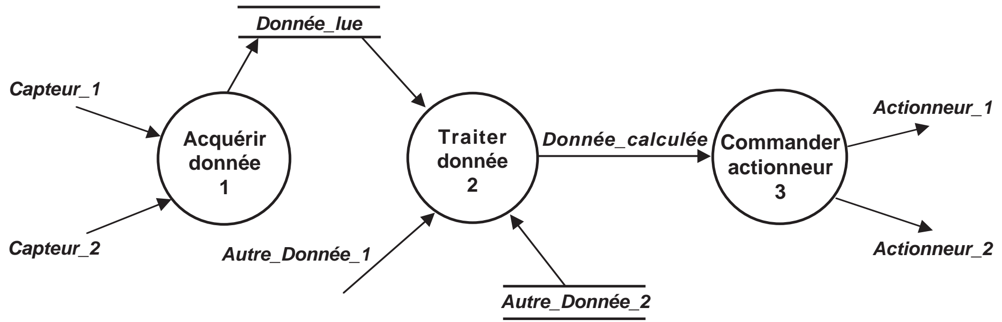


Figure 2.14 – Décomposition fonctionnelle générique.

Le passage des données entre les processus fonctionnels peut être réalisé selon les besoins avec les deux méthodes de base : flots de données direct (exemple entre les processus 2 et 3) ou unité de stockage (cas des processus 1 et 2).

Ainsi, dans l'exemple simple d'un système de freinage automobile (§ 2.3.1), le diagramme préliminaire est constitué de cinq processus fonctionnels (figure 2.15). Au niveau de cet exemple simple, nous n'avons pas une décomposition fonctionnelle aussi complexe que l'exemple générique présenté précédemment : seules les parties

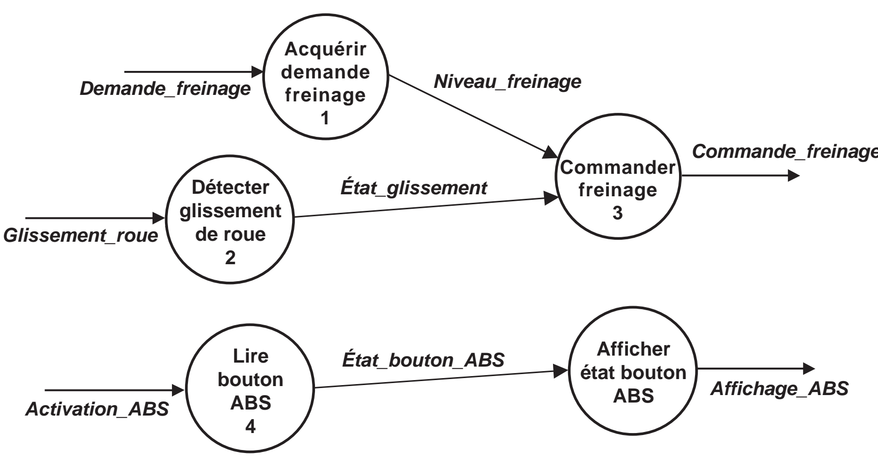


Figure 2.15 – Diagramme flot de données préliminaire de l'application « système de freinage automobile ».

acquisition et commande sont présentes dans le cas du contrôle-commande du freinage et du bouton ABS.

Nous pouvons souligner immédiatement la cohérence obligatoire entre le diagramme de contexte (figure 2.13) et ce diagramme préliminaire au niveau des flots de données entrants et sortants.

Le passage des données entre les processus fonctionnels est effectué de façon directe. Il est important de noter que la donnée « *Niveau_freinage* » est de type entier ou réel alors que les données « *Etat_glissement* » et « *Etat_bouton_ABS* » sont de type booléen. Cela va entraîner dans la suite de cette analyse une modification de ce diagramme préliminaire.

Cette décomposition fonctionnelle en diagrammes flots de données peut se poursuivre en raffinant de plus en plus la description des processus fonctionnels (figure 2.16). Chaque **diagramme de décomposition**, associé à un processus fonctionnel – numéroté « N » – du diagramme hiérarchiquement supérieur, est référencé au niveau des processus fonctionnels par des numéros « N.x ».

L'exemple simple « système de freinage automobile » choisi pour illustrer la méthodologie n'est pas assez complexe pour justifier la décomposition d'un des processus fonctionnels. Donc tous les processus fonctionnels du diagramme préliminaire de la figure 2.15 sont des processus primitifs.

Lorsqu'il n'y a plus d'intérêt à décomposer un processus, celui-ci est appelé **processus primitif** et doit être décrit par une spécification sous forme textuelle, tabulaire ou graphique (voir ci-après).

Nous pouvons énoncer les deux premières règles de cohérence de cette décomposition hiérarchique :

– l'ensemble des flots entrants et sortants du processus décomposé doit se retrouver dans le diagramme de décomposition de ce processus avec les mêmes typages (données ou événements) ;

© Dunod – La photocopie non autorisée est un délit.

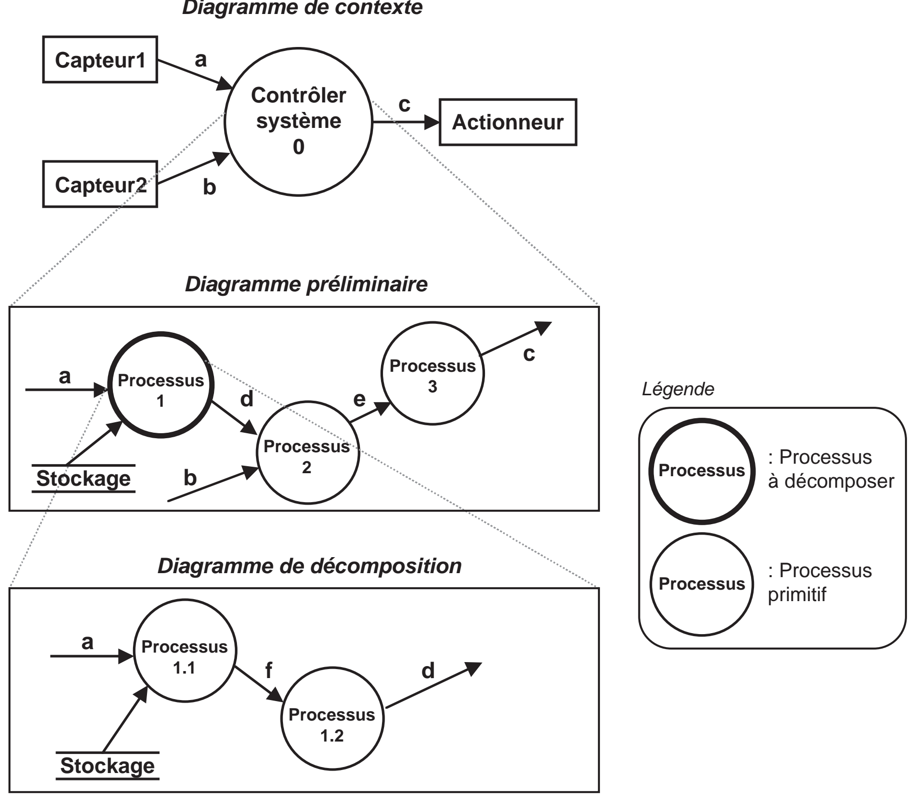


Figure 2.16 – Décomposition hiérarchique en diagramme flots de données de la méthode d'analyse SA_RT.

– la numérotation des différents processus fonctionnels doit intégrer le numéro du processus fonctionnel analysé « N » sous la forme « N.x » ;
– les stockages doivent apparaître dans tous les diagrammes où les processus les utilisent.

2.3.4 Conclusion

La décomposition hiérarchique fonctionnelle, explicitée dans ce paragraphe, est la base de la méthode d'analyse SA-RT. Mais l'aspect temporel ou plus exactement le contrôle de l'enchaînement dans l'exécution de ces différents processus fonctionnels n'est pas décrit au travers de ces diagrammes flots de données. Ceci va donc faire l'objet d'un complément au niveau de ces diagrammes par l'aspect « contrôle ».

2.4 L'aspect contrôle de la méthode SA-RT

2.4.1 Mise en place du processus de contrôle

Le processus de contrôle, tel que nous l'avons déjà explicité dans le paragraphe 2.2.2, va permettre de spécifier l'enchaînement des différents processus fonctionnels de l'application. Ainsi, ce processus de contrôle va être positionné dans le diagramme préliminaire et dans les diagrammes de décomposition s'ils existent.

■ Flots de contrôle et diagramme de contexte

Nous pouvons souligner qu'il ne peut pas exister de processus de contrôle au niveau du diagramme de contexte puisqu'un seul diagramme fonctionnel est représenté. En revanche, il est tout à fait possible d'avoir des flots de contrôle au niveau de ce diagramme de contexte entre les terminaisons et le processus fonctionnel. Ces flots de contrôle, correspondant à des signaux tout ou rien, doivent être réservés à des signaux particuliers ne nécessitant aucun processus fonctionnel particulier (acquisition, mise en forme…). Ainsi, nous pouvons citer les événements externes tels que « frappe clavier », « click souris », « interrupteur de mise sous tension », etc. Pour préciser de façon plus complète ces notions de signaux et d'événements, nous pouvons considérer qu'un signal nécessite une acquisition (scrutation périodique par exemple) et une mise en forme minimum. Le résultat de ce « prétraitement » peut alors effectivement devenir un événement. La figure 2.17a illustre cette différence signal/événement en considérant l'exemple d'un interrupteur. Le signal électrique fourni par l'interrupteur tout ou rien peut être considéré comme un événement

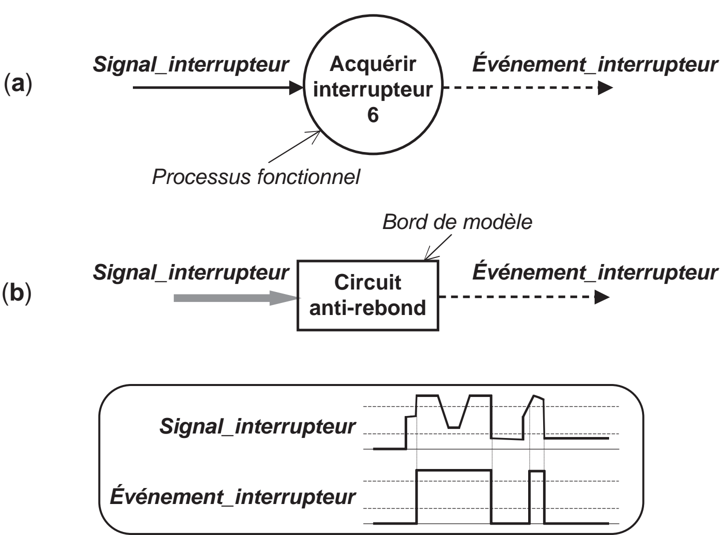


Figure 2.17 – Méthodes de prise en compte d'un signal électrique afin de réaliser une transformation données-événements : (a) processus fonctionnel dans le diagramme préliminaire, (b) circuit électronique visualisé au niveau du diagramme de contexte.

© Dunod – La photocopie non autorisée est un délit.

s'il est directement compréhensible par le système informatique, par exemple connexion à une ligne d'interruption. Dans le cas contraire, il nécessite une lecture régulière et une mise en forme pour être intégré à la logique du programme. Il est important de noter que ces processus de mise en forme des signaux afin de transformer une donnée en un événement peuvent parfois être remplacés de façon beaucoup plus facile et efficace par un circuit électronique (figure 2.17b).
Dans le cas de l'exemple simple du système de freinage automobile (§ 2.3.1), le diagramme de contexte peut être enrichi d'un flot de contrôle émanant d'un nouveau bord de modèle « Conducteur » qui fournit un flot de contrôle « *Mise_en_marche* » au processus fonctionnel (figure 2.18).

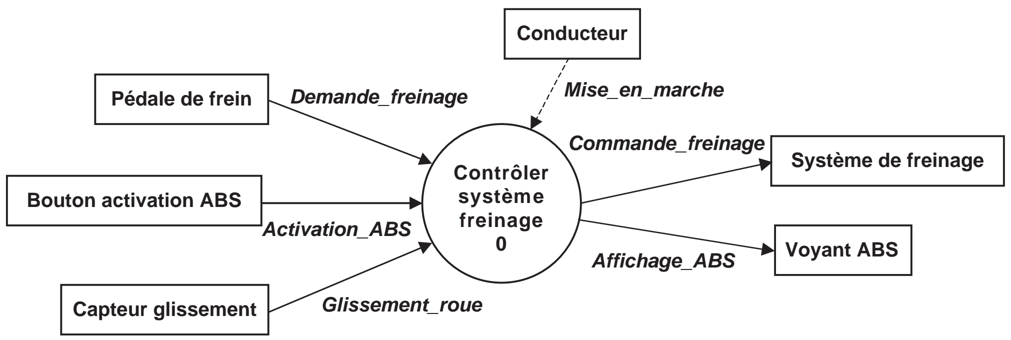


Figure 2.18 – Diagramme de contexte complet de l'application « système de freinage automobile ».

Rappelons que l'ensemble des flots entrants et sortants de l'unique processus fonctionnel du diagramme de contexte doit se retrouver dans le diagramme préliminaire y compris les flots de contrôle s'ils existent.

■ Processus de contrôle et diagramme préliminaire

Nous allons donc implanter un processus de contrôle dans le diagramme préliminaire afin de coordonner l'exécution des différents processus fonctionnels. Les concepts et les règles de mise en place de ce processus de contrôle sont identiques dans le cas des diagrammes de décomposition. Ce processus de contrôle va donc interagir avec un processus fonctionnel pour lancer ou activer son exécution et, en retour, le processus fonctionnel fournira si nécessaire un événement indiquant le résultat de son traitement afin de donner des informations utiles aux changements d'états du contrôle. Nous pouvons ainsi illustrer cette interaction entre le processus de contrôle et les processus fonctionnels selon le schéma générique utilisant la décomposition fonctionnelle complète présentée sur la figure 2.14. Rappelons que l'événement couplé « *E/D* » est utilisé pour piloter un processus fonctionnel de type « boucle sans fin » et l'événement « *T* » est utilisé pour activer un processus fonctionnel de type « début-fin ». Dans cet exemple de la figure 2.19, deux processus fonctionnels, supposés de type « début-fin », sont activés l'un après l'autre par le processus de contrôle par un événement « T » et envoient un événement de fin d'exécution vers

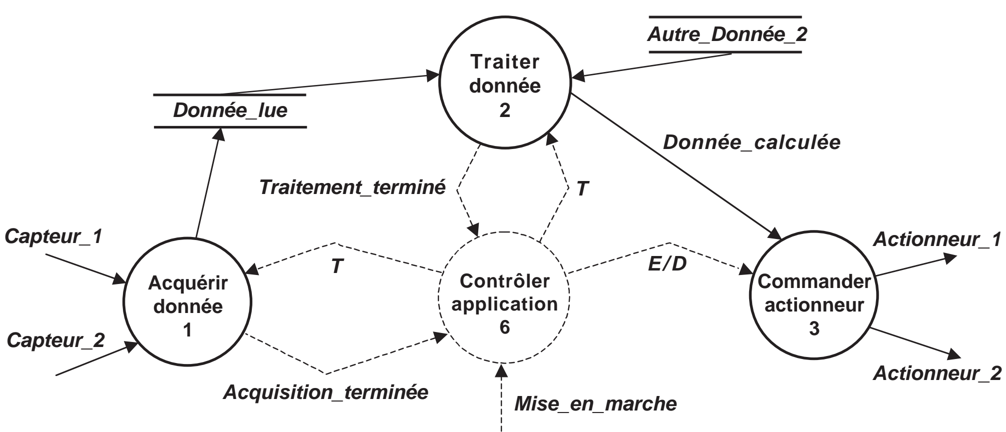


Figure 2.19 – Schéma générique de l'interaction entre le processus de contrôle et les processus fonctionnels.

ce même processus de contrôle. Le troisième processus fonctionnel, activé par un événement « E/D », est lié à la commande des actionneurs, qui doit être faite en continu, soit type « boucle sans fin ».

Concernant cette mise en place d'un processus de contrôle dans un diagramme préliminaire, nous pouvons faire plusieurs remarques :

– Un diagramme préliminaire ne doit contenir qu'un seul processus de contrôle. En effet, il est difficilement concevable d'avoir plusieurs organes de contrôle-commande pour une seule application, pour des raisons de cohérence.
– Un diagramme préliminaire, ou, *a fortiori*, un diagramme de décomposition, peut ne pas avoir de processus de contrôle. Dans ce cas, tous les processus fonctionnels sont supposés s'exécuter en même temps avec pour seule règle celle des flots de données.
– Un processus fonctionnel peut ne pas être connecté au processus de contrôle. Dans ce cas, il est supposé être activé au démarrage de l'application et ne jamais s'arrêter. Aussi, pour augmenter la lisibilité, il est préférable de le connecter au processus de contrôle avec un événement de type « E/D » et de l'activer définitivement au début de l'application en utilisant l'événement « E ».

Dans le cas de l'exemple du système de freinage automobile, le diagramme préliminaire, représenté sur la figure 2.15, va être modifié pour intégrer un processus de contrôle. En particulier, les deux flots de données « *Etat_glissement* » et « *Etat_bouton_ABS* » de type booléen deviennent des événements envoyés respectivement par les processus fonctionnels « Détecter glissement roue » et « Lire boutons ABS ». Nous obtenons alors le diagramme préliminaire complet de la figure 2.20.

Cet exemple de diagramme préliminaire, présenté sur la figure 2.20, doit faire l'objet de plusieurs remarques qui sont généralisables à la réalisation d'un diagramme préliminaire quelconque. Ainsi, nous pouvons noter :

– Tous les différents processus fonctionnels sont liés au processus de contrôle par des flots de contrôle de type « E/D » ou « T ». Le choix de l'un ou l'autre de ces

© Dunod – La photocopie non autorisée est un délit.

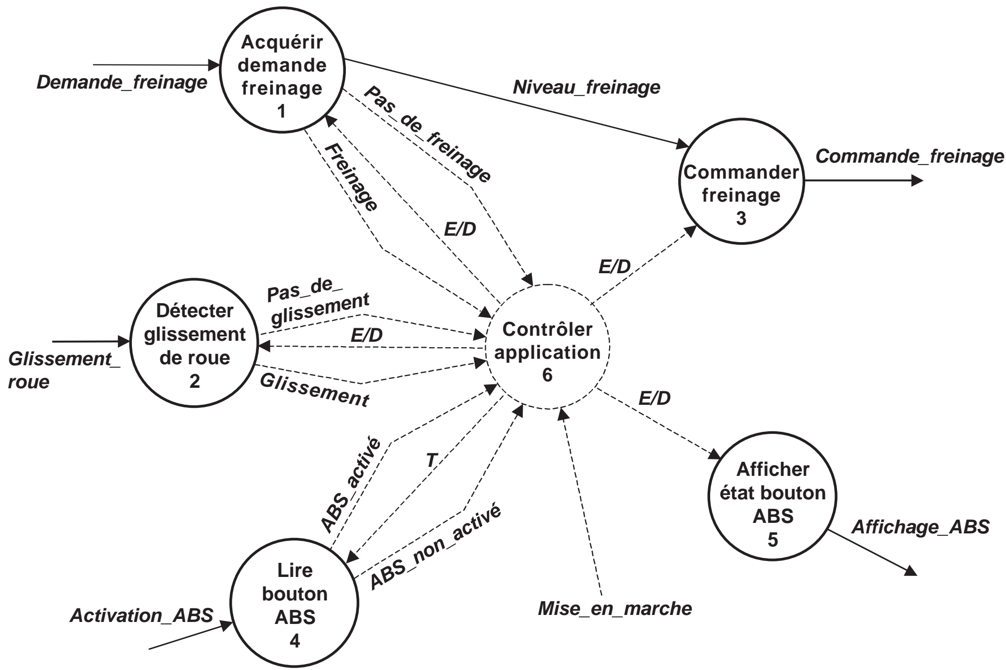


Figure 2.20 – Diagramme préliminaire complet de l'application « système de freinage automobile ».

événements est fait, comme nous l'avons déjà vu, en fonction de la structure de fonctionnement interne du processus fonctionnel : processus de type « boucle sans fin » ou de type « début-fin ».

– Certains processus fonctionnels possèdent des retours d'exécution vers le processus de contrôle. Dans ce cas, il est souhaitable d'expliciter les événements indiquant le résultat du traitement, par exemple « *ABS_activé* ». Ces événements pourraient être caractérisés par des variables de type booléen ; ainsi l'absence d'occurrence de l'événement « *ABS_activé* » pourrait être interprétée comme NON (« *ABS_activé* ») – NON étant l'opération booléenne. Mais, dans beaucoup de cas, il est préférable pour des raisons de clarté opérationnelle, de faire apparaître les deux événements possibles : « *ABS_activé* » et « *ABS_non_activé* ».

– Il ne faut pas oublier de connecter les flots de contrôle du diagramme précédent réalisé dans l'analyse descendante, par exemple l'événement « *Mise_en_marche* » du diagramme de contexte.

La mise en place du processus de contrôle au niveau d'un diagramme préliminaire ou d'un diagramme de décomposition avait pour but d'exprimer l'exécution ou l'enchaînement des processus fonctionnels. L'objectif n'est pas complètement atteint puisque le diagramme flot de données ainsi complété ne reflète pas cette exécution. Il est nécessaire d'ajouter cette information supplémentaire décrivant le fonctionnement du processus de contrôle, cela se traduit généralement par un diagramme état/transition que nous allons décrire dans le paragraphe suivant.

2.4.2 Diagramme état/transition

■ Représentation d'un diagramme état/transition

Dans le cas où un diagramme flots de données possède un processus de contrôle, la compréhension de ce diagramme de flot de données, tracé à un certain niveau d'analyse, nécessite une description ou spécification du processus de contrôle. Cette spécification, représentant l'aspect comportemental ou temps réel de l'application, peut être faite de diverses manières : diagramme état-transition, table état-transition, matrice état-transition ou éventuellement un grafcet.
La représentation la plus courante est le **diagramme état-transition**, ou automate synchronisé, qui est composé de quatre éléments (figure 2.21) :

– **état courant** correspondant à un fonctionnement précis du système, en particulier à un état des processus fonctionnels (exécution ou non) ;
– **événement** : occurrence d'un événement émanant d'un processus fonctionnel vers le processus de contrôle qui va provoquer le franchissement de la transition et donc faire changer l'état du système ;
– **action** : occurrence d'un événement émanant du processus de contrôle vers un ou plusieurs processus fonctionnels pour les activer (« E » ou « T ») ou les désactiver (« D »). Ces actions caractérisent l'effet du franchissement de la transition ;
– **état suivant** correspondant au fonctionnement après les actions faites par le processus de contrôle en direction des processus fonctionnels.

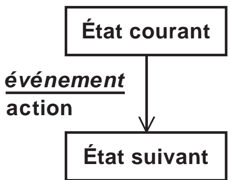


Figure 2.21 – Représentation de base d'une cellule élémentaire d'un diagramme état/transition.

Pour illustrer cette description du processus de contrôle avec un exemple très générique, reprenons le diagramme préliminaire de la figure 2.19 qui représente la coordination d'un système acquisition-traitement-commande par un processus de contrôle. La figure 2.22 montre la séquentialité évidente de l'exécution d'une telle application. À propos de ce diagramme état/transition, nous pouvons faire les remarques générales suivantes :

– Les noms des différents états atteints par le système explicitent l'état de fonctionnement du système. Ces états ne doivent pas en principe être des états fugitifs.
– Les événements intervenant au niveau de la transition entre ces états doivent tous appartenir au diagramme préliminaire et réciproquement tous les événements du diagramme état/transition doivent être représentés dans le diagramme préliminaire.

© Dunod – La photocopie non autorisée est un délit.

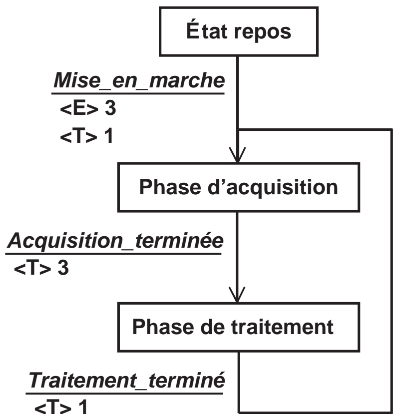


Figure 2.22 – Représentation du diagramme état/transition du processus de contrôle du diagramme préliminaire de la figure 2.19.

– Les actions du processus de contrôle sur les processus fonctionnels sont notées par « **< E** *ou* **D** *ou* **T >** + ***numéro*** *ou* ***nom*** *du processus fonctionnel* ». Il peut y avoir plusieurs actions de ce type associées à une seule transition.

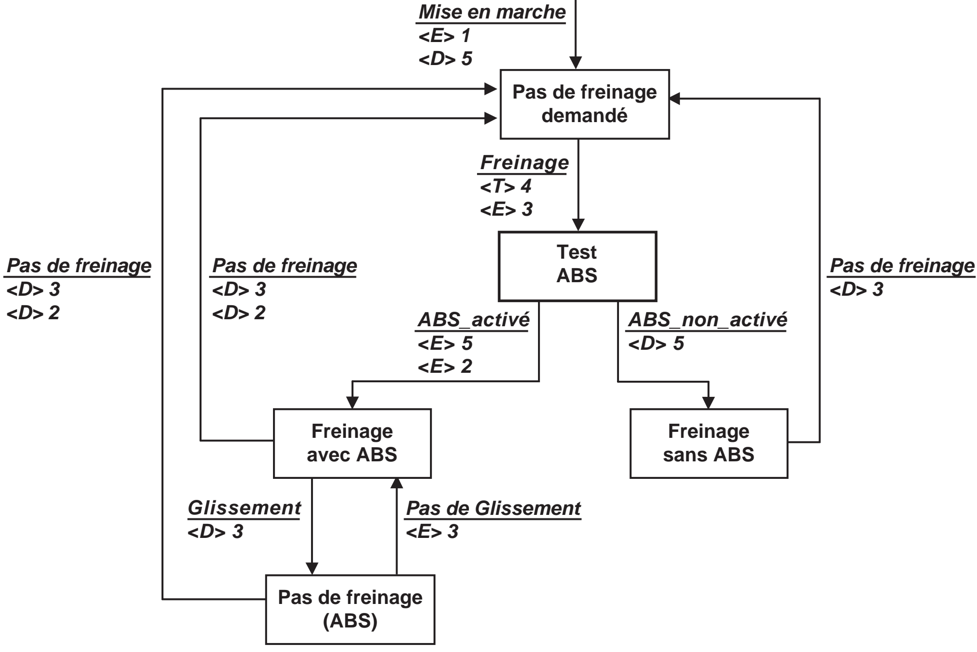


Figure 2.23 – Première représentation du diagramme état/transition du processus de contrôle « Contrôler application » du diagramme préliminaire de l'application « système de freinage automobile » : comportement non conforme.

Si nous considérons l'exemple du système de freinage automobile, une solution possible du diagramme état/transition du processus de contrôle « Contrôler application » du diagramme préliminaire est présentée sur la figure 2.23. Ce diagramme état/transition explique le comportement de l'application.

Une analyse rapide de cet exemple simple montre une incohérence dans ce comportement ; en effet, l'état initial de l'application « Pas de freinage demandé » n'est sensible qu'à l'occurrence de l'événement « *Freinage* » provenant du processus fonctionnel 1 « Acquérir demande freinage ». Le bouton d'activation ou de désactivation du système ABS avec le voyant associé n'est pris en compte que lors d'un freinage (événement « *Freinage* »). Ce comportement ne correspond pas à un fonctionnement correct du système où le conducteur doit accéder à cette fonction et voir son résultat en dehors de la phase de freinage. En effet, avec un tel diagramme état/transition, le conducteur ne peut voir la modification du voyant ABS qu'en actionnant la pédale de frein.

Un deuxième diagramme état/transition propose un autre comportement de l'application plus conforme aux spécifications d'un tel système. Ce diagramme état/transition, présenté sur la figure 2.24 (page suivante), montre clairement les deux fonctionnements du système en cas de freinage avec le système ABS activé ou sans le système ABS. Dans ce cas du diagramme état/transition, le conducteur voit immédiatement la modification du voyant ABS lors de l'appui sur le bouton ABS.

Une modification au niveau des événements d'activation par le processus de contrôle a été effectuée pour le processus 4 « Lire bouton ABS ». Dans le diagramme préliminaire de la figure 2.20, l'événement est de type « T ». Or, dans le nouveau diagramme état/transition de la figure 2.24 par rapport au premier diagramme de la figure 2.23, cet événement est de type « E/D ». Il est très important de noter que, pour conserver la cohérence entre les diagrammes d'analyse, c'est-à-dire le diagramme préliminaire et le diagramme état/transition du processus de contrôle, il est indispensable de modifier le diagramme préliminaire de la figure 2.20 comme le montre la figure 2.25, page suivante.

■ Règles générales d'élaboration d'un diagramme état/transition

L'exemple simple traité précédemment montre de façon évidente qu'il est indispensable de conduire en parallèle d'une part la réalisation du diagramme préliminaire au niveau de la mise en place du processus de contrôle et d'autre part la construction du diagramme état/transition de ce processus de contrôle. En effet, que ce soit au niveau des événements d'activation ou de désactivation issus du processus de contrôle, ou pour les événements provenant des processus fonctionnels, il est nécessaire de les mettre en place dans le diagramme préliminaire au fur et à mesure de la construction du diagramme état/transition. Aussi, à la fin de ce travail, nous devons vérifier impérativement que tous les événements du diagramme préliminaire ont été utilisés et uniquement ceux-ci.

D'autre part, le passage d'un état à un autre dans un diagramme état/transition peut être réalisé par l'occurrence d'un événement ou de la combinaison logique de plusieurs événements. Il est donc possible d'utiliser les opérateurs de la logique combinatoire pour grouper les événements : ET, OU, NON. Dans ce cas, il faut être vigilant pour ne pas diminuer fortement la lisibilité du diagramme qui représente

© Dunod – La photocopie non autorisée est un délit.

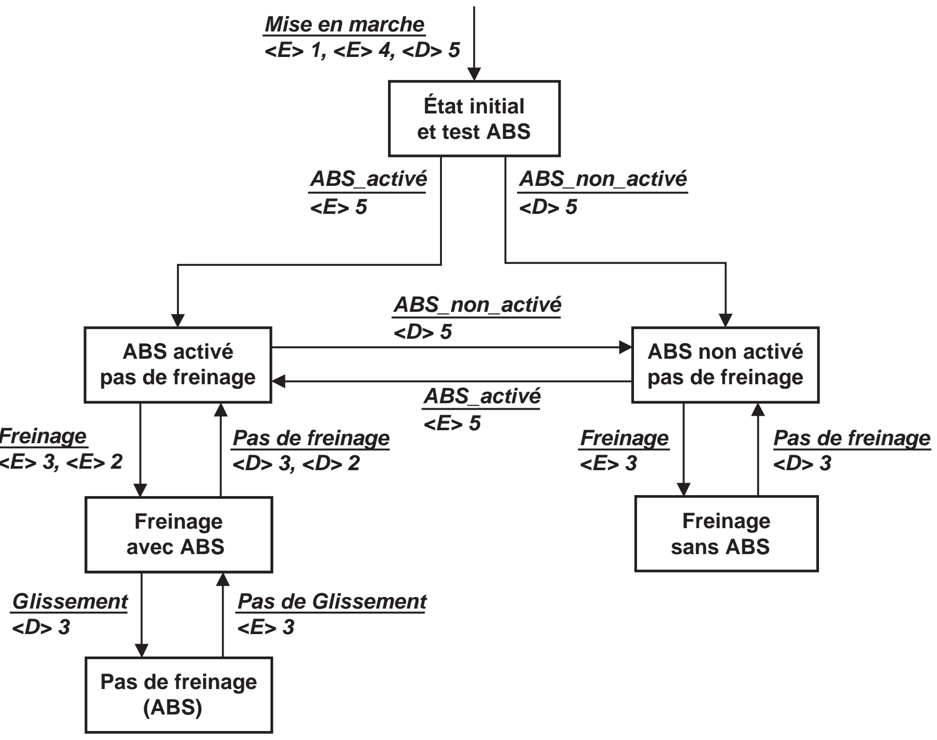


Figure 2.24 – Deuxième représentation plus correcte du diagramme état/transition du processus de contrôle « Contrôler application » du diagramme préliminaire de l'application « système de freinage automobile ».

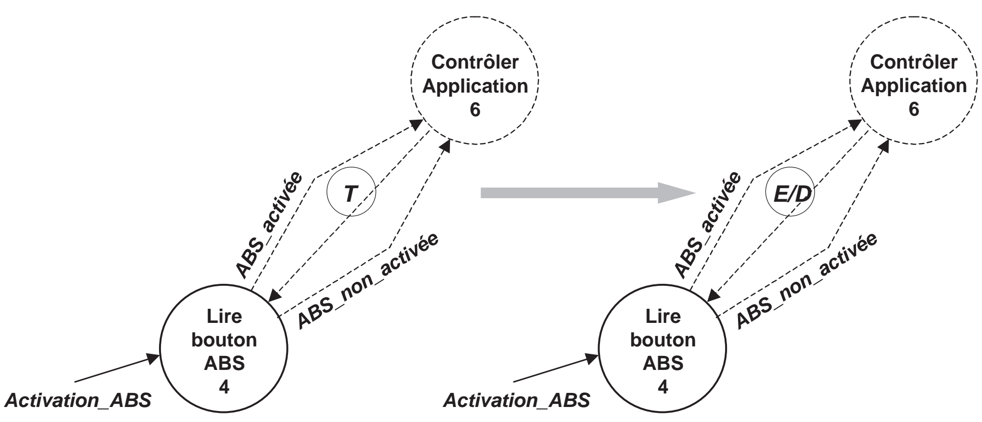


Figure 2.25 – Modification du diagramme préliminaire de l'application « système de freinage automobile » de la figure 2.20 pour garder la cohérence avec le diagramme état/transition de la figure 2.24.

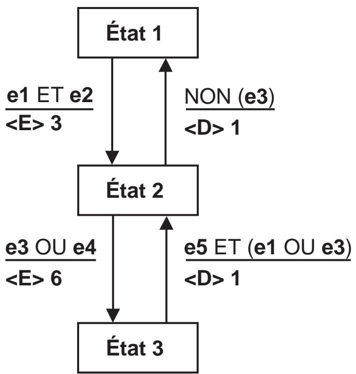


Figure 2.26 – Exemple d'utilisation de combinaisons logiques d'événements.

le fonctionnement de l'application (figure 2.26). Aussi, il est fortement déconseillé d'utiliser des combinaisons « complexes » d'événements comme celui du passage de l'état 3 à l'état 2 de la figure 2.26. Ainsi, se limiter à la combinaison de deux événements semble être une règle de bon compromis : lisibilité et puissance d'expression.

2.5 Spécification des processus primitifs

Le processus du diagramme de contexte étant numéroté « 0 », les processus du diagramme préliminaire seront notés 1, 2, 3… Les processus de ce diagramme flot de données sont ensuite décomposés si nécessaire et décrits par des diagrammes flot de données, appelés diagrammes de décomposition. Ainsi, la décomposition du processus « 1 » donnera naissance à des processus numérotés « 1.1, 1.2, 1.3… ». Lorsque le processus fonctionnel est suffisamment simple, c'est un processus primitif ; il doit être décrit par une spécification sous forme textuelle (spécification procédurale, par précondition et postcondition), tabulaire ou graphique. La méthode SA-RT n'étant pas normalisée et sans indication particulière concernant la spécification des processus primitifs, les utilisateurs décrivent ces processus fonctionnels selon les méthodes usitées dans l'entreprise.
Une des méthodes de spécifications de processus fonctionnels, la plus usitée et adaptée à ce domaine, est celle qui s'appuie sur une spécification procédurale. Celle-ci se décline en 6 mots-clés :

– « E/ données : *Nom_flots_de_données…* » : liste des flots de données en entrée du processus fonctionnel ;
– « E/ événements : *Nom_flots_d'événements…* » : liste des flots d'événements en entrée du processus fonctionnel, c'est-à-dire en général « E/D » ou « T », ou éventuellement les événements produits directement par les bords de modèle ;
– « S/ données : *Nom_flots_de_données…* » : liste des flots de données en sortie du processus fonctionnel ;
– « S/ événements : *Nom_flots_d'événements…* » : liste des flots d'événements en sortie du processus fonctionnel ;

© Dunod – La photocopie non autorisée est un délit.

– « Nécessite : » : liste des contraintes sur les données en entrée du processus fonctionnel ;
– « Entraîne : » description algorithmique du traitement à réaliser.

Dans le cas de l'exemple simple « système de freinage automobile », nous allons considérer le processus fonctionnel 1 « Acquérir demande de freinage ». La description procédurale de ce processus est la suivante :

Processus_fonctionnel : Acquérir demande freinage
E/ données : *Demande_freinage*
E/ événements : *E/D*
S/ données : *Niveau_freinage*
S/ événements : *Freinage*, *Pas_de_freinage*
Nécessite : *Demande_freinage* = 0
Entraîne :
  Faire toujours
    Acquérir *Demande_freinage*
    Émettre *Niveau_freinage* vers Processus 3 (Commander freinage)
    Si *Demande_freinage* > 0
      Alors Émettre *Freinage* vers Processus de contrôle 6 (Contrôler freinage)
      Sinon Émettre *Pas_de_freinage* vers Processus de contrôle 6 (Contrôler freinage)
    Finsi
  Fin faire

Il est aussi possible de faire une spécification de type pré et postcondition sur le modèle suivant :

– « Précondition : » : liste des flots de données ou d'événements en entrée du processus fonctionnel ave les conditions associées.
– « Postcondition » : liste des flots de données ou d'événements en sortie du processus fonctionnel ave les conditions associées.

Donc, dans le cas de l'exemple simple « système de freinage automobile », nous avons le processus fonctionnel 1 « Acquérir demande de freinage » spécifié de la façon suivante :

Processus_fonctionnel : Acquérir demande freinage
Précondition :
  Demande_freinage est fourni et *Demande_freinage* = 0
  E est fourni
Postcondition :
  Niveau_freinage est fourni
  Freinage est fourni si *Demande_freinage* > 0
  Pas_de_freinage est fourni si *Demande_freinage* = 0

2.6 Spécification des données

Au fur et à mesure de la réalisation des différents diagrammes flots de données (diagramme de contexte, diagramme préliminaire et diagrammes de décomposition), un ensemble de données et d'événements est défini. Les données et les événements, qui interviennent à tous les stades du modèle, sont alors réunis dans un dictionnaire de données.

De la même manière que, pour la spécification des processus primitifs, il n'y a pas de méthode imposée, les utilisateurs spécifient les données selon les méthodes personnalisées dans l'entreprise. Pour cet aspect informationnel de SA-RT, nous allons décrire deux méthodes qui peuvent répondre à des besoins simples de spécifications des données : représentation textuelle et représentation graphique. Des méthodes plus complexes et plus formelles existent et sont nécessaires si les données au niveau de l'application ont une place importante en nombre ou en complexité.

2.6.1 Représentation textuelle des données

La spécification des données de manière textuelle peut utiliser une notation précise comme celle basée sur la notation BNF (*Backus-Naur Form*) décrite dans le tableau 2.2.

Tableau 2.2 – Notation B.N.F. permettant de spécifier les données.

Symbole	Signification
= …	Composé de…
……	Commentaire
+	Regroupement sans ordre
{……}	Itération non bornée
n{……}p	Itération de *n* à *p*
(……)	Optionnel
" ……"	Expression littérale
(…l…) ou (…/…)	Ou exclusif

Ainsi, nous pouvons proposer une description de chaque donnée ou événement de la façon suivante :

– Donnée ou événement à spécifier = désignation.

– **Rôle** : description fonctionnelle de la donnée ou de l'événement.

– **Type** : description du codage, domaine de valeurs, etc.

© Dunod – La photocopie non autorisée est un délit.

La désignation de la donnée ou de l'événement peut être faite avec quelques mots prédéfinis : signal en entrée, signal en sortie, événement en entrée, événement en sortie, donnée interne, événement interne ou composition formée d'un ensemble des éléments précédents.

La formalisation de la représentation est limitée, mais elle permet toutefois d'avoir une description homogène et précise de l'ensemble des données ou des événements. En particulier, cette représentation permet d'exprimer la composition entre plusieurs données ou événements. Ainsi, nous pouvons donner les quelques exemples suivants :

– Opérateur de composition « = » et opérateur de séquence « **+** »

Pression = *Pression_air* + *Pression_huile*

– Opérateur de sélection « **[ ]** » et ou exclusif « **|** » ou « **/** »

Pression = [*Pression_calculée*|*Pression_estimée*]

– Opérateur d'itération « **{ }** » et avec bornes

*Pression*s = 1{*Capteur_pression*}5

– Opérateur indiquant le caractère optionnel « **( )** »

Température = *Capteur_température* + (*Valeur_par_défaut*)

Pour l'exemple simple « système de freinage automobile » qui nous intéresse, le dictionnaire de données est assez limité puisque nous avons seulement 6 données et 7 événements. Ainsi, nous obtenons :

Demande_freinage = **Signal en entrée**
    Rôle : donne un signal proportionnel à l'appui sur la pédale de frein
    Type : entrée analogique codée de type entier sur 8 bits, conversion : [0-5V] -> [0-255]
Activation_ABS = **Signal en entrée**
    Rôle : donne la position de l'interrupteur ABS
    Type : entrée numérique de type booléen
Glissement_roue = **Signal en entrée**
    Rôle : donne l'état du glissement de la roue
    Type : entrée numérique de type booléen
Affichage_ABS = **Signal en sortie**
    Rôle : permet d'allumer le témoin d'ABS (lampe ou LED)
    Type : sortie numérique de type booléen
Commande_freinage = **Signal en sortie**
    Rôle : permet de commander le frein
    Type : sortie analogique de type entier codée sur 8 bits, conversion : [0-255] -> [0-10V]
Mise en marche = **Événement en entrée**
    Rôle : indique la mise en marche du véhicule
    Type : entrée numérique de type booléen

Niveau_Freinage = **Donnée interne**
  Rôle : donne la valeur du freinage à transmettre au frein
  Type : entier codée sur 8 bits
Freinage, Pas_de_freinage = **Événements internes**
  Rôle : indique l'appui ou non du conducteur sur la pédale de frein
  Type : type booléen (*vrai* : freinage demandé et *faux* : pas de freinage)
ABS_activé, ABS_non_activé = **Événements internes**
  Rôle : indique l'activation du système ABS par le conducteur
  Type : type booléen (*vrai* : ABS activé et *faux* : ABS non activé)
Glissement, Pas_de_glissement = **Événements internes**
  Rôle : indique l'état du glissement de la roue
  Type : type booléen (*vrai* : glissement et *faux* : pas de glissement)

2.6.2 Représentation graphique des données

Une autre représentation des données et des événements, qui interviennent à tous les stades du modèle, est une représentation graphique de type Jackson. Cette notation est équivalente à la notation BNF. Nous pouvons donner quelques exemples simples correspondant à ceux du paragraphe précédent, soit la représentation d'une séquence ou d'un « ou exclusif » (figure 2.27), soit la représentation d'une itération avec ou sans bornes (figure 2.28).

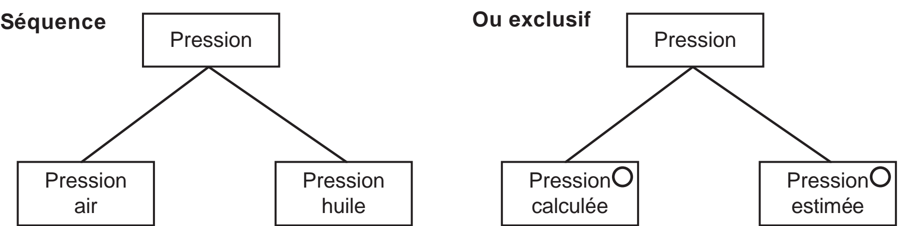


Figure 2.27 – Représentation graphique d'une donnée : séquence et « ou exclusif ».

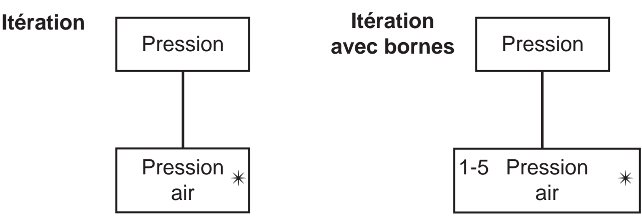


Figure 2.28 – Représentation graphique d'une donnée : itération avec ou sans bornes.

© Dunod – La photocopie non autorisée est un délit.

2.7 Organisation générale de la méthode SA-RT

La description précédente nous a permis de présenter la méthode d'analyse SA-RT dans sa globalité. Les traits essentiels de cette méthodologie hiérarchique descendante résident dans la mise en avant des aspects fonctionnels et comportementaux (exécution) de l'application analysée. L'aspect informationnel, description des données au sens large du terme, est traité de façon minimale.

Le schéma de la figure 2.29 présente l'organisation générale de la méthode SA-RT avec l'enchaînement des différentes étapes et l'ensemble des documents produits. Nous pouvons ainsi décliner :

– Diagramme de contexte : premier diagramme flot de données permettant de décrire l'environnement de l'application à développer.

– Diagramme préliminaire : diagramme flot de données présentant le premier niveau d'analyse fonctionnelle de l'application.

– Diagrammes de décomposition : diagramme flot de données présentant les analyses des processus fonctionnels non primitifs.

– Spécifications des processus fonctionnels primitifs : spécification textuelle des fonctions réalisées par les processus fonctionnels.

– Spécifications des processus de contrôle : diagrammes état/transition décrivant le fonctionnement des processus de contrôle.

– Dictionnaire de données : liste exhaustive des données et des événements utilisés dans la spécification.

Il est évident que l'ensemble de ces diagrammes doit être cohérent par rapport aux deux points de vue :

– Cohérence de l'analyse :
  - Diagramme de contexte, diagramme préliminaire, diagramme de décomposition, spécification d'un processus primitif.
  - Processus de contrôle dans un diagramme « flots de données », diagramme état/transition du processus de contrôle.

– Cohérence de l'enchaînement des différentes étapes
  - Données et événements correspondant entre deux diagrammes flots de données.
  - Événements identiques entre un diagramme flot de données intégrant un processus de contrôle et le diagramme état/transition du processus de contrôle.

Nous pouvons ainsi rappeler les différentes règles d'élaboration des diagrammes flots de données que nous avons vues au cours des paragraphes précédents :

– Un seul processus fonctionnel et pas de processus de contrôle dans un diagramme de contexte.

– Le nombre de processus fonctionnels, composant un diagramme préliminaire, doit être limité de **5** à **9** maximum.

– Les stockages doivent apparaître dans tous les diagrammes où les processus les utilisent.

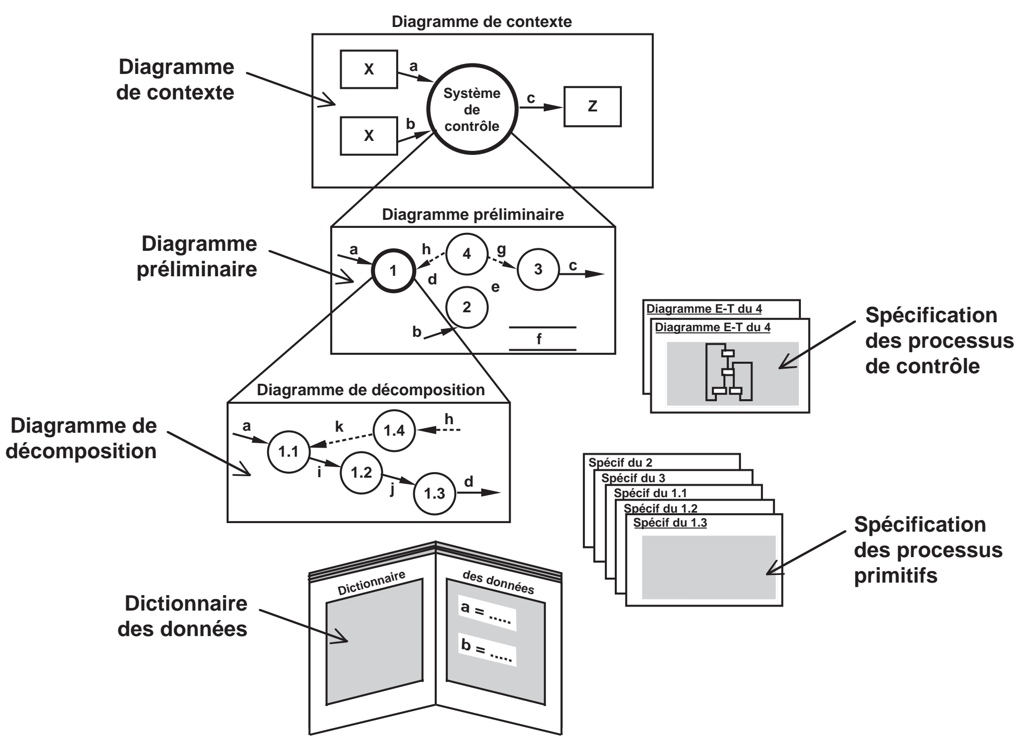


Figure 2.29 – Organisation générale de la méthode SA-RT.

– Un seul ou aucun processus de contrôle par niveau de diagramme : diagramme préliminaire ou diagrammes de décomposition.

– L'ensemble des flots entrants et sortants du processus décomposé doit se retrouver dans le diagramme de décomposition de ce processus avec les mêmes typages.

– La numérotation des différents processus fonctionnels doit intégrer le numéro du processus fonctionnel analysé « N » sous la forme « N.x ».

– Un processus fonctionnel peut ne pas être connecté au processus de contrôle. Dans ce cas, il est supposé être activé au démarrage de l'application et ne jamais s'arrêter.

– Les événements intervenant au niveau de la transition entre les états du diagramme état/transition doivent tous appartenir au diagramme flot de données et réciproquement tous les événements du diagramme flot de données doivent être utilisés dans le diagramme état/transition.

– Pas de flots de données entre les processus fonctionnels et le processus de contrôle.

– Pas de flots d'événements entre les processus fonctionnels.

Les deux dernières règles sont essentielles et, de façon plus générale, nous pouvons décrire les possibilités de relations entre les différentes entités du modèle SA-RT par le tableau 2.3 pour les flots de données et tableau 2.4 pour les flots de contrôle.

© Dunod – La photocopie non autorisée est un délit.

Tableau 2.3 – Liaisons flots de données entre les entités du modèle SA-RT.

De \ Vers	Processus fonctionnel	Processus de contrôle	Unité de stockage	Bord de modèle
Processus fonctionnel	OUI			
Processus de contrôle	NON	NON		
Unité de stockage	OUI	NON	NON	
Bord de modèle	OUI	NON	OUI	–

Tableau 2.4 – Liaisons flots de contrôle entre les entités du modèle SA-RT.

De \ Vers	Processus fonctionnel	Processus de contrôle	Unité de stockage	Bord de modèle
Processus fonctionnel	NON			
Processus de contrôle	OUI	OUI		
Unité de stockage	NON	OUI*	NON	
Bord de modèle	OUI	OUI	OUI*	–

* si le modèle intègre le stockage d'événements.

2.8 Exemples

Nous allons mettre en œuvre cette méthodologie d'analyse SA-RT pour quelques exemples plus complexes que l'exemple décrit jusqu'à présent « système de freinage automobile ». Toutefois, nous nous limitons à la description principale, c'est-à-dire : diagramme de contexte, diagramme préliminaire avec un processus de contrôle et diagramme état/transition du processus de contrôle.

2.8.1 Exemple : gestion de la sécurité d'une mine

■ Présentation du cahier des charges

En plus de la gestion automatisée d'une mine, l'aspect sécurité est le point essentiel de la gestion d'une zone d'extraction de minerai, située en sous-sol, avec une présence humaine. Nous allons limiter notre étude au système de gestion de la sécurité qui concerne principalement le contrôle des deux fluides :

– Eau : les eaux de ruissellement sont évacuées par un ensemble de conduites vers un lieu unique, appelé puisard. Le niveau de ce puisard, qui récupère toutes les

eaux, est contrôlé en permanence avec une évacuation à l'aide de pompes afin de la maintenir entre deux valeurs de niveaux.

– Air : la ventilation des galeries de la mine est effectuée en permanence. Lors de l'extraction, il peut se produire un accès accidentel à une poche de gaz comme du méthane. Fortement toxique, la meilleure protection consiste à procéder à l'évacuation de la zone ou de la mine entière sachant que le volume de méthane n'est, *a priori*, pas connu. D'autre part si le niveau de méthane est élevé, il faut éviter toutes les productions d'étincelles (moteur…).

Nous allons considérer un système simple de gestion de la sécurité de la mine vis-à-vis de ces deux facteurs. Le pilotage de cet ensemble comprend donc les éléments suivants (figure 2.30) :

– un capteur analogique de niveau d'eau, appelé **LS** (*Level Sensor*), pour détecter les deux niveaux limites de régulation, soit le niveau bas (**LLS**, *Low Level Sensor*) et le niveau haut (**HLS**, *High Level Sensor*) ;
– une pompe à eau à débit réglable permettant l'évacuation du puisard ;
– un capteur analogique du taux de méthane contenu dans l'air, appelé **MS** (*Methane Sensor*) ;
– une interface vers l'opérateur pour affichage de l'alarme.

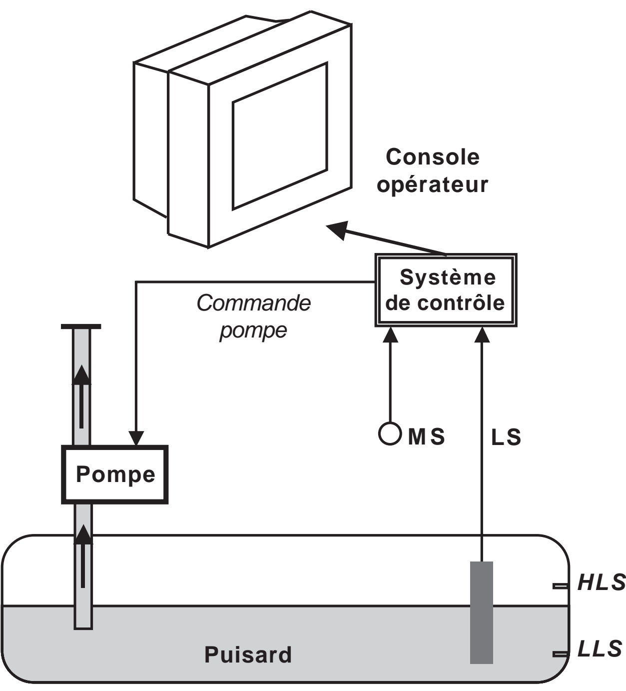


Figure 2.30 – Représentation schématique de l'application de la gestion de l'aspect sécurité d'une mine.

© Dunod – La photocopie non autorisée est un délit.

La mine doit donc fonctionner tant que la sécurité maximale peut être maintenue. Ainsi, les indications générales ou règles de fonctionnement sont les suivantes :

– Règle 1 : La pompe doit être mise en route si le niveau d'eau dépasse le niveau maximum (**LS** > **HLS**). La pompe s'arrête dès que le niveau descend en dessous de la valeur inférieure (**LS** < **LLS**).
– Règle 2 : Une alarme doit être lancée vers la console de l'opérateur dès que le niveau limite du capteur **MS** est franchi afin de pouvoir opérer une évacuation de la mine. Cette valeur limite est appelée **MS_L1** (*Methane Sensor Level 1*).
– Règle 3 : La pompe ne doit pas fonctionner quand le niveau du capteur de méthane (**MS**) est supérieur à une limite fixée **MS_L2** (*Methane Sensor Level 2*) avec la condition **MS_L2** > **MS_L1** afin d'éviter les risques d'explosion.

Dans cet exemple d'application simple, nous sommes donc en présence de deux lignes de régulation : le contrôle du niveau de l'eau dans le puisard (règle 1) et le contrôle du taux de méthane dans l'air (règle 2). L'interaction entre ces deux régulations se situe au niveau de la commande de la pompe pour l'évacuation de l'eau du puisard dont le fonctionnement est lié non seulement au niveau d'eau, mais aussi au taux de méthane (règle 3).

■ Analyse SA-RT

□ Diagramme de contexte

Le premier niveau d'analyse consiste à élaborer le diagramme de contexte de l'application. Ce diagramme, représenté sur la figure 2.31, intègre les quatre bords de modèle correspondant aux deux entrées ou capteurs (capteur de méthane, capteur de niveau d'eau) et aux deux sorties ou actionneurs (pompe, console opérateur). Il est souvent nécessaire d'ajouter un événement de démarrage « *Mise_sous_tension* » délivré par un bord de modèle « interrupteur ». Le processus fonctionnel initial 0 « Gérer sécurité mine » constitue l'application à réaliser. En résumé, en plus de

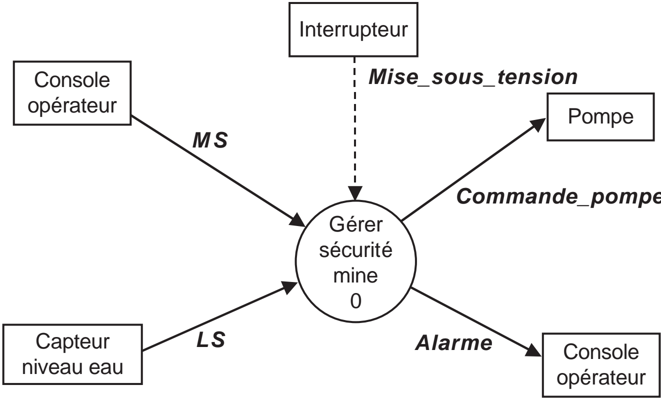


Figure 2.31 – Analyse SA-RT de l'application de la gestion de l'aspect sécurité d'une mine : Diagramme de contexte.

l'événement « *Mise_sous_tension* », nous avons quatre flots de données : deux entrants (*MS*, *LS*) et deux sortants (*Commande_pompe*, *Alarme*). L'ensemble de ces flots doit se retrouver dans le diagramme préliminaire : premier niveau d'analyse du processus fonctionnel 0.

□ Diagramme préliminaire et diagramme état/transition

Le diagramme préliminaire, présenté sur la figure 2.32, donne une analyse ou décomposition fonctionnelle du processus fonctionnel initial 0 « Gérer sécurité mine ». Cette analyse fait apparaître quatre processus fonctionnels de base et un processus de contrôle permettant de séquencer l'ensemble. Nous pouvons vérifier la cohérence des flots de données ou d'événements entrants ou sortants par rapport au diagramme de contexte.

Les processus 1 (Acquérir capteur méthane) et 4 (Afficher alarme) concernent le contrôle du taux de méthane dans l'air avec un processus d'acquisition et de comparaison aux niveaux de consignes (MS_L1, MS_L2) stockés dans une mémoire de stockage « *Niveaux_consignes_méthane* » et un processus de commande pour déclencher l'alarme.

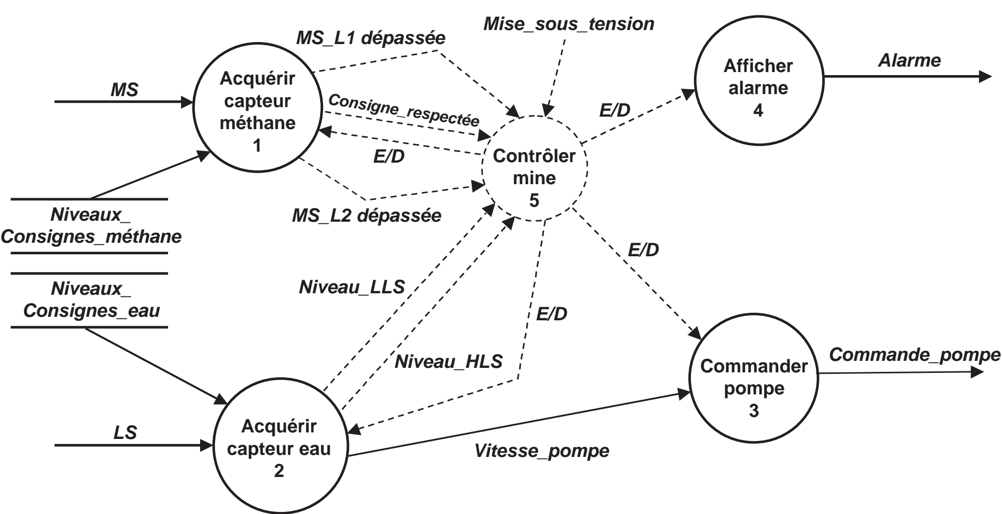


Figure 2.32 – Analyse SA-RT de l'application de la gestion de l'aspect sécurité d'une mine : Diagramme préliminaire.

Les processus 2 (Acquérir capteur eau) et 3 (Commander pompe) concernent la régulation du niveau d'eau dans le puisard avec un processus d'acquisition et de comparaison aux niveaux de consignes (LLS, HLS) stockés dans une mémoire de stockage « *Niveaux_consignes_eau* » et un processus de commande de la pompe. Contrairement à la chaîne de régulation du taux de méthane où les deux processus fonctionnels sont indépendants en termes de données, les deux processus fonctionnels de la chaîne de régulation du niveau d'eau sont liés par le transfert d'une donnée « *Vitesse_pompe* » qui est, par exemple, proportionnelle à la hauteur du niveau d'eau.

© Dunod – La photocopie non autorisée est un délit.

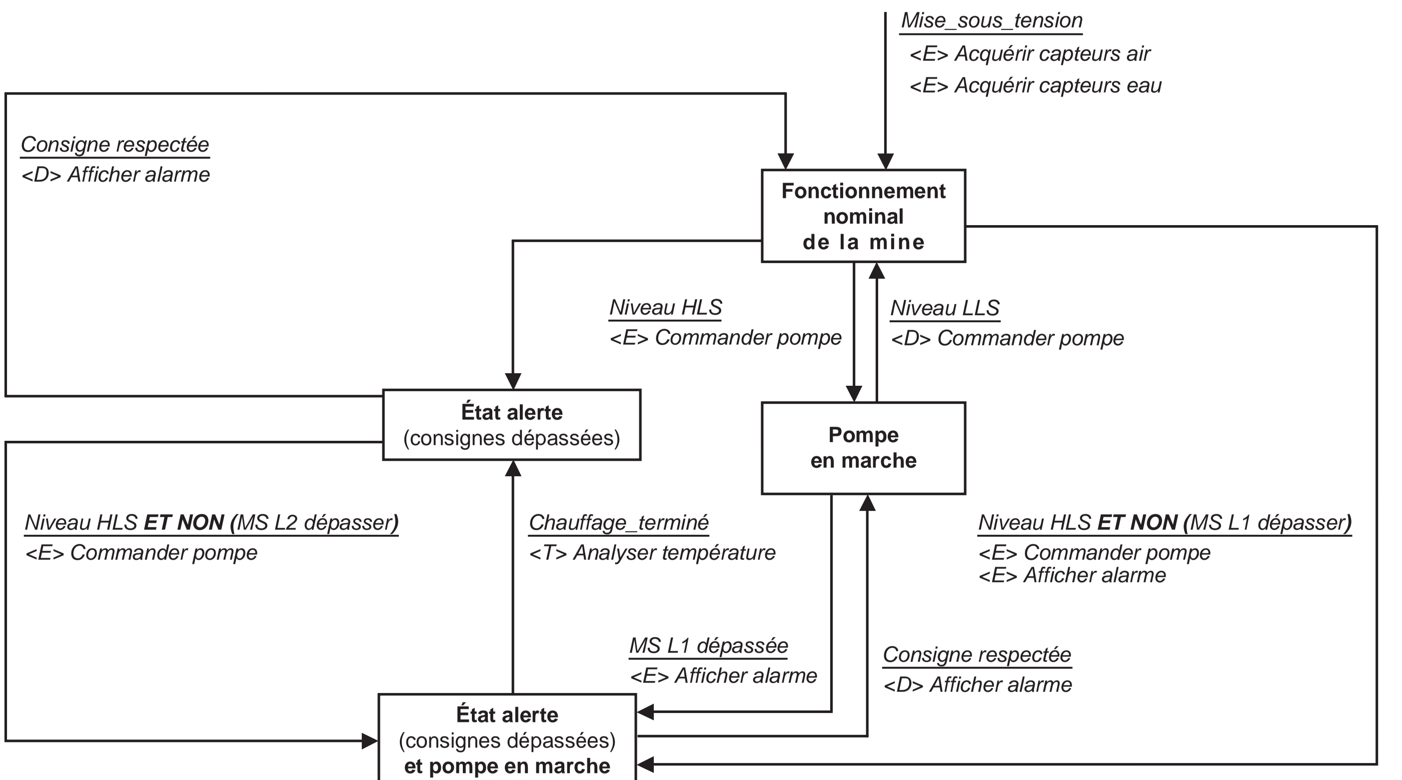


Figure 2.33 – Analyse SA-RT de l'application de la gestion de l'aspect sécurité d'une mine : Diagramme état/transition.

Le processus de contrôle est lié aux différents processus fonctionnels par des événements qui sont mis en place en même temps que la réalisation du diagramme état/transition de la figure 2.33. Ce diagramme état/transition, description du fonctionnement du processus de contrôle 5 (Contrôler mine), comprend quatre états :

– fonctionnement nominal de la mine (niveau d'eau inférieur à HLS et taux de méthane inférieur à MS_L1) ;
– pompe en marche (niveau d'eau supérieur à LLS) ;
– état alerte (consigne MS_L1 dépassée et niveau d'eau inférieur à HLS, ou consigne MS_L2 dépassée et niveau d'eau quelconque) ;
– état alerte (consigne MS_L1 dépassée) et pompe en marche.

Nous pouvons remarquer que, dans ce diagramme état/transition complexe, nous avons utilisé dans certains cas des combinaisons logiques de deux événements pour passer d'un état à l'autre, par exemple « *MS_L2_dépassée* **OU** *Niveau_LLS* » pour passer de l'état « État alerte et pompe en marche » à l'état « État alerte ».
D'autre part, ce diagramme état/transition fait l'hypothèse pour la surveillance du taux de méthane que les événements « *MS_L1_dépassée* » et « *MS_L2_dépassée* » se produisent toujours dans l'ordre cité et, lors du retour à la situation normale, l'événement « *Consigne_respectée* » est émis.

2.8.2 Exemple : pilotage d'un four à verre

■ Présentation du cahier des charges

Un four pour la fabrication du verre fonctionne de façon continue aussi bien du point de vue de l'approvisionnement en matières premières (sables) que du point de vue de l'utilisation du produit (verre). En effet, le four doit rester en fonctionnement permanent avec un niveau toujours suffisant de matières fondues à température constante, une évacuation du trop plein étant prévue en cas d'attente prolongée d'utilisation du verre. Cette application a été simplifiée afin de limiter l'analyse.
Le contrôle-commande de cette application est fait par l'intermédiaire de **3 capteurs** (capteur de température, capteur de niveau du four et capteur de détection de l'arrivée de matières premières) et de **2 actionneurs** (commande d'approvisionnement en matières premières, chauffage du four). Nous avons donc comme précédemment deux chaînes de régulation : température et approvisionnement en sable.
Une représentation schématique de cette application est présentée sur la figure 2.34.
L'acquisition de la température, à partir de capteurs de type thermocouple, doit se faire à des moments réguliers en utilisant l'horloge temps réel interne du système.
Le traitement du « signal température » permet de faire un calcul précis de la température (approximation polynomiale correspondant au thermocouple) et lance une tâche de commande de chauffage si la température du four est inférieure à la température de consigne. Le principe de chauffage du four se fait à partir d'ondes hautes fréquences pulsées. Ce chauffage est effectué pendant un temps fixé court mais avec une intensité qui peut dépendre du chauffage nécessaire.
L'acquisition du niveau de matière est liée à l'interruption générée de façon apériodique par les tombées successives mais non régulières du sable détectées par le capteur.
Cette détection est réalisée par un capteur tout ou rien comme une cellule photo-

© Dunod – La photocopie non autorisée est un délit.

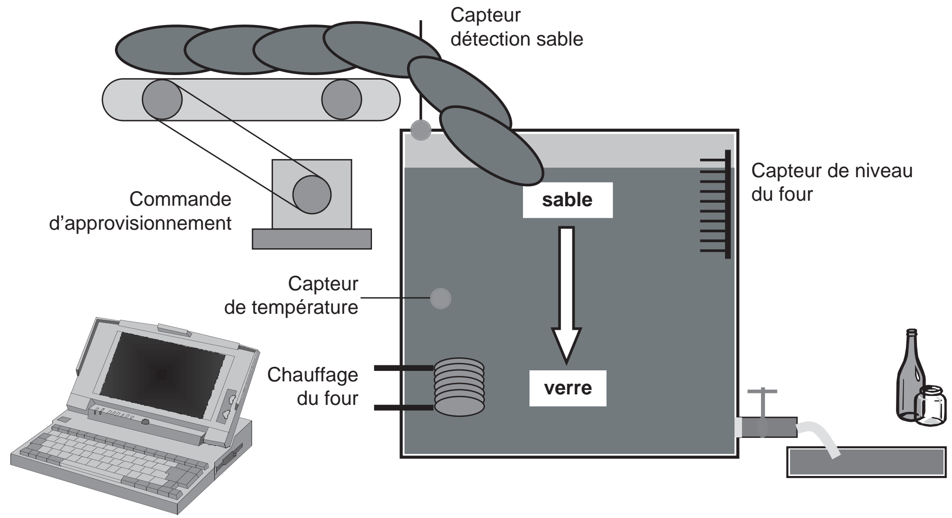


Figure 2.34 – Représentation schématique de l'application de la gestion d'un four à verre.

électrique. Le paramètre « niveau de matière » va impliquer l'approvisionnement ou non en matières premières en commandant la vitesse d'approvisionnement en fonction du paramètre « niveau du four ». Mais cette régulation dépend aussi de la valeur de la température. En effet, afin d'éviter une solidification du sable fondu, il est nécessaire de limiter l'apport en matières premières si la température n'est pas suffisante.

■ Analyse SA-RT

□ Diagramme de contexte

La première étape d'analyse consiste à élaborer le diagramme de contexte de l'application. Ce diagramme, représenté sur la figure 2.35, intègre les six bords de modèles correspondant aux trois entrées ou capteurs (capteur de température – thermocouple, capteur de niveau de sable, capteur tout ou rien de détection d'arrivée du sable) et aux deux sorties ou actionneurs (approvisionnement en sable, commande du four de chauffage). Le dernier bord de modèle correspond à la console opérateur qui fournit les deux événements : « *Arrêt* » et « *Marche* ». Ces événements ne sont utilisés que pour le démarrage de l'application et éventuellement son arrêt. Le processus fonctionnel initial 0 « Piloter four à verre » constitue l'application à étudier. En résumé, en plus des deux événements précédemment cités, nous avons cinq flots de données : trois flots entrants (*Température*, *Niveau_sable*, *Arrivée_sable*) et deux flots sortants (*Sable*, *Chauffage*). L'ensemble de ces flots doit se retrouver dans le diagramme préliminaire : premier niveau d'analyse du processus fonctionnel 0.

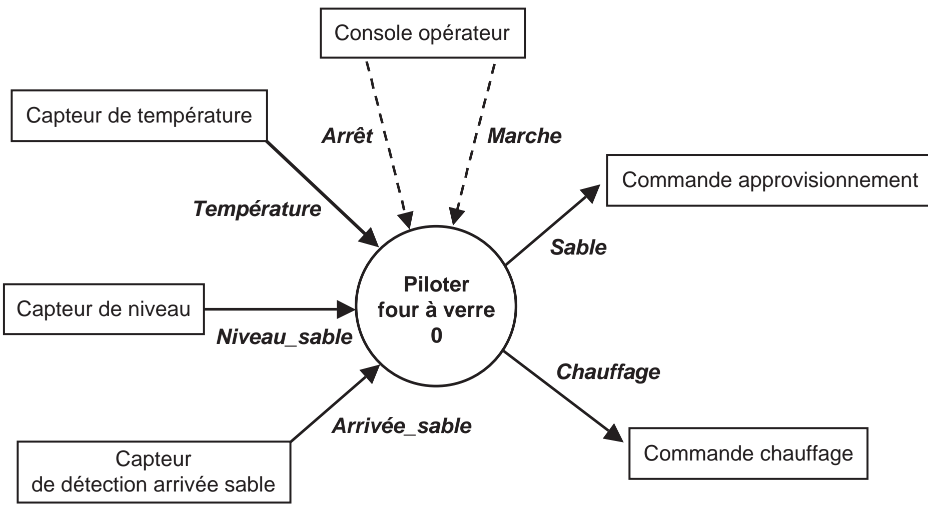


Figure 2.35 – Analyse SA-RT de l'application de la gestion d'un four à verre : Diagramme de contexte.

☐ Diagramme préliminaire et diagramme état/transition

Le diagramme préliminaire, présenté sur la figure 2.36, donne une analyse ou décomposition fonctionnelle du processus fonctionnel initial 0 « Piloter four à verre ». Cette analyse fait apparaître six processus fonctionnels de base et un processus de contrôle permettant de séquencer l'ensemble. Nous pouvons vérifier la cohérence des flots de données ou d'événements entrants ou sortants par rapport au diagramme de contexte.

Les différents processus fonctionnels correspondent aux deux chaînes de régulation : température (processus fonctionnels 1 à 3) et approvisionnement en sable (processus fonctionnels 4 à 6). La régulation de la température suit exactement la décomposition fonctionnelle générique que nous avons vue (figure 2.14). Ainsi, les trois processus fonctionnels de base existent : acquisition (1 – Acquérir température), traitement (2 – Analyser température) et commande (3 – Chauffer four). Les deux unités de stockage sont utilisées dans les deux cas classiques : soit pour la mémorisation d'une constante (*Température_consigne*) soit pour une donnée partagée (*Température_mesurée*). Dans ce dernier cas, nous pouvons noter que l'unité de stockage est utilisée deux fois dans le diagramme et donc comporte une « * ».

Dans le cas de la régulation du niveau du sable, les trois processus fonctionnels mis en œuvre ne correspondent pas exactement au modèle de décomposition générique précédent ; mais nous avons uniquement deux processus fonctionnels : acquisition (5 – Acquérir niveau), traitement et commande (6 – Analyser besoin sable). Nous pouvons remarquer que ce dernier processus utilise pour élaborer la décision de commande trois données : *Niveau_consigne* (constante placée dans une unité de stockage), *Niveau_mesurée* (donnée fournie directement par le processus 5) et *Température_mesurée* (unité de stockage partagée avec la chaîne de régulation de la

© Dunod – La photocopie non autorisée est un délit.

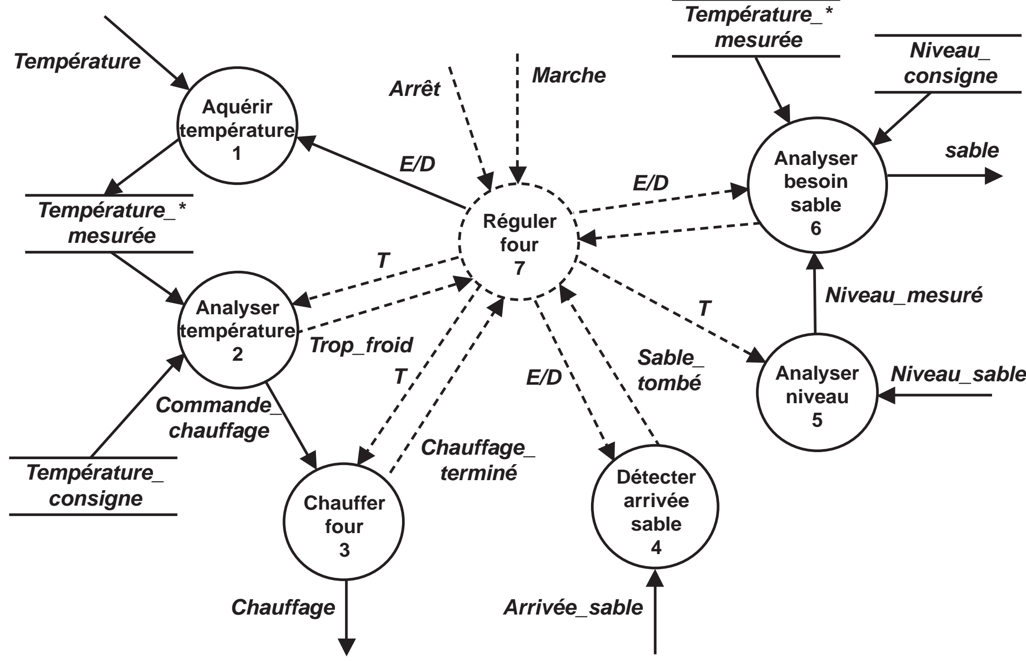


Figure 2.36 – Analyse SA-RT de l'application de la gestion d'un four à verre : Diagramme préliminaire.

température). Le troisième processus utilisé dans cette régulation de niveau (4 – Détecter arrivée sable) correspond en fait à un processus de déclenchement qui est activé sur interruption liée à la donnée « *Arrivée_sable* ».

Le processus de contrôle (7 – Réguler four) est lié aux différents processus fonctionnels par des événements qui sont mis en place en même temps que la réalisation du diagramme état/transition de la figure 2.37. Ce diagramme état/transition, description du fonctionnement du processus de contrôle 7, comprend quatre états :

– état repos (attente de fonctionnement ou arrêt du four) ;
– fonctionnement nominal (four en fonctionnement et attente des événements pour effectuer les régulations soit de température, soit de niveau de sable) ;
– chauffage du four (la température de consigne n'étant pas atteinte, un cycle de chauffage est lancé) ;
– régulation du niveau de sable (une interruption due à la chute d'un paquet de sable lance la régulation du niveau du four).

Nous pouvons remarquer que, dans ce diagramme état/transition simple, nous avons utilisé un état stationnaire (Fonctionnement nominal) dans lequel l'application reste et deux autres états plus transitoires dans lesquels l'application se situe en cas de régulation de l'un des deux paramètres.

D'autre part nous pouvons noter la difficulté concernant la régulation du niveau de sable ; en effet l'ajustement de l'approvisionnement en sable ne peut se produire

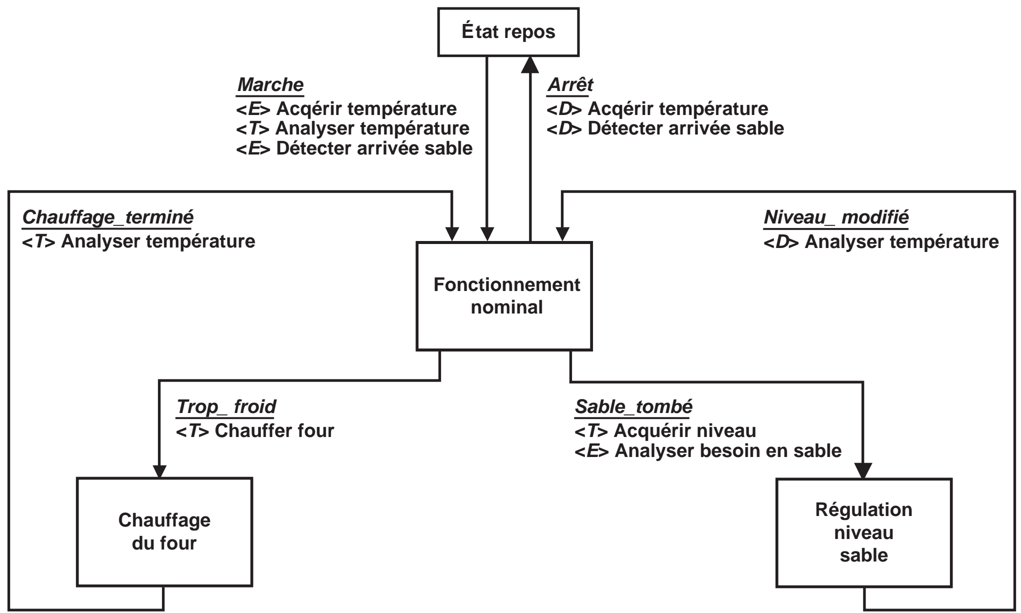


Figure 2.37 – Analyse SA-RT de l'application de la gestion d'un four à verre : Diagramme état/transition.

que si la mesure du niveau est effectuée, c'est-à-dire que le processus fonctionnel « 5 – Acquérir niveau » est activé. Pour cela, il est nécessaire que la donnée « *Arrivée_ sable* » se produise pour qu'elle soit transformée en événement « *Sable_tombé* » par le processus fonctionnel « 4 – Détecter arrivée sable ». Dans le cas contraire, la régulation du niveau de sable ne peut s'exécuter, conduisant à un dysfonctionnement de l'application. Ce problème est étudié au niveau de la conception décrite dans le chapitre 3.

2.8.3 Exemple : Commande d'un moteur à combustion

■ Présentation du cahier des charges

Le contrôle-commande d'un moteur à combustion est devenu de plus en plus complexe au fur et à mesure des besoins en rendement, consommation et pollution. En effet, en plus de fournir une puissance mécanique en fonction de l'appui sur la pédale de l'accélérateur, il est nécessaire d'avoir une optimisation de la consommation du véhicule en contrôlant les différents paramètres de la combustion (pression air, température, mélange, allumage, etc.). D'autre part, la pollution du véhicule doit être minimisée. Les différents états du moteur peuvent être présentés au conducteur afin de le prévenir de sa consommation excessive ou de la pollution de son véhicule pour qu'il effectue un changement dans son mode de conduite. L'ensemble de ces paramètres moteur est géré par un calculateur spécifique. Actuellement, un véhicule possède de nombreux calculateurs dédiés à des fonctions très diverses (freinage ABS, gestion moteur, éclairage, climatisation, etc.). Ces différents calculateurs commu-

© Dunod – La photocopie non autorisée est un délit.

niquent entre eux par un bus de terrain comme CAN (voir chapitre 4) afin de partager des informations et gérer ainsi le véhicule de façon cohérente.
Cette application de commande d'un moteur à combustion est représentée schématiquement sur la figure 2.38. Le contrôle-commande de cette application est fait par l'intermédiaire de **sept capteurs** (pédale accélérateur, température air, pression air, température eau, rotation vilebrequin et deux capteurs de pollution) et de **quatre actionneurs** (injection essence, allumage, admission air, réinjection gaz échappement ou brulés). Le calculateur est donc aussi relié au bus de communication CAN.

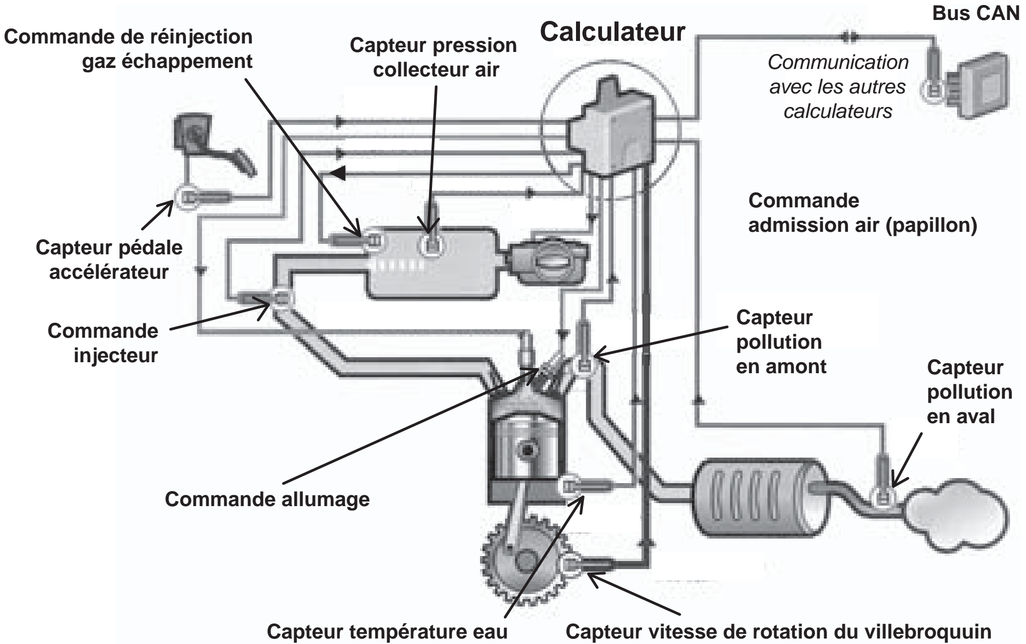


Figure 2.38 – Représentation schématique de l'application de la commande d'un moteur à combustion.

Excepté les deux capteurs de pollution amont et aval, la loi de régulation du moteur à combustion prend en compte l'ensemble des données d'entrée et élabore les différentes sorties de commande. Nous n'analyserons pas cette loi de commande qui présente une relative complexité et correspond à une spécificité constructeur pour un type de moteur donné. Ainsi, l'élaboration de la loi de commande du moteur à combustion est considérée comme une « boîte noire » fonctionnelle qui utilise un ensemble de données en entrée (*Paramètres_moteur_entrée*) et qui fournit des données en sortie (*Paramètres_moteur_sortie*). Les données sur la pollution ne sont pas utilisées dans ce calculateur ; mais elles sont fournies par l'intermédiaire du réseau de communications à un autre calculateur, par exemple, pour affichage.

■ Analyse SA-RT

□ Diagramme de contexte

Le diagramme de contexte de l'application est représenté sur la figure 2.39. Il donne les 11 bords de modèles correspondant aux sept entrées ou capteurs et aux quatre sorties ou actionneurs, énumérés précédemment. Nous avons ajouté un bord de modèle correspondant à la connexion au bus CAN et un bord de modèle représentant l'action du conducteur. Le dernier bord de modèle fournit les deux événements : « *Arrêt* » et « *Marche* ». Pour le bord de modèle du bus CAN, nous supposons que les communications bidirectionnelles sont identifiées : en sortie (*Com_S*) et en entrée (*Com_E*).

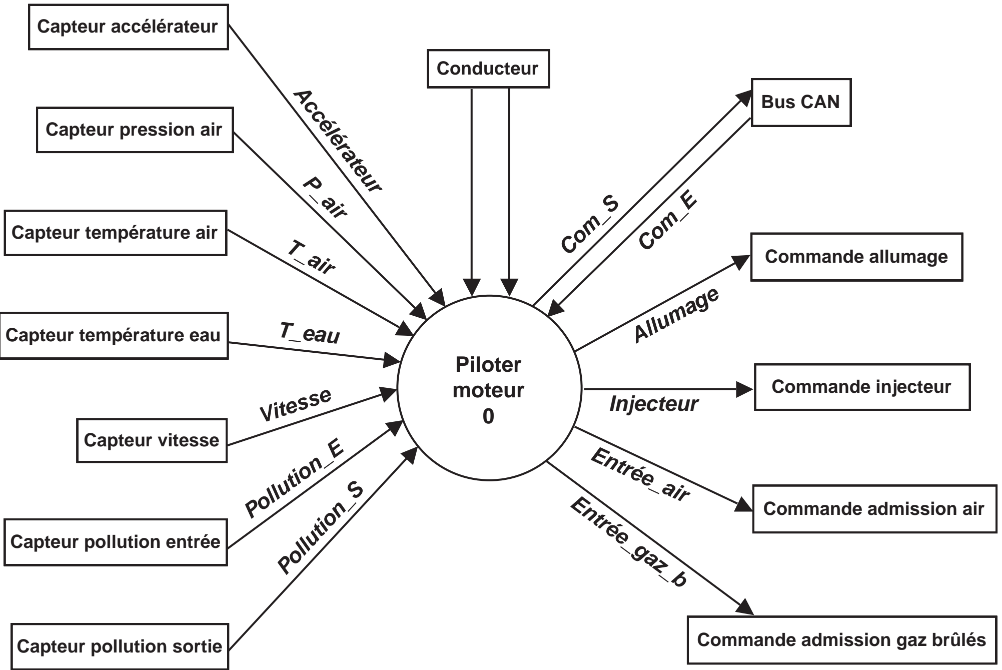


Figure 2.39 – Analyse SA-RT de l'application de la commande d'un moteur à combustion : Diagramme de contexte.

Le processus fonctionnel initial 0 « Piloter moteur » constitue l'application à étudier. En résumé, en plus des deux événements précédemment cités (*Arrêt*, *Marche*), nous avons 13 flots de données : huit entrants (*Accélérateur*, *P_air*, *T_air*, *T_eau*, *Vitesse*, *Pollution_E*, *Pollution_S*, *Com_E*) et cinq sortants (*Allumage*, *Injecteur*, *Entrée_air*, *Entrée_gaz_b*, *Com_S*). L'ensemble de ces flots doit se retrouver dans le diagramme préliminaire : premier niveau d'analyse du processus fonctionnel 0.

© Dunod – La photocopie non autorisée est un délit.

☐ Diagramme préliminaire et diagramme état/transition

Le diagramme préliminaire, présenté sur la figure 2.40, donne une analyse ou décomposition fonctionnelle du processus fonctionnel initial 0 « Piloter moteur ». Cette analyse fait apparaître neuf processus fonctionnels de base et un processus de contrôle permettant de séquencer l'ensemble. Malgré le regroupement de certaines fonctions d'acquisition ou de commande, le nombre de processus fonctionnel de cette première décomposition correspond à la limite fixée pour avoir une bonne compréhension globale du diagramme flots de données. Nous pouvons remarquer que les processus fonctionnels 6, 7 et 8 auraient pu être regroupés car ils sont déclenchés en même temps. Ce travail de regroupement est effectué en partie au niveau de la conception étudiée dans le chapitre 3.

Nous pouvons vérifier la cohérence des flots de données ou d'événements entrants ou sortants par rapport au diagramme de contexte de la figure 2.39. Afin d'augmenter la lisibilité du diagramme, nous avons utilisé des unités de stockage de données complexes, ou encore appelées énumérées, dans le sens où elles intègrent un ensemble ou groupe de données. Comme nous l'avons déjà présenté, deux unités de stockages renferment les paramètres en entrée du moteur, excepté les données pollutions (*Paramètres_moteur_entrée*) et les paramètres en sortie du moteur (*Paramètres_moteur_sortie*).

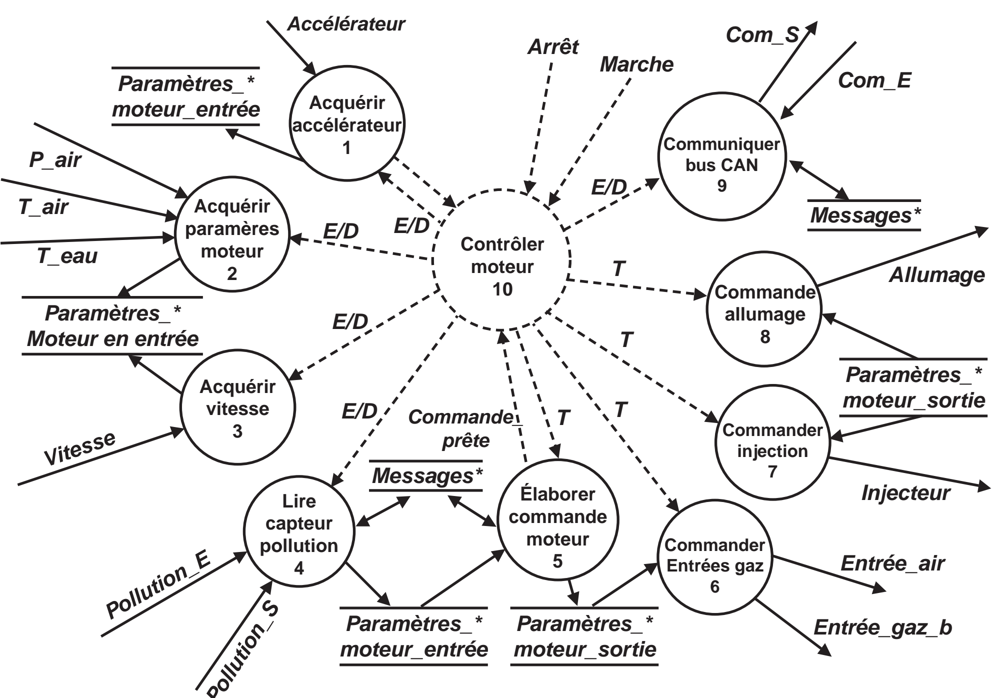


Figure 2.40 – Analyse SA-RT de l'application de la commande d'un moteur à combustion : Diagramme préliminaire.

Une autre unité de stockage est dédiée aux messages (*Messages*). Cette unité de stockage est un peu particulière par rapport aux autres unités de stockage. En effet, en entrée, nous avons une file gérée de façon FIFO ou à priorité et, en sortie, la file est gérée de manière FIFO.

Remarquons que toutes ces unités de stockages sont dupliquées au niveau de ce diagramme et donc notées avec une « * ».

Le processus de contrôle (10 – Contrôler moteur) est lié aux différents processus fonctionnels par des événements qui sont mis en place en même temps que la réalisation du diagramme état/transition de la figure 2.41. Nous avons simplifié le fonctionnement de ce processus de contrôle en supposant que les processus fonctionnels d'acquisition étaient lancés au début de l'exécution (1, 2, 3 et 4) et fournissaient les données de façon périodique ; en particulier le processus fonctionnel « 1 – Acquérir accélérateur » fournit régulièrement l'événement *État_accélérateur* qui déclenche l'élaboration de la loi de commande à partir de toutes les données d'entrée acquises. Lorsque la commande est prête, le processus « 5 – Élaborer commande moteur » déclenche l'ensemble des processus fonctionnels de commande (6, 7 et 8). Enfin, le processus fonctionnel 9 de communication gère périodiquement les messages au niveau réception et émission.

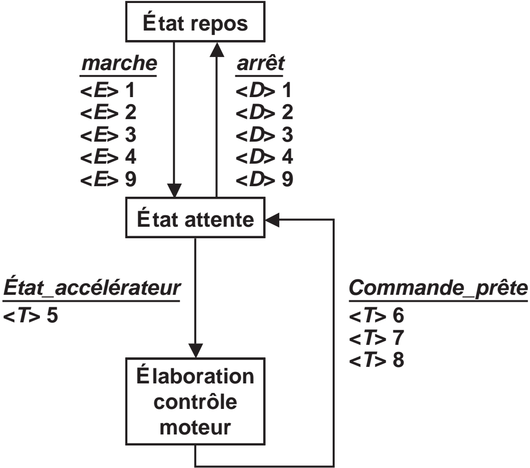


Figure 2.41 – Analyse SA-RT de l'application de la commande d'un moteur à combustion : Diagramme état/transition.

Nous pouvons remarquer que, dans ce diagramme état/transition très simple, nous avons utilisé uniquement deux états actifs en plus de l'état repos : Un état attente qui est périodiquement interrompu par l'événement « *État_accélérateur* » et un état correspondant à l'élaboration de la loi de commande du moteur à combustion.

© Dunod – La photocopie non autorisée est un délit.

2.9 Extensions de la méthode SA-RT

2.9.1 Extensions de la syntaxe graphique de la méthode SA-RT

Comme nous l'avons déjà dit précédemment en introduction de cette méthode d'analyse SA-RT, la description non formelle des applications est très abstraite et ne permet pas dans beaucoup de cas de spécifier précisément certaines fonctions ou certaines données. D'autre part, la méthode n'étant pas normalisée, il est tout à fait possible d'étendre la méthode afin d'offrir plus d'expressivité. Aussi de nombreuses entreprises ont adapté la méthode à leurs besoins et ajouté des éléments graphiques pour exprimer des spécifications plus précises. Nous présentons dans ce chapitre deux de ces extensions très utilisées : une extension concernant les flots de données et une extension pour les événements. Ces extensions sont intégrées dans la méthode « ESML++ » développée dans le cadre de la société Boeing. Cette méthode, utilisée pour les développements des applications de contrôle-commande du domaine de l'avionique, est une méthode SA-RT enrichie.

■ Extensions liées aux flots de données

Une syntaxe graphique plus complète permet de préciser les flots de données discrets ou continus. Ainsi, l'analyse des flots de données permet et, donc oblige, de distinguer un flot de données discret (arc orienté simple comme précédemment) et un flot de données continu (arc orienté avec une double flèche). Mais, dans ce cas, où la richesse d'expression des données véhiculées par ces flots est augmentée, il est nécessaire de préciser la sémantique attachée à cette nouvelle description. Or deux types de sémantique peuvent être attachés aux flots de données (figure 2.42) :

– Sémantique 1 :
  - Flot de donnée discret : valeur discrète de donnée (type booléen) ;
  - Flot de donnée continu : valeur continue de la donnée (type entier ou réel).
– Sémantique 2 :
  - Flot de donnée discret : donnée consommable ou lisible une fois (existence de la donnée à des temps discrets) ;

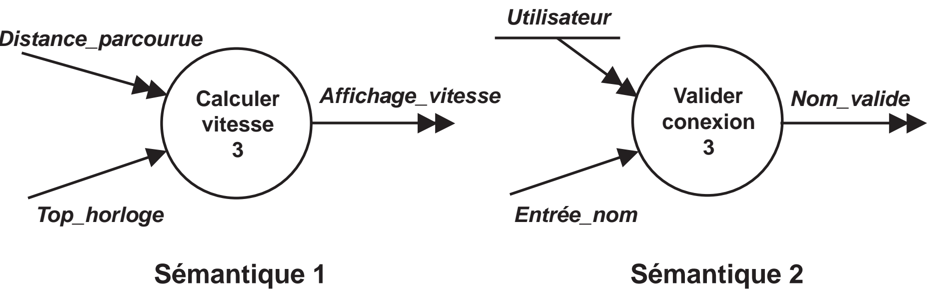


Figure 2.42 – Extension de la syntaxe graphique des flots de données selon deux sémantiques.

• Flot de donnée continu : donnée continuellement disponible (existence permanente de la donnée).

Extensions liées aux événements

Comme pour les flots de données, une syntaxe graphique plus complète permet de préciser les flots d'événements discrets ou continus. Ainsi, l'analyse des flots d'événements permet et, donc oblige, de distinguer un flot d'événements discret (arc orienté simple comme précédemment) qui concerne des événements consommables ou lisibles une seule fois (existence à des temps discrets). De même un flot d'événements continu (arc orienté avec une double flèche) décrit un événement continuellement disponible (existence permanente).
Il est important de noter que les événements prédéfinis « E, D, T » sont des événements discrets, envoyés à chaque fois pour activer ou arrêter un processus fonctionnel. En revanche, un flot d'événements continu permet au processus de contrôle de tester en permanence le résultat d'un processus fonctionnel qui est indépendant du processus de contrôle (figure 2.43).

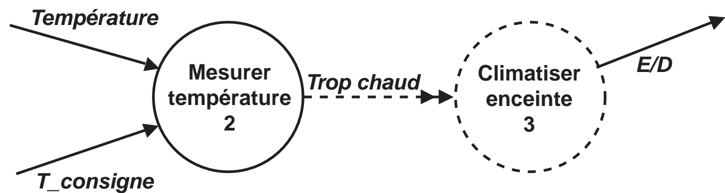


Figure 2.43 – Extension de la syntaxe graphique des flots d'événements.

De même, pour les stockages de données, le **stockage d'événements** modélise le besoin de mémorisation d'un événement de façon à ce que son occurrence soit utilisée plusieurs fois. Comme le flot d'événements auquel il est étroitement associé, il est nommé par une étiquette ou label explicite formé de :

Étiquette_Stockage_événements = nom (+ qualifiant)

et il est représenté par deux traits parallèles en pointillés (figure 2.44).

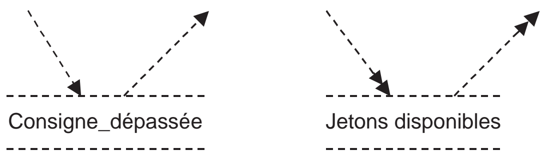


Figure 2.44 – Extension de la syntaxe graphique de SA-RT : Stockages d'événements.

© Dunod – La photocopie non autorisée est un délit.

2.9.2 Méthode SA-RT et méthode formelle : projet IPTES

■ Introduction

La méthode SA-RT, qui vient d'être présentée dans les sections précédentes, est une méthode de spécification simple avec un langage graphique très compréhensible par un utilisateur. Cette facilité d'expression a un inconvénient majeur, celui de pouvoir conduire à des ambiguïtés. En effet, lors de l'analyse d'une application, basée sur la méthode SA-RT, la réalisation des différents diagrammes flots de données reflète l'idée du concepteur au travers d'un langage simple, limité et très abstrait. Ainsi, deux utilisateurs peuvent avoir exprimé deux analyses différentes sous la forme d'un même diagramme flots de données SA-RT. Pour lever cette ambiguïté, il est nécessaire de disposer d'un outil formel qui offre un moyen d'expression précis et qui peut être validé.

Ainsi, il est possible d'utiliser en complément ou en parallèle de cette méthodologie SA-RT, un modèle formel comme des automates à états finis ou des réseaux de Petri. Ces modèles présentent l'avantage d'être rigoureux au niveau de l'expression et d'être analysable mathématiquement. En revanche, ces modèles sont en général complexes et d'une lisibilité faible. Aussi le projet européen IPTES (*Incremental Prototyping Technology for Embedded Real-Time Systems*), dont les résultats ont été présentés en 1998, a voulu donner une sémantique à la méthode SA-RT basée sur les réseaux de Petri. Nous pouvons schématiser cette association par le graphique de la figure 2.45. Ainsi, la méthode SA-RT peut être vue comme l'interface de la méthode d'analyse globale avec un langage graphique de haut niveau pour exprimer la spécification, c'est l'interface utilisateur. Directement liée à ce modèle graphique, nous trouvons la correspondance dans le modèle formel apporté par les réseaux de Petri qui constituent le noyau fonctionnel, c'est-à-dire la partie permettant d'exécuter ce modèle graphique représentant la spécification.

L'utilisateur décrit sa spécification avec la syntaxe graphique SA-RT et, de façon automatique, le modèle formel, basé sur les réseaux de Petri, se construit. Cela permet ensuite une analyse mathématique de la spécification : simulation et vérification. Pour atteindre ce but, il est nécessaire d'avoir une traduction unique d'un modèle vers l'autre.

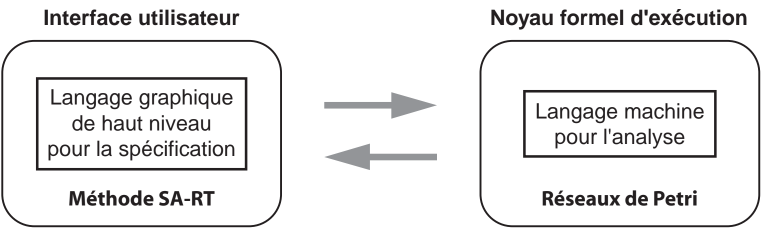


Figure 2.45 – Représentation schématique de l'association d'un modèle formel avec la méthode SA-RT.

■ Principes généraux

La méthode SA-RT, utilisée pour ce projet, correspond à celle décrite précédemment avec toutefois l'extension à des flots de données discrets et continus suivant la sémantique 2 (§ 2.9.1).

Les réseaux de Petri ont de nombreuses variantes : abréviations et extensions. Les réseaux de Petri choisis pour ce projet sont des réseaux de Petri autonomes ayant les principales caractéristiques suivantes :

– Association des données et des événements aux jetons.
– Jeton correspondant à une variable typée.
– Places typées.
– Association à chaque transition :
  • Un prédicat lié aux places en amont de la transition ;
  • Une action.
– Transition temporisée suivant le modèle Merlin et Farber [∂_{min}, ∂_{max}] où t_0 est la date de sensibilisation de la transition :
  • Franchissement de la transition après le temps $t_0 + \partial_{min}$;
  • Franchissement de la transition avant le temps $t_0 + \partial_{max}$.

En partant de ces hypothèses sur les modèles utilisés, nous avons donc une traduction ou correspondance entre tous les éléments syntaxiques du modèle SA-RT et un réseau de Petri. Toutefois, afin de simplifier cette présentation, les éléments traduits font abstraction de la modélisation temporisée des transitions qui a été présentée précédemment, c'est-à-dire que toutes les transitions présentées dans la suite sont considérées comme des transitions immédiates, soit [0,0] avec la notation du modèle Merlin et Farber.

En premier, pour les flots de données, nous trouvons le modèle d'un flot de données discret, qui sont des données consommables ou lisibles une fois, sur la figure 2.46 et le modèle d'un flot de données continu, qui sont des données continuellement disponibles, sur la figure 2.47. Ces deux représentations réseaux de Petri nécessitent deux places (Vide et Valeur) et trois transitions (*Écrire_donnée_vide*, *Écrire_donnée* et *Lire_donnée*). La place « Vide » désigne une donnée qui n'a jamais été produite

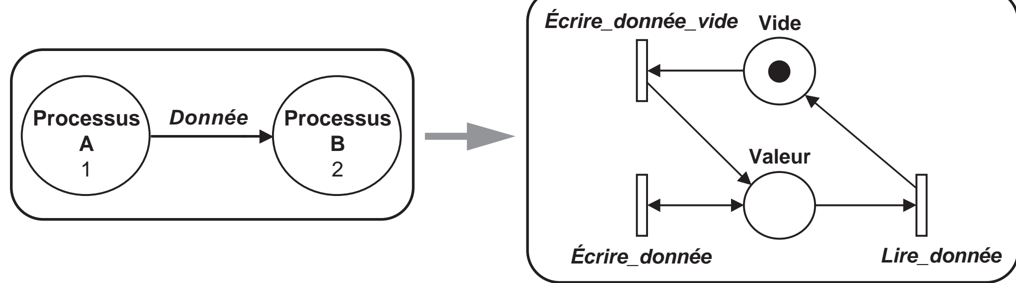


Figure 2.46 – Traduction d'un flot de données discret de la méthode SA-RT par un réseau de Petri.

© Dunod – La photocopie non autorisée est un délit.

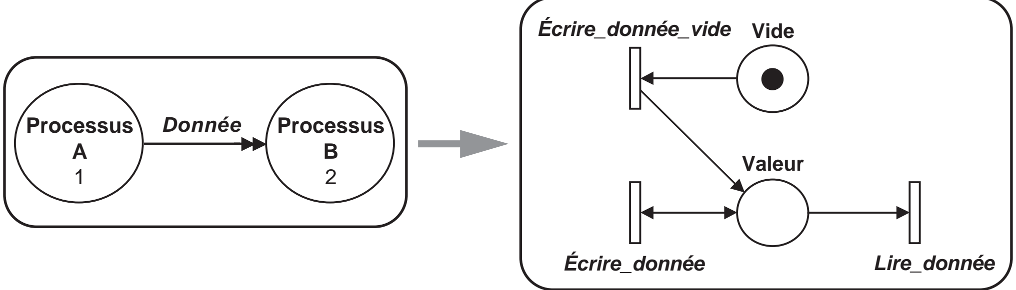


Figure 2.47 – Traduction d'un flot de données continu de la méthode SA-RT par un réseau de Petri.

(cas flot continue) ou qui a été consommée (cas du flot discret). Cette place possède un jeton à l'initialisation. Dans le cas du flot de données continu, à la première écriture de la donnée la place « Vide » perd son jeton et la place « Valeur » sera ensuite toujours marquée. En revanche, dans le cas du flot de données discret, le franchissement de la transition « *Lire_donnée* » fait passer le jeton de la place « Valeur » à la place « Vide ».

La modélisation de l'unité de stockage, représentée sur la figure 2.48, est plus simple d'un point de vue réseau de Petri. Ainsi, nous avons une place « Stockage », intégrant un jeton représentant la donnée, et les deux transitions correspondant à l'écriture et à la lecture (*Écrire_donnée* et *Lire_donnée*). En revanche, dans le but d'obtenir un modèle formel précis, une sémantique particulière est associée au jeton décrivant l'état de la donnée selon la lecture ou non de cette donnée. L'enregistrement dans l'unité de stockage comprend deux champs : la donnée elle-même et son état (tableau 2.5). Après chaque écriture, la donnée a un état déclaré nouveau qui peut donc être utilisé par le processus fonctionnel qui lit cette donnée.

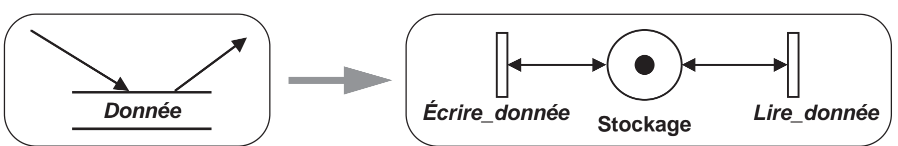


Figure 2.48 – Traduction d'une unité de stockage de la méthode SA-RT par un réseau de Petri.

Tableau 2.5 – Caractérisation du contenu de l'enregistrement dans l'unité de stockage du modèle SA-RT.

	Donnée	État
Initialement	Non définie	Ancien
Après « ***Écrire_donnée*** »	Définie	Nouveau
Après « ***Lire_donnée*** »	Définie	Ancien

La modélisation d'un processus fonctionnel est relativement complexe dans le sens où il peut posséder de nombreuses entrées et sorties en termes de flots de données. La figure 2.49 décrit un cas général avec des entrées de type flot discret, flot continu et unité de stockage. Ainsi, nous avons en entrée un flot discret (*Donnée_d_1*), deux flots continus (*Donnnée_c_1* et *Donnée_c_2*) et deux unités de stockages (*Donnnée_s_1* et *Donnée_s_2*). En sortie, nous trouvons deux flots discrets (*Donnée_d_1* et *Donnée_d_2*), un flot continu (*Donnée_c_1*) et une unité de stockage (*Donnée_s_3*).

Le processus fonctionnel est décrit par deux places qui représentent les deux états possibles : en exécution ou traitement (place « Exécution ») et à l'arrêt (place « Oisif »).

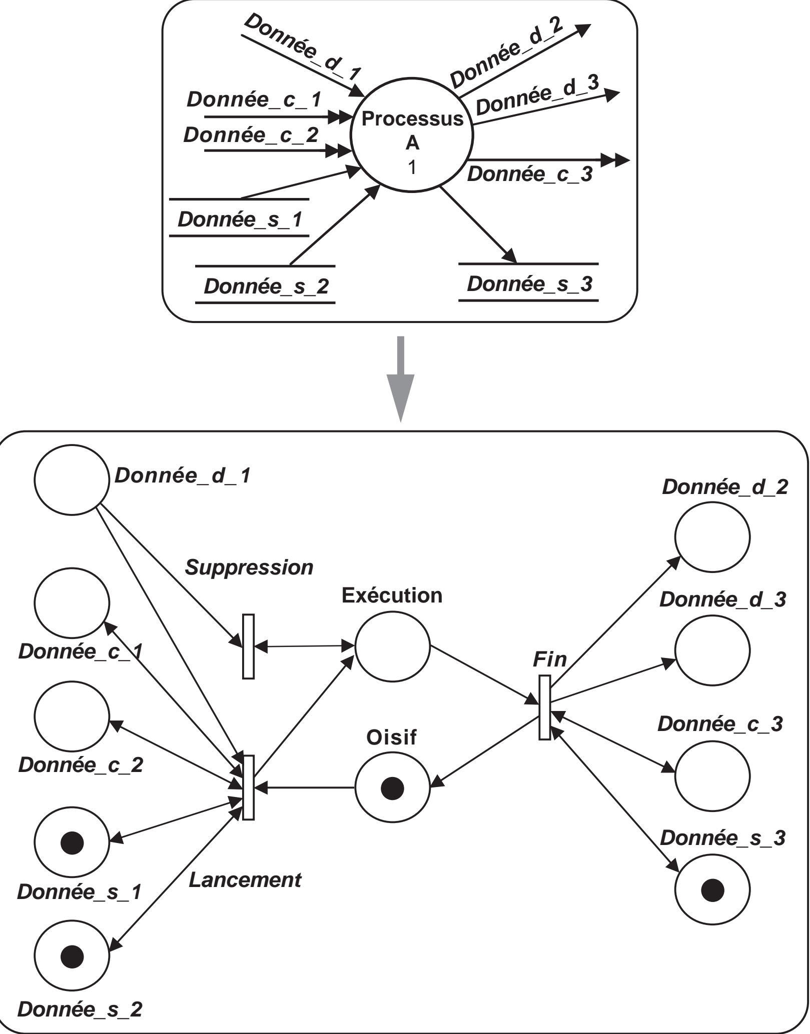


Figure 2.49 – Traduction d'un processus fonctionnel de la méthode SA-RT par un réseau de Petri.

© Dunod – La photocopie non autorisée est un délit.

Trois transitions permettent de modéliser le fonctionnement du processus : une transition « *Lancement* » au démarrage de l'exécution qui fait passer le jeton de la place « Oisif » à la place « Exécution », et une transition « *Fin* » à la fin de l'exécution qui fait passer le jeton de la place « Exécution » à la place « Oisif ».
Une transition supplémentaire est ajoutée : « *Suppression* ». En effet, une hypothèse fondamentale, concernant les flots de données discrets, est mise en exergue sur cet exemple. Le flot de données discret ne peut être consommé qu'une seule fois pendant l'exécution d'un processus fonctionnel ; par conséquent une transition supplémentaire (« *Suppression* ») est nécessaire pour éliminer les flots de données discrets arrivées pendant l'exécution.
La modélisation du processus de contrôle est très complexe et déborde largement le propos de cet ouvrage. Le lecteur intéressé pourra se reporter aux documents de référence cités dans la bibliographie en fin d'ouvrage.

■ Exemple simple

La difficulté de ce genre de modélisation par élément est le recollement des différents modèles lors de l'analyse d'une application complète. Il n'est pas possible d'ajouter ou de juxtaposer les modèles de chaque élément d'une application sans analyser l'ajustement de deux modèles l'un à la suite de l'autre. En effet, dans notre cas, les transitions ou les places situées aux limites des modèles doivent se lier avec le modèle précédent ou suivant.
Nous allons étudier cet aspect sur un exemple simple composé de deux processus fonctionnels (Processus A et Processus B) et d'une unité de stockage, appelée « *Donnée_s* » (figure 2.50). Ces éléments sont liés par des flots de données discrets. D'autre part des flots de données discrets arrivent sur les deux processus fonctionnels (*Donnée_a* et *Donnée_b*) et un flot de données discret est émis (*Donnée_c*).

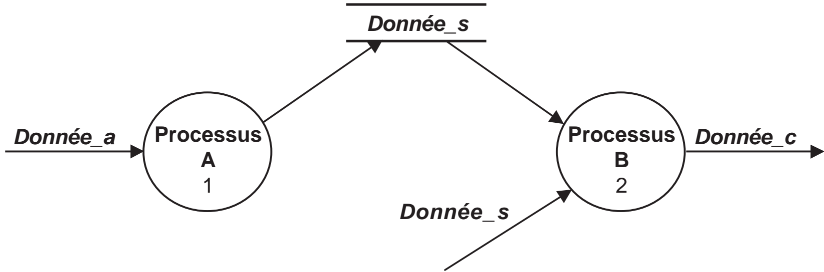


Figure 2.50 – Exemple d'un diagramme flots de données de la méthode SA-RT pour appliquer la transformation en réseaux de Petri.

La traduction de ce diagramme flot de données est représentée sur la figure 2.51. Ce réseau de Petri est composé de 11 places et de 7 transitions. Nous pouvons immédiatement remarquer que nous retrouvons les différentes places correspondant aux modèles initiaux de chaque élément. Ainsi, les deux processus fonctionnels « Processus A » et « Processus B » possèdent chacun les deux places : « A_Exécution » et « A_Oisif » pour le processus A et « B_Exécution » et « B_Oisif » pour le proces-

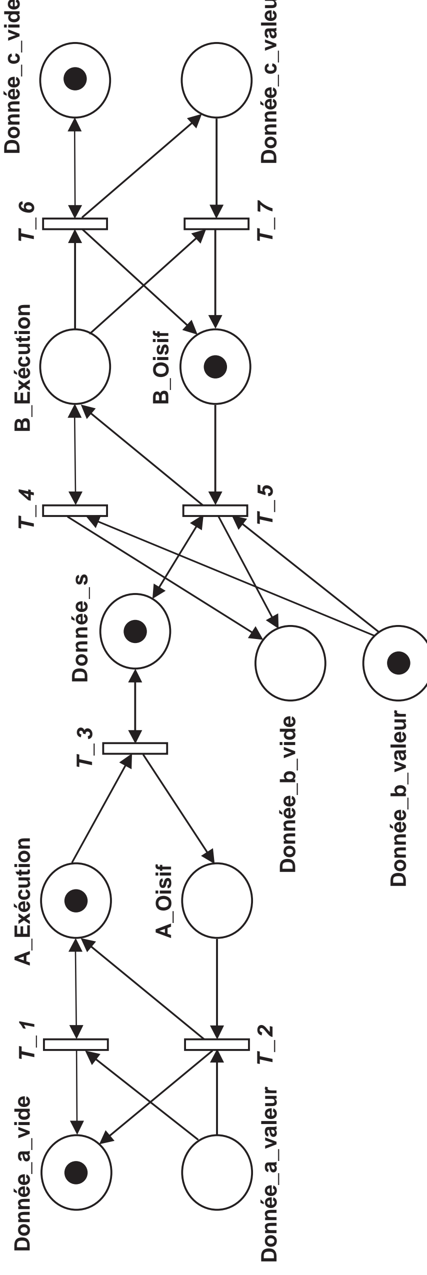


Figure 2.51 – Traduction du diagramme flots de données de la figure 2.50 en réseaux de Petri.

© Dunod – La photocopie non autorisée est un délit.

sus B. L'unité de stockage est modélisée par une seule place « Donnée_s » comme le montre le modèle initial de la figure 2.48. En se basant sur le modèle générique de la figure 2.46, les trois flots de données discrets sont représentés chacun par deux places, soit, par exemple pour le flot de données « *Donnée_a* » du modèle SA-RT : « Donnée_a_vide » et « Donnée_a_valeur ». En résumé nous pouvons noter que, lors de la traduction d'un diagramme flots de données de SA-RT en un réseau de Petri, il y aura toujours un nombre de places égal à la somme des places des modèles génériques des différents éléments du diagramme SA-RT.

En revanche, nous pouvons constater que les transitions sont partagées entre les différents éléments modélisés. Considérons le cas du raccordement du flot de données « *Donnée_a* » au processus fonctionnel A. La transition *T_2* représente à la fois la transition « *Lire_donnée* » du modèle du flot de données discret et la transition « *Exécution* » du modèle d'un processus fonctionnel. De même la transition *T_1* représente à la fois la transition « *Écrire_donnée* » du modèle du flot de données discret et la transition « *Suppression* » du modèle d'un processus fonctionnel. La dernière transition *T_3*, attachée à la modélisation du processus fonctionnel « Processus A », correspond en même temps à la transition « *Fin* » du modèle générique d'un processus fonctionnel et à la transition « *Écrire_donnée* » du modèle de l'unité de stockage « *Donnée_s* ». Nous pouvons répéter cette constatation pour l'ensemble du réseau de Petri.

Le réseau de Petri, qui est l'exact modèle du diagramme flots de données SA-RT, peut être utilisé pour vérifier certaines propriétés de la spécification comme la non occurrence de l'exécution simultanée de deux processus fonctionnels, le cheminement des données, etc. Une fonction de simulation peut être réalisée à l'aide du modèle formel réseau de Petri sous-jacent aux diagrammes SA-RT. Dans ce cadre, le réseau de Petri est analysé en traçant le graphe de marquages dans le cas de transition immédiate (modèle [0,0] de Merlin et Farber), c'est-à-dire l'évolution de la position des jetons dans le réseau. Si le modèle des transitions temporisées est pris compte l'analyse est réalisée à l'aide d'un graphe des classes. En parallèle avec cette évolution des jetons correspondant à l'activation ou non de l'action associée à la place, il est alors possible de visualiser l'activité au niveau du diagramme flots de données SA-RT. Prenons l'exemple précédent, le marquage présenté sur la figure 2.51 correspond à un moment de l'exécution où nous avons les éléments suivants :

– pas de donnée sur le flot de données discret « ***Donnée_a*** » entrant ;
– le processus fonctionnel « **Processus A** » en exécution ;
– une donnée présente dans l'unité de stockage « ***Donnée_s*** » ;
– une donnée présente sur le flot de données discret « ***Donnée_b*** » entrant ;
– le processus fonctionnel « **Processus B** » en arrêt ;
– pas de donnée sur le flot de données discret « ***Donnée_c*** » sortant.

Par conséquent, il est possible de visualiser ces différents états sur le diagramme flots de données initial, représenté sur la figure 2.50. Cette visualisation peut être animée et donc ainsi permettre au concepteur de voir le déroulement de l'exécution de diagramme flots de données SA-RT. Cette visualisation est schématisée sur la

figure 2.52 qui reprend la figure 2.50 avec une marque symbolisant l'activité d'un processus fonctionnel ou la présence d'une donnée.

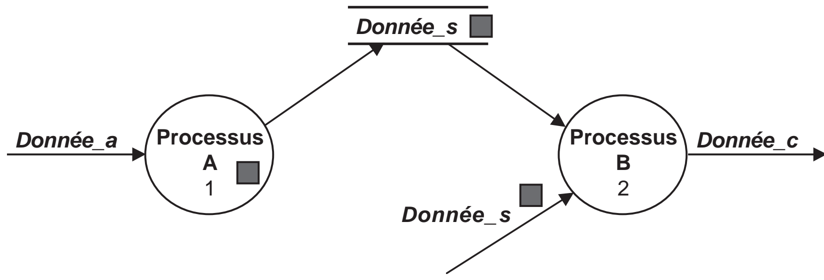


Figure 2.52 – Visualisation de l'exécution d'un diagramme flots de données en utilisant le réseau de Petri.

3 • CONCEPTION SELON LA MÉTHODE DARTS

3.1 Introduction

La méthode de conception va permettre de passer d'un modèle de spécification aux programmes codés dans un langage exécutable. Étant donné l'analyse de type fonctionnel et structuré qui a été réalisée avec la méthode SA-RT, nous pourrions être tentés de traduire directement ces modules fonctionnels (processus fonctionnels des diagrammes flots de données de SA-RT) en des entités de programmes. Cette approche serait très néfaste pour deux raisons :

– d'une part, le découpage modulaire de l'application a été fait en se préoccupant essentiellement des fonctions à réaliser et non pas de la structuration efficace en termes de réalisation et d'exécution, c'est-à-dire que le nombre de tâches de l'application ne correspond pas obligatoirement au nombre de processus fonctionnels des diagrammes SA-RT ;

– d'autre part, les relations entre les processus fonctionnels des diagrammes flots de données sont décrites de manière très abstraite (zone mémoire, passage de données) sans se préoccuper de leurs implémentations possibles avec ou sans synchronisation.

La question qui se pose concerne le choix de cette méthode. En effet, il serait tout à fait envisageable, comme nous l'avons vu dans les exemples industriels du paragraphe 1.3.4, d'associer une méthode d'analyse de type « flot de données » à une méthode de conception de type « orienté objets ». En revanche, le choix de l'une ou l'autre voie amène quasi obligatoirement un choix de langage. Ainsi, la figure 3.1 représente les choix possibles selon les deux groupes de méthodes de conception. La méthode DARTS (*Design Approach for Real-Time Systems* – H. Gomaa, 1984) trouve naturellement sa place entre la méthode d'analyse SA-RT et une implémentation avec un langage de programmation de haut niveau exécuté dans un environnement de type noyau temps réel, ou avec un langage multitâche (Ada) ou un langage flots de données (LabVIEW).

3.1.1 Les besoins d'une méthode de conception

La méthode de conception doit offrir tous les moyens d'expression à l'utilisateur pour décrire son application multitâche. Ainsi, la conception doit pouvoir exprimer

© Dunod – La photocopie non autorisée est un délit.

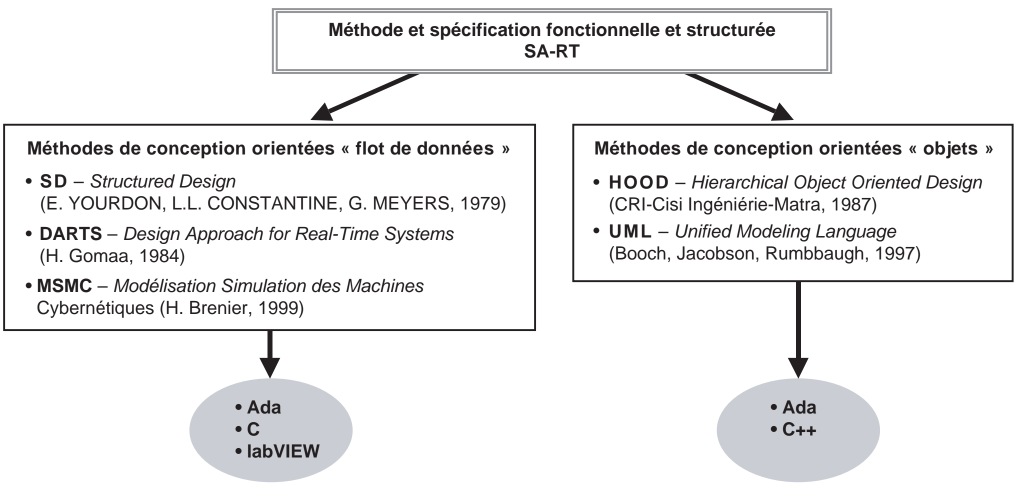


Figure 3.1 – Les choix d'une méthode de conception et du langage d'implémentation suite à une analyse de type SA-RT.

l'architecture donnée sur la figure 3.2, c'est-à-dire les différentes tâches, les relations entre les tâches et la possibilité d'accès à des ressources critiques.
Ainsi, nous pouvons donc décliner les différentes caractéristiques à représenter avec la méthode DARTS :

– les tâches : type d'activation, paramètres ou données en entrées ou en sorties ;
– les relations entre les tâches :
  • synchronisation de type asynchrone ou synchrone (rendez-vous),
  • communication avec des données qui transitent entre deux tâches ;
– le partage de ressources critiques.

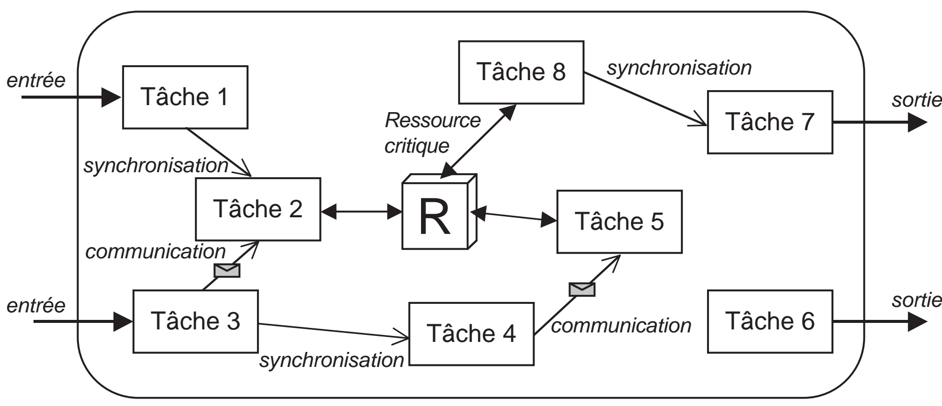


Figure 3.2 – Architecture multitâche à décrire avec la méthode de conception DARTS.

3.1.2 Modélisation des tâches

Nous pouvons définir deux grandes catégories de tâches : les tâches matérielles et les tâches logicielles.

– Les **tâches matérielles** (ou tâches immédiates) sont associées à des événements matériels externes au programme. L'activation de ces tâches se fait donc par une interruption provenant soit du procédé externe, soit du système de contrôle lui-même (horloge temps réel, temporisateur « chien de garde »…). Dans cette catégorie de tâches, nous trouvons en particulier toutes les tâches d'acquisitions qui sont de type régulier, c'est-à-dire une acquisition par scrutation (*polling task*) du capteur. Nous avons aussi les tâches d'acquisition qui sont déclenchées par une interruption externe comme une alerte générée par un dépassement de paramètre.

– Les **tâches logicielles** (ou tâches différées) sont celles dont l'activation est demandée par une autre tâche, tâche appelante, qui peut être de type matériel ou logiciel. Dans le cas des tâches logicielles, les caractéristiques temporelles de la tâche appelée sont déterminées d'une part à partir des attributs temporels de la tâche appelante et d'autre part en fonction des relations temporelles spécifiques entre la tâche appelante et la tâche appelée. Ces tâches doivent obligatoirement être liées avec une autre tâche.

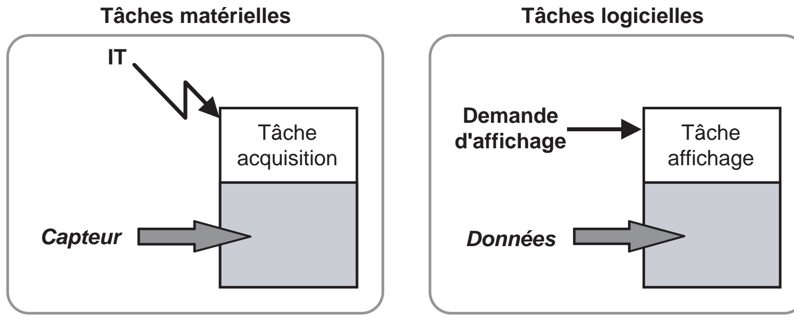


Figure 3.3 – Exemple des deux types de tâches matérielles et logicielles.

En ce qui concerne les tâches matérielles, les spécifications de l'application vont en particulier décrire les caractéristiques temporelles des signaux d'interruptions internes ou externes associés à ces tâches matérielles. Les principaux types de signaux sont (figure 3.4) :

– les **signaux périodiques** qui peuvent être caractérisés par une première date d'occurrence et une période. Ces signaux (référencés 1 sur la figure 3.4) proviennent en général d'une horloge interne « HTR – Horloge Temps Réel ». ;

– les **signaux apériodiques** qui se produisent de façon répétitive mais sans période fixée (référencés 2 sur la figure 3.4). Pour déterminer les caractéristiques des tâches associées à ces signaux apériodiques, il est nécessaire de posséder des informations temporelles les concernant, par exemple la durée minimale entre deux occurrences ;

© Dunod – La photocopie non autorisée est un délit.

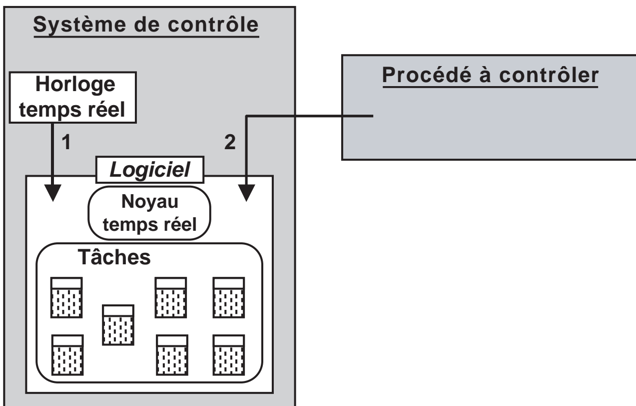


Figure 3.4 – Les différents types d'activation des tâches matérielles.
Interactions du logiciel temps réel avec des événements externes : (1) interruptions internes provenant d'une horloge temps réel, (2) interruptions externes provenant du procédé à contrôler.

– les **signaux sporadiques** qui se produisent une seule fois, comme les signaux d'alarme provenant du procédé.

3.1.3 Relations entre les tâches

Le point le plus complexe est la traduction des différentes relations entre les tâches. En effet, cette relation traduit « un lien de dépendance entre les tâches », c'est-à-dire qu'une ou plusieurs tâches ne commencent leur exécution uniquement que lorsqu'une autre tâche s'est exécutée en partie ou en totalité. Cette synchronisation ou communication peut être unilatérale ou asynchrone (seule la tâche en attente est bloquée), ou bilatérale ou synchrone (les deux tâches ont un rendez-vous, exemple du langage Ada). La figure 3.5 montre cette différence de relations de synchronisation entre deux tâches. Dans le premier cas I de la relation asynchrone, la tâche 1 bloque la tâche 2 qui doit attendre l'exécution d'une partie de la tâche 1 en un certain point de son code ; en revanche, dans le second cas, les deux tâches doivent s'attendre mutuellement.

Attachée à cette synchronisation de deux ou plusieurs tâches, un transfert de données peut être associé, nous parlons alors de communication. Une communication est caractérisée par la taille de cette zone d'échange de données (une ou plusieurs données) et le mode de gestion de cette zone de données (FIFO – *First In Fisrt Out*, à priorité). Le critère principal, lié à cet échange de données, est l'aspect bloquant ou non de l'interaction entre les tâches. Si la production et la consommation de données sont synchronisées, c'est-à-dire que, si la zone de données est vide, la tâche consommatrice ne peut s'exécuter et réciproquement, si la zone de données est pleine, la tâche productrice ne peut s'exécuter. Nous avons dans ce cas une communication mettant en jeu une vraie synchronisation entre les tâches. Dans le cas contraire, la zone de données est dite « à écrasement » et nous avons un blocage uniquement en lecture des données.

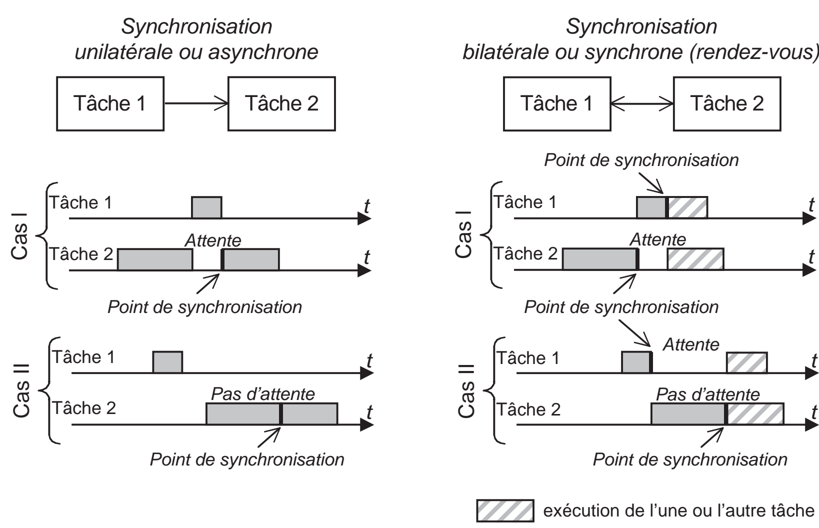


Figure 3.5 – Relations de synchronisation entre les tâches.

3.2 Présentation de la méthode DARTS

La méthode de conception DARTS constitue le lien entre la méthode d'analyse SA-RT et l'implémentation de l'application. Nous allons étudier la méthode DARTS dans sa forme la plus simple, c'est-à-dire s'appliquant à des applications de contrôle-commande générales. Il est intéressant de noter que des versions enrichies de cette méthode de conception ont été créées afin de répondre à des besoins spécifiques. Nous trouvons ainsi :

– méthode DARTS adaptée à une implémentation avec le langage Ada : **ADARTS** : *Ada Based Design Approach for Real-Time Systems* (GOMAA, 1987) ;
– méthode DARTS adaptée à une application distribuée : **CODARTS** : *Concurrent Design Approach for Real-Time Systems* (GOMAA, 1987).

Pour s'harmoniser de façon complète avec la méthode d'analyse SA-RT, la méthode de conception DARTS est de type flots de données. Ainsi, les diagrammes flots de données de la méthode SA-RT (diagrammes préliminaires ou diagrammes de décomposition) sont traduits en diagramme flots de données DARTS représentant l'architecture multitâche de l'application.

D'autre part, la méthode DARTS que nous présentons dans cet ouvrage permet de décrire de nombreuses applications de type contrôle-commande. Mais afin d'obtenir des applications dont le test ou la vérification peuvent être conduits de façon fiable, nous avons volontairement limité la possibilité d'expression en se donnant des règles de « bonne conception ». Ces règles correspondent en partie à un profil de programmation adopté dans le cadre de la programmation en langage Ada d'applications à haut niveau de sécurité. La présentation détaillée de ce profil de programmation,

© Dunod – La photocopie non autorisée est un délit.

appelé *Ravenscar profile* (1997), est effectuée dans le chapitre 5. Dans ce contexte, nous pouvons ainsi définir les règles suivantes :

– Nombre fixé de tâches au démarrage de l'application.
– Un seul événement de déclenchement par tâche : signal temporel, synchronisation avec une autre tâche.
– Interaction entre les tâches qui doit être réalisée uniquement par données partagées gérées de manière atomique.
– Possibilité d'analyse fonctionnelle des tâches de façon individuelle.

Il est à noter que d'autres règles concernant plus la partie implémentation et l'aspect ordonnancement sont étudiées dans les chapitres concernés.

3.2.1 Syntaxe graphique de la méthode DARTS

Nous allons présenter dans cette section les différents éléments de la bibliothèque graphique de la méthode DARTS correspondant à la phase de conception.

■ Modélisation des tâches et modules

En premier lieu, nous trouvons la **tâche** qui représente l'entité de base de l'architecture multitâche. Nous pouvons avoir un ou plusieurs flots de données en entrées et un ou plusieurs flots de données en sortie (figure 3.6). Les tâches sont modélisées par un parallélogramme qui comporte une étiquette ou label explicite formé de :

Étiquette_Tâche = verbe (+ un ou plusieurs compléments d'objets)

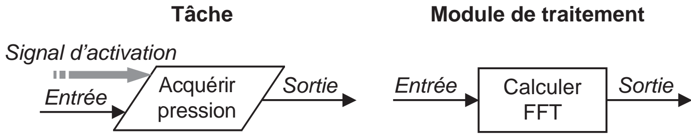


Figure 3.6 – Représentations des tâches et des modules.

Cette étiquette peut correspondre à celle donnée dans le cadre d'un diagramme flots de données SA-RT s'il y a, par exemple, une correspondance entre une tâche et un processus fonctionnel comme nous allons l'étudier dans la suite de ce chapitre. Nous avons aussi un signal d'activation qui a une provenance et un type différents selon que nous modélisons une tâche matérielle (activation de type événement externe au programme) ou une tâche logicielle (activation de type événement interne au programme : synchronisation ou communication). Ce signal d'activation doit obligatoirement exister et doit être unique pour répondre au profil de conception fixé.

Nous avons aussi le **module de traitement** qui correspond à des programmes spécifiques appelés par les tâches pour effectuer des calculs particuliers. Nous pouvons avoir un ou plusieurs flots de données en entrées et un ou plusieurs flots de données en sortie (figure 3.6). Les modules de traitement sont modélisés par un rectangle qui comporte une étiquette ou label explicite formé de :

Étiquette_Module = verbe (+ un ou plusieurs compléments d'objets)

Ces modules ne peuvent pas émettre ou recevoir des éléments de type multitâche : synchronisation ou communication. Ils peuvent être considérés comme réentrants, c'est-à-dire partagés par plusieurs tâches sans mettre en œuvre une gestion spéciale de l'accès. Ces modules concernent généralement des calculs de type traitement du signal, traitement d'images, loi de régulation de l'automatique… Cette séparation des unités de traitement permet de ne pas alourdir les tâches en termes de code et d'y conserver uniquement l'aspect dynamique et comportemental.

■ Modélisation des synchronisations et des communications

En ce qui concerne les **synchronisations**, nous avons les modèles correspondant aux deux synchronisations précédemment citées, c'est-à-dire la synchronisation de type asynchrone et la synchronisation de type synchrone, c'est-à-dire le rendez-vous (figure 3.7). Le modèle « synchronisation de type asynchrone » est généralement le plus utilisé car il permet une validation plus aisée de l'application. Ces modèles représentent le cheminement des dépendances entre les tâches en termes d'exécution. Les synchronisations sont donc représentées par un symbole orienté avec une étiquette ou label explicite formé de :

Étiquette_Synchronisation = nom (+ qualifiant)

Cette synchronisation peut relier plusieurs tâches en amont d'une tâche synchronisée en considérant une opération « OU » entre toutes les synchronisations en provenance des tâches en amont. Rappelons qu'il n'est pas souhaitable d'avoir deux synchronisations à l'entrée d'une tâche ; cela correspondrait alors à deux activations de la tâche, ce qui est contraire au profil de programmation énoncé.

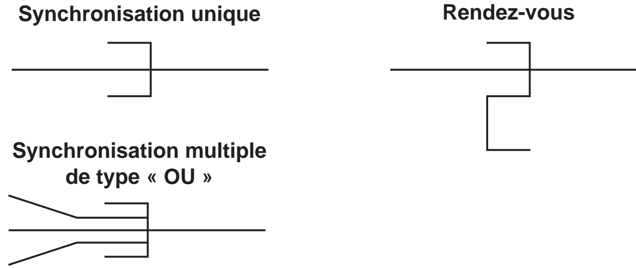


Figure 3.7 – Représentations des synchronisations entre les tâches.

Pour les **communications**, nous avons plusieurs modèles correspondant d'une part à la taille de la zone de communication et d'autre part à la gestion de cette zone de stockage des données (figure 3.8). D'une manière générale, ces modèles « boîtes aux lettres – BAL » représentent le cheminement des données entre les tâches avec un aspect relation de précédence pour certaines communications. Les communications sont donc représentées par un symbole orienté avec une étiquette ou label explicite formé de :

© Dunod – La photocopie non autorisée est un délit.

Étiquette_Communication = nom (+ qualifiant)

Nous pouvons ainsi distinguer deux types de gestion de la zone de stockage des données d'un point de vue dépendance (relation synchrone ou non) :

– les boîtes aux lettres bloquantes qui se déclinent en trois modèles :
  - boîtes aux lettres bloquantes pouvant contenir *n* messages gérés selon une file FIFO,
  - boîtes aux lettres bloquantes ne pouvant contenir qu'un seul message,
  - boîtes aux lettres bloquantes pouvant contenir *n* messages gérés selon une file FIFO, classés par priorités correspondantes généralement à celle de la tâche émettrice de la donnée ;

– les boîtes aux lettres non bloquantes en écriture ou BAL à écrasement qui se déclinent en deux modèles :
  - boîtes aux lettres non bloquantes en écriture pouvant contenir n messages gérés selon une file FIFO,
  - boîtes aux lettres non bloquantes en écriture ne pouvant contenir qu'un seul message.

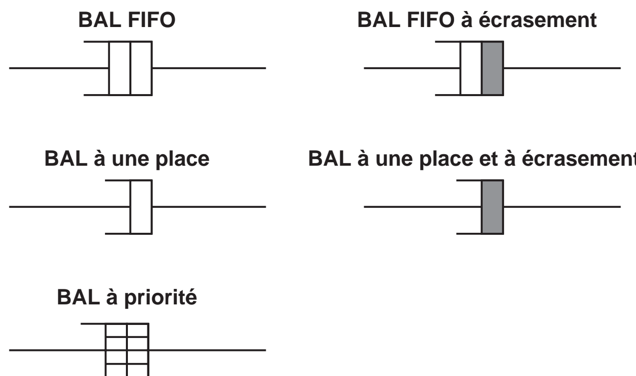


Figure 3.8 – Représentations des communications entre les tâches.

Les communications, dites à écrasement, fonctionnent en éliminant la donnée la plus vieille dans le cas d'une file FIFO ou la donnée stockée dans le cas de la boîte aux lettres à une place. Il est important de noter que ces communications sont non bloquantes en écriture uniquement et représentent donc une synchronisation et une activation d'une tâche comme les autres types de communications. Comme pour les synchronisations, ces communications peuvent posséder plusieurs tâches en amont en considérant une opération « OU » entre toutes les communications en provenance des tâches en amont (figure 3.9). En revanche, il est déconseillé d'avoir plusieurs tâches en aval car cela nuit au déterminisme du programme.

BAL FIFO multiple de type « OU »

BAL FIFO à écrasement multiple de type « OU »

Figure 3.9 – Représentations des communications entre les tâches avec entrées multiples de type « OU ».

■ Modélisation de l'activation des tâches

Comme nous l'avons présenté dans le paragraphe 3.1.2, il existe deux types de tâches : tâches matérielles et tâches logicielles. Dans les deux cas, l'activation est très différente. Ainsi, pour les tâches matérielles qui sont déclenchées par des événements ou signaux externes, nous distinguons trois types de signaux :

– Signal « Horloge temps réel – HTR ». Ce signal, qui provient d'une horloge matérielle interne à l'ordinateur, correspond à un signal rigoureusement périodique.
– Signal « Interruption – IT ». Ce signal qui provient du procédé externe doit toujours être considéré comme apériodique du fait de l'asynchronisme du monde extérieur par rapport au cadencement de l'ordinateur.
– Signal « Chien de garde – CG ». Ce signal provient d'une horloge interne utilisée comme un réveil. Son utilisation et son fonctionnement sont décrits de façon détaillée dans la suite de cet ouvrage. En termes de signal, il est identique à l'horloge temps réel (signal interne) ; mais il se produit de façon apériodique.

Les activations sont donc représentées par un symbole orienté (ligne brisée) avec une étiquette ou label explicite formé de (figure 3.10) :

Étiquette_Activation_HTR = HTR (durée en ms)

Étiquette_Activation_IT = IT (+ nom interruption)

Étiquette_Activation_CG = CG (+ nom chien de garde)

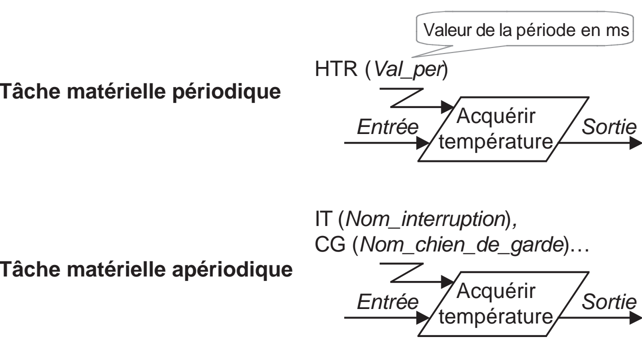


Figure 3.10 – Représentations de l'activation des tâches matérielles.

© Dunod – La photocopie non autorisée est un délit.

Comme nous l'avons déjà explicité, les tâches logicielles sont déclenchées par d'autres tâches (matérielles ou logicielles) avec les mécanismes de synchronisation ou de communication de type bloquant. Ainsi, le signal d'activation de ces tâches peut être l'action des éléments suivants (figure 3.11) :

– boîtes aux lettres à écrasement ou non gérées selon une file FIFO ou FIFO à priorité ;
– boîtes aux lettres à écrasement ou non à une seule place ;
– boîtes aux lettres à *n* places gérées selon la priorité.

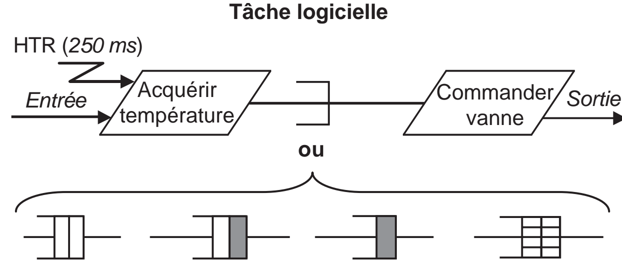


Figure 3.11 – Représentations de l'activation des tâches logicielles.

Ainsi, nous pouvons noter qu'une tâche logicielle ne peut pas être activée par un élément de synchronisation et en même temps être connectée à une autre tâche en amont par une boîte aux lettres à écrasement, c'est-à-dire bloquante en lecture (figure 3.12). Ainsi, la tâche « Commander vanne » doit être synchronisée soit par la tâche matérielle « Attendre mesures » qui est activée par une interruption, soit par l'autre tâche matérielle « Acquérir niveau », dite à scrutation, qui est périodique et activée par l'horloge temps réel dont la période est de 350 ms.

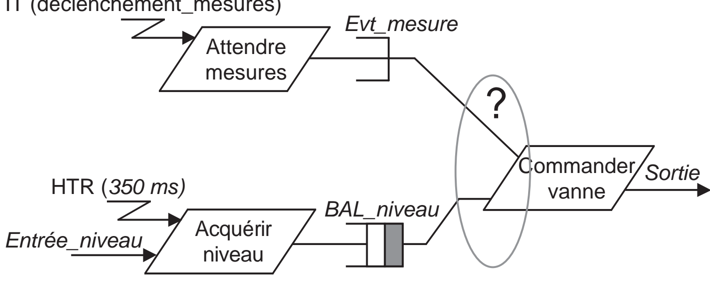


Figure 3.12 – Exemple de dysfonctionnement de synchronisation d'une tâche logicielle par deux tâches matérielles.

■ Modélisation des stockages de données

Le dernier élément à modéliser est le **module de données** qui permet une protection des accès à une unité de gestion de données en **exclusion mutuelle** par deux ou plusieurs tâches. Les modules de données sont représentés par un rectangle associé à des entrées permettant de réaliser une action sur les données : LIRE, ÉCRIRE, etc. Ce symbole du module de données est représenté avec une étiquette ou label explicite formé de (figure 3.13) :

Étiquette_module_données = nom (+ qualifiant)

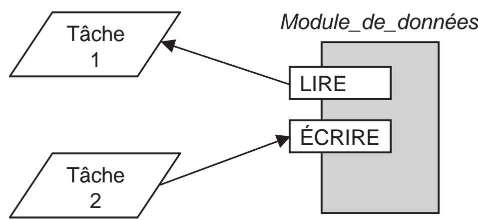


Figure 3.13 – Représentations d'un module de données partagé par plusieurs tâches.

Si l'action met en jeu un transfert de donnée entre la tâche et le module de données (LIRE, ÉCRIRE, etc.), la flèche est orientée dans le sens de ce transfert de données. Si ce n'est pas le cas (INITIALISER, TRIER, etc.), la flèche est orientée de la tâche vers le module de données afin de montrer la demande qu'effectue la tâche sur ce module de données (figure 3.14). Il est important de noter que plusieurs tâches peuvent accéder à un module de données puisque ce module de données est géré en exclusion mutuelle.

Il est important de noter que cet élément « module de données » ne peut en aucun cas être utilisé comme un élément de synchronisation. Les deux tâches 1 et 2 de la figure 3.13 sont totalement asynchrones et doivent posséder leur propre mécanisme d'activation. Nous pourrions définir le module de données comme une boîte aux lettres à écrasement en écriture et à lecture non destructive avec un message par défaut à l'initialisation.

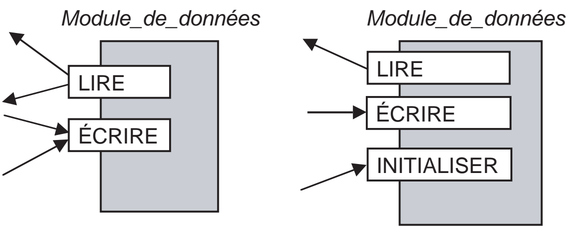


Figure 3.14 – Représentations de différents modules de données partagés par plusieurs tâches.

© Dunod – La photocopie non autorisée est un délit.

3.2.2 Mise en œuvre de la méthode DARTS

Cette section décrit le passage de la spécification SA-RT à la conception DARTS. Après quelques règles de base permettant de donner un guide méthodologique à ce passage SA-RT/DARTS, des exemples simples sont décrits et nous finissons par l'exemple du « système de freinage automobile » déjà spécifié.

■ Règles générales

Pour cette méthode de conception non formelle, nous allons définir des règles générales permettant de passer d'un diagramme flots de données de la méthode SA-RT à un diagramme multitâche DARTS. Pour cela, nous devons réaliser la traduction selon les quatre phases suivantes :

– Phase 1 : **Création des tâches**. Cela correspond à la traduction des processus fonctionnels ou de contrôle en tâches.

– Phase 2 : **Détermination du typage et de l'activation des tâches**. Les tâches sont déclarées matérielles ou logicielles. Dans le cas des tâches matérielles, le signal d'activation doit être défini précisément (HTR, IT, CG). Pour les autres tâches logicielles, il est nécessaire d'identifier les synchronisations permettant de les activer. Celles-ci sont souvent déjà déclarées comme des événements traités par le processus de contrôle.

– Phase 3 : **Mise en place des synchronisations et des transferts de données**. Les relations par communications sont traduites par des boîtes aux lettres ou par des modules de données.

– Phase 4 : **Regroupement ou rassemblement des tâches**. Afin d'améliorer et de simplifier la première conception réalisée de façon semi-automatique en suivant les trois règles de base que nous allons décrire, le diagramme multitâche est de nouveau analysé, et les tâches sont regroupées selon un ensemble de critères qui sont exposés dans la suite.

Pour la première phase de création des tâches, nous pouvons définir trois règles de base qui servent de guide à cette traduction semi-automatisée :

– Règle 1.1 : Une tâche du modèle DARTS est créée pour chaque processus fonctionnel du diagramme SA-RT.

– Règle 1.2 : Une tâche supplémentaire du modèle DARTS est associée au processus de contrôle du diagramme SA-RT si le diagramme état/transition, associé à ce processus de contrôle, est complexe, c'est-à-dire qu'il possède au moins une structure conditionnelle.

En ce qui concerne le typage et l'activation des tâches, nous pouvons énoncer les règles suivantes :

– Règle 2.1 : Une tâche en entrée (acquisition de données) est obligatoirement de type matériel déclenchée soit par l'horloge temps réel (ex. : tâche de scrutation d'un paramètre physique), soit par une interruption provenant du procédé (ex. : tâche d'alarme).

– Règle 2.2 : Les autres tâches sont déclarées soit logicielles (activation par synchronisation ou communication avec les tâches matérielles), soit matérielles déclenchée par un événement interne (horloge temps réel, chien de garde…).
– Règle 2.3 : Les événements importants du diagramme flots de données de SA-RT peuvent être traduits par des synchronisations qui sont utilisées pour activer des tâches logicielles.

Enfin, les communications entre les tâches de DARTS sont établies en se basant sur les deux règles suivantes :

– Règle 3.1 : Les communications directes entre les processus fonctionnels (flot de données d'un processus fonctionnel à un autre) sont traduites préférentiellement par des boîtes aux lettres.
– Règle 3.2 : Les communications par une zone de stockage entre les processus fonctionnels sont traduites préférentiellement par des modules de données, en particulier si cette zone de stockage se trouve partagée par plus de deux tâches.

Les règles (ou critères de regroupement des tâches) concernant la dernière phase sont étudiées dans la suite de cette section sur un exemple précis.
Cet ensemble de règles ne constitue pas une traduction automatique et formelle de la spécification SA-RT en des diagrammes multitâches DARTS. En effet, la règle, qui concerne les regroupements ou les scissions éventuelles des tâches créées en se basant sur les deux premières règles 1.1 et 1.2, laisse un libre choix de la configuration multitâche. De même les deux règles 3.1 et 3.2, concernant les communications entre les tâches, ne définissent en rien la gestion de ces boîtes aux lettres (FIFO ou priorité, à écrasement ou non, à une ou plusieurs places). Ainsi, en partant d'une même spécification SA-RT (diagramme flots de données), cette phase de conception, basée sur DARTS, donne naturellement plusieurs solutions selon le concepteur qui effectue la traduction.

■ Exemples simples de traduction de SA-RT vers DARTS

Nous allons appliquer quelques règles énoncées précédemment en les illustrant d'exemples simples. Soit la figure 3.15, considérons le processus fonctionnel correspondant à l'acquisition de données provenant de deux capteurs (thermocouple et capteur rotatif). D'après la règle 2.1, ce processus fonctionnel se transforme en une tâche matérielle d'entrée qui va être activée périodiquement (période de 2 ms).

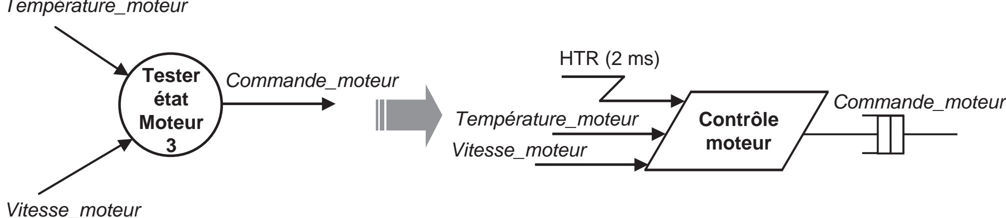


Figure 3.15 – Traduction d'un processus fonctionnel d'acquisition de données en une tâche périodique.

© Dunod – La photocopie non autorisée est un délit.

Nous pouvons noter les noms des différents éléments qui sont conservés autant que possible. Cette tâche matérielle est donc une tâche d'acquisition de données dite à scrutation. La transmission de la donnée « *Commande_moteur* » est traduite par une boîte aux lettres à *n* places conformément à la préconisation de la règle 3.1.

Dans l'exemple de la figure 3.16, nous avons aussi un processus fonctionnel correspondant à l'acquisition de données provenant de deux capteurs (thermocouple et capteur rotatif). D'après la règle 2.1, ce processus fonctionnel se transforme en une tâche matérielle d'entrée qui va être activée par une interruption (*Dépassement_température*) qui est issue de la mesure du capteur de température. La tâche ainsi créée est donc apériodique et s'active au rythme de l'interruption. Le processus fonctionnel émet un événement « *Afficher_alarme* » vers le processus de contrôle supposé. Dans ce cas, en appliquant la règle 3.2, nous traduisons cet événement par une synchronisation nommée « *Alarme* ».

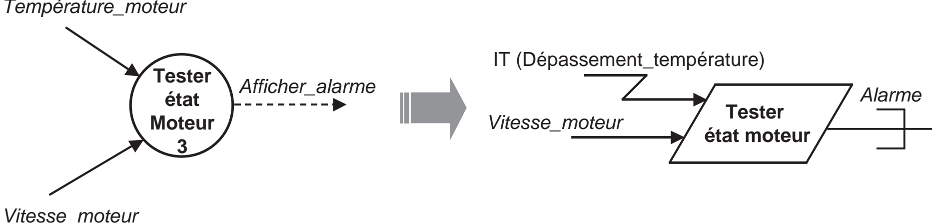


Figure 3.16 – Traduction d'un processus fonctionnel d'acquisition de données en une tâche apériodique.

De la même manière que pour la création des tâches à partir des diagrammes flots de données de SA-RT, la traduction des transferts de données peut suivre les règles affichées. Ainsi, la figure 3.17, qui présente un flot de données direct entre deux processus fonctionnels, montre la traduction par un élément de communication de type boîte aux lettres conformément à la règle 3.1. Nous pouvons remarquer que la première tâche est de type matériel, déclenchée par une horloge temps réel, et la deuxième tâche de type logiciel est activée par la synchronisation de la communication par boîte aux lettres.

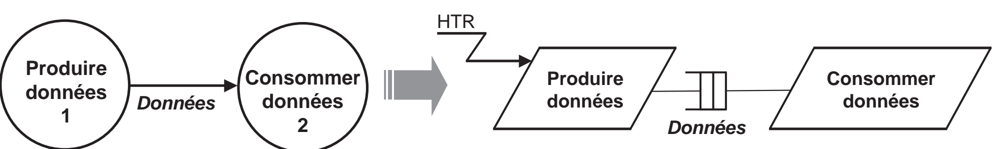


Figure 3.17 – Traduction d'un flot de données direct entre deux processus fonctionnels par une boîte aux lettres.

Le deuxième cas traité, illustré sur la figure 3.18, est celui du transfert de données par une unité de stockage au niveau du diagramme flots de données SA-RT. Selon

la règle 3.2, il est naturel de traduire ce flot de données par un module de données ; toutefois ce transfert étant limité à deux processus fonctionnels, aurait pu aussi être traduit par un élément de type boîte aux lettres comme dans le cas de la figure 3.17. En revanche, la première traduction par un module de données, qui n'est pas une synchronisation, oblige à synchroniser aussi la deuxième tâche par une autre horloge temps réel par exemple. Les tâches sont alors indépendantes au niveau de leur cadence d'exécution.

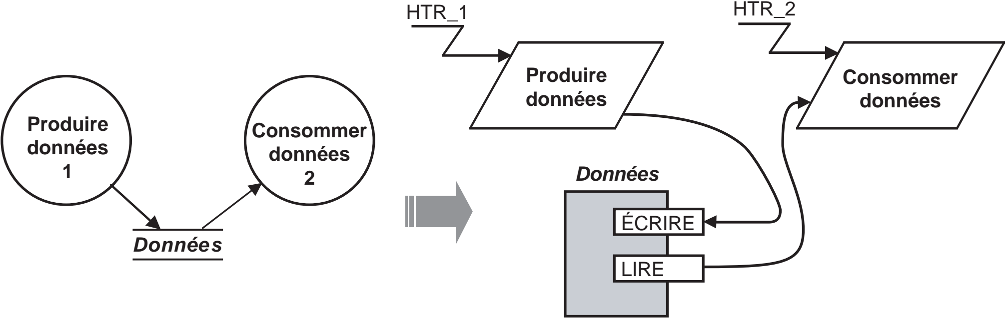


Figure 3.18 – Traduction d'une unité de stockage entre deux processus fonctionnels par un module de données.

■ Une première conception avec DARTS

Nous allons mettre en œuvre cette méthodologie DARTS pour l'exemple décrit jusqu'à présent, c'est-à-dire le « système de freinage automobile ». Pour cela nous allons considérer le dernier diagramme préliminaire défini, celui de la figure 2.20, modifié avec la partie de diagramme représenté sur la figure 2.25. Ce diagramme préliminaire, présenté sur la figure 3.19, sert de base pour la traduction en diagramme multitâche DARTS.

La première phase à mettre en œuvre est la création des tâches. Ainsi, selon la règle 1.1, nous allons créer cinq tâches correspondant aux cinq processus fonctionnels du diagramme préliminaire. Le processus de contrôle de ce diagramme préliminaire a un fonctionnement décrit par le diagramme état/transition relativement complexe représenté sur la figure 2.24. Par conséquent, d'après la règle 1.2, nous devons dans une première étape traduire ce processus de contrôle par une tâche. Ainsi, nous obtenons un diagramme multitâche DARTS comportant six tâches ayant les mêmes noms que les processus fonctionnels SA-RT du diagramme préliminaire de la figure 3.19. Ce diagramme multitâche DARTS, représenté sur la figure 3.20, comporte de plus pour chacune des tâches les entrées/sorties des processus fonctionnels correspondant avec des noms identiques. La nomination correspondante entre les diagrammes SA-RT et DARTS n'est pas obligatoire, mais elle joue un rôle important sur la traçabilité lors du passage de la spécification à la conception.

La deuxième phase concerne la détermination du type des tâches et de leur activation. En suivant la règle 2.1, les tâches d'acquisition « Acquérir demande freinage »,

© Dunod – La photocopie non autorisée est un délit.

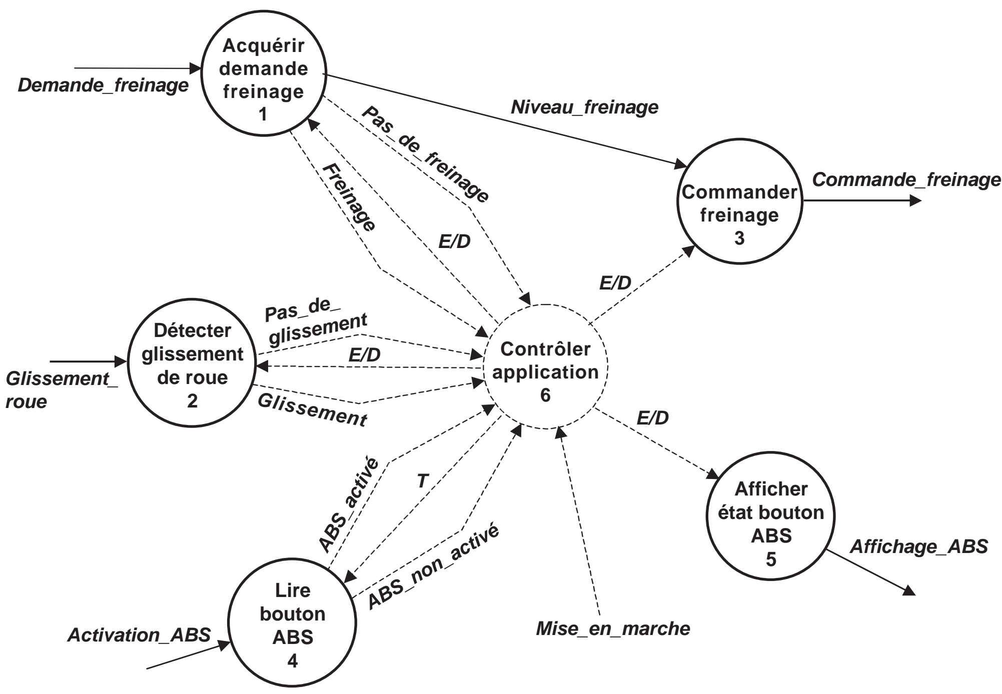


Figure 3.19 – Diagramme préliminaire de l'application « système de freinage automobile » servant de base à la conception DARTS.

« Détecter glissement » et « Lire boutons ABS » sont tout naturellement des tâches matérielles de type scrutation (tâche périodique déclenchée par l'horloge temps réel). Nous allons définir les périodes respectives de ces tâches : 100, 150 et 1 000 ms.

Les trois autres tâches peuvent être des tâches logicielles, si elles sont activées par l'une des tâches précédentes, ou éventuellement une tâche matérielle. Nous pouvons identifier sur le diagramme préliminaire de la figure 3.19 deux événements « *Freinage* » et « *ABS_activé* » qui vont agir respectivement sur les tâches « Contrôler application » et « Afficher état bouton ABS ». Ces deux événements peuvent donc être traduits par des synchronisations (figure 3.20).

Nous arrivons maintenant à la troisième et dernière phase de traduction quasi-automatique du diagramme SA-RT en diagramme DARTS, sachant que la phase suivante correspond à la partie de la conception par reprise et amélioration du diagramme DARTS obtenu après les trois premières phases. La mise en place des communications se fait en suivant les règles 3.1 et 3.2. Nous avons donc une traduction du flot de données direct entre les deux processus fonctionnels « Acquérir demande freinage » et « Commander freinage » par une boîte aux lettres. Le type de boîte aux lettres choisi est une BAL FIFO à écrasement. En effet, comme nous allons le voir, la tâche « Commander freinage » peut s'exécuter moins souvent que la tâche de scrutation du capteur « Acquérir demande freinage » du fait de l'arrêt du freinage lors du glissement et du système ABS en fonctionnement. Toutefois il est nécessaire de synchroniser cette tâche « Commander freinage » avec la tâche « Contrôler

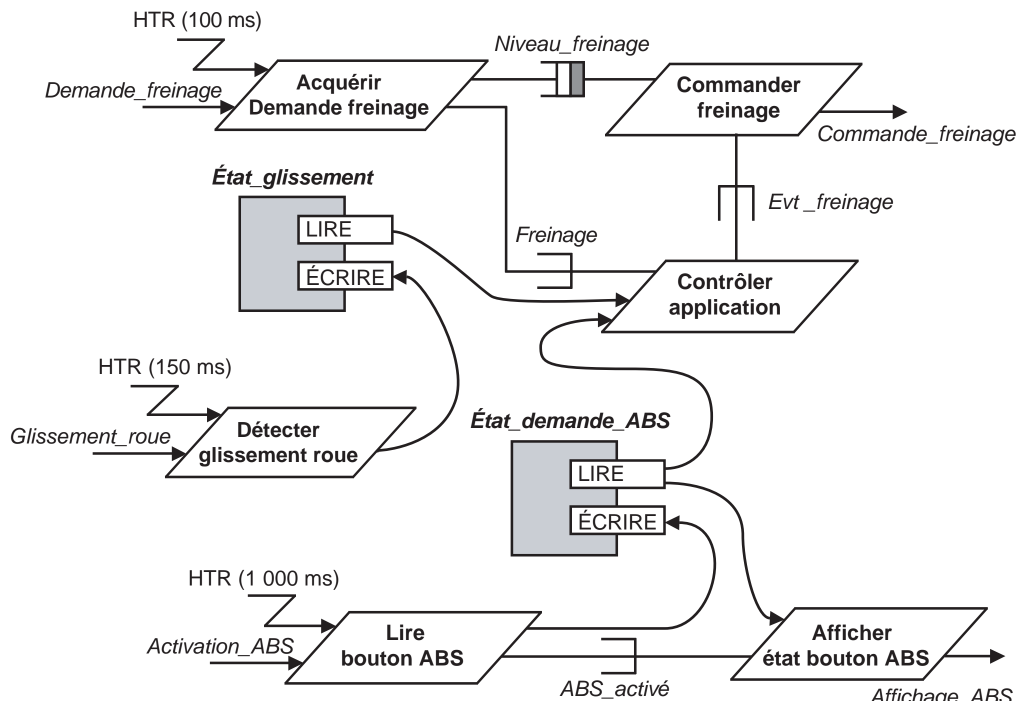


Figure 3.20 – Première conception du diagramme multitâche DARTS à partir du diagramme préliminaire de la figure 3.19.

freinage » ; cela est réalisé avec la synchronisation « *Evt_freinage* ». D'autre part, il est nécessaire de passer les informations sur l'état du bouton ABS et sur le glissement de la roue. Comme les tâches d'acquisitions de ces deux paramètres sont activées à des rythmes différents de la tâche « Contrôler freinage » qui travaille à la période de la tâche « Acquérir demande freinage », nous sommes contraints de découpler ces passages de paramètres à l'aide de deux modules de données : « *Etat_glissement* » et « *Etat_demande_ABS* ». Enfin, la tâche « Afficher état bouton ABS » peut puiser son information dans le module de données « *Etat_demande_ABS* ».

Avant de continuer la conception à partir de ce premier diagramme multitâche ainsi élaboré, nous allons nous intéresser à ce dernier cas de relation entre deux tâches qui doivent se synchroniser et s'échanger une donnée partagée par d'autres tâches. Cette liaison peut se faire de trois manières différentes. Les conditions initiales sont une première tâche dont l'activation est définie (dans cet exemple, une horloge temps réel), une deuxième tâche de type logiciel qu'il est nécessaire de synchroniser et un module de données créé afin de distribuer une donnée vers d'autres tâches. Ainsi, les différents cas, présentés sur la figure 3.21, sont les suivants :

– Cas I : ce cas est celui adopté dans la solution du diagramme DARTS de la figure 3.20. La deuxième tâche est donc liée à la première par une synchronisation et la transmission de la donnée entre ces deux tâches est faite au travers du module de données.

© Dunod – La photocopie non autorisée est un délit.

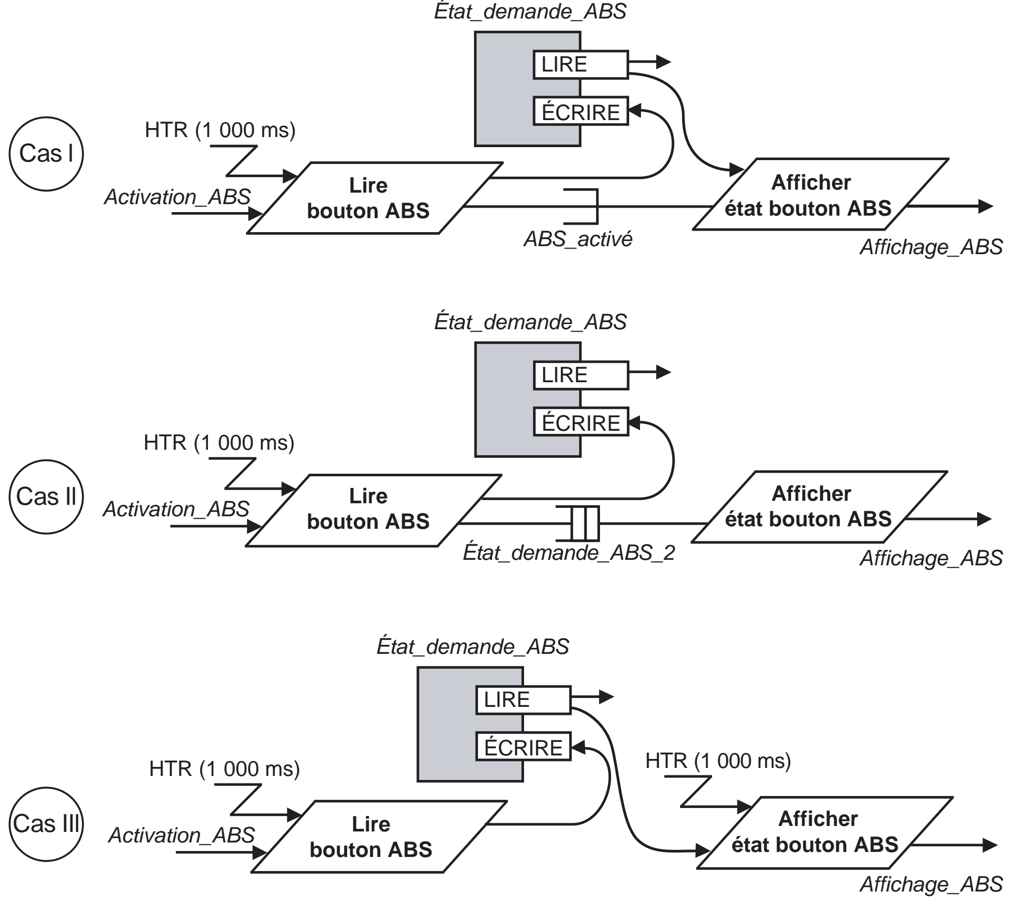


Figure 3.21 – Différentes méthodes de relations entre deux tâches communicantes d'un diagramme multitâche DARTS.

– Cas II : la deuxième tâche récupère la donnée par une boîte aux lettres qui sert en même temps d'activation. Ainsi, la lecture de la donnée par cette deuxième tâche vers le module de données devient inutile.

– Cas III : tous les liens de synchronisation entre les deux tâches sont supprimés et la deuxième tâche lit la donnée dans le module de données. Pour l'activation de la deuxième tâche, il est nécessaire de mettre un événement de type « horloge temps réel ». Les deux tâches ont donc des activations périodiques de même période mais non synchronisées. Pour conserver la relation de dépendance entre ces deux tâches, la seule solution réside dans l'utilisation des priorités : la première tâche « Lire bouton ABS » ayant la priorité la plus grande.

Reprenons le travail de conception du diagramme multitâche réalisé (figure 3.20) par la dernière phase, c'est-à-dire le regroupement ou le rassemblement de tâches. Pour obtenir un programme multitâche plus simple à valider et à réaliser, nous devons

réduire si possible le nombre de tâches. Le regroupement peut être effectué en amont directement sur les processus fonctionnels ou ensuite sur les tâches créées de façon automatique (un processus fonctionnel = une tâche). Ce regroupement de processus fonctionnels ou de tâches va s'effectuer sur les bases de critères de cohésion. Même si ce guide méthodologique n'est pas formel, il permet de travailler de façon efficace sur le diagramme SA-RT ou le diagramme DARTS. Ainsi, nous pouvons lister les critères possibles de regroupement de tâches :

– Cohésion temporelle : le regroupement concerne des processus fonctionnels qui doivent être activés par le même événement ou à la même période. La figure 3.22 montre l'exemple de deux processus fonctionnels qui se traduisent par deux tâches qui s'exécutent au même rythme.

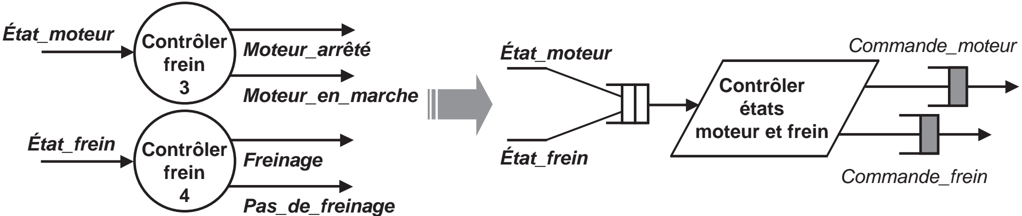


Figure 3.22 – Exemple d'un regroupement de tâches d'un diagramme multitâche DARTS basé sur une cohésion temporelle.

– Cohésion séquentielle : le regroupement concerne des processus fonctionnels qui doivent s'exécuter en séquence. La figure 3.23 montre deux processus fonctionnels qui doivent obligatoirement s'exécuter en séquence.

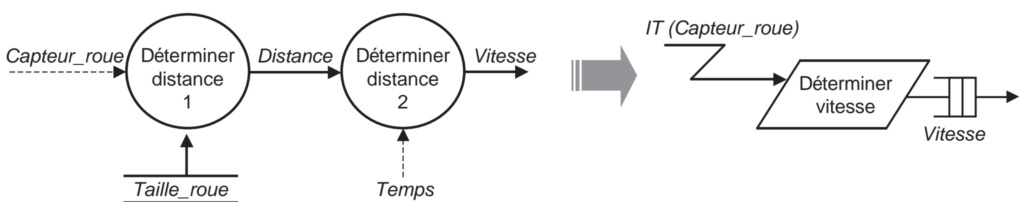


Figure 3.23 – Exemple d'un regroupement de tâches d'un diagramme multitâche DARTS basé sur une cohésion séquentielle.

– Cohérence du contrôle : le regroupement concerne des processus fonctionnels contrôlés par un ou plusieurs événements corrélés. La figure 3.24 montre deux processus fonctionnels qui remplissent une fonction très corrélée et qui sont cadencés par un processus de contrôle.

© Dunod – La photocopie non autorisée est un délit.

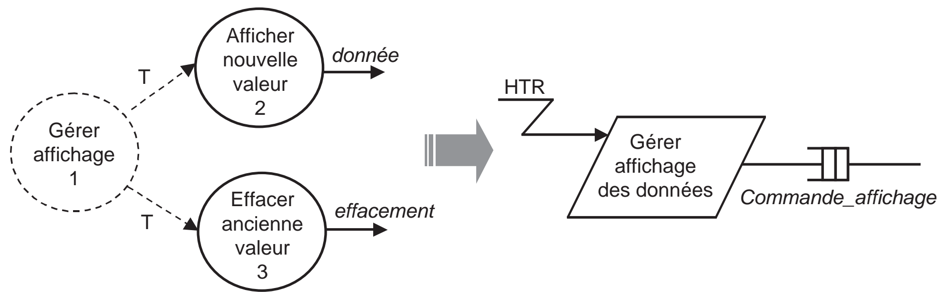


Figure 3.24 – Exemple d'un regroupement de tâches d'un diagramme multitâche DARTS basé sur une cohésion de contrôle.

– Cohésion fonctionnelle : le regroupement concerne des processus fonctionnels liés à une fonctionnalité unique. La figure 3.25 montre le regroupement de trois processus fonctionnels qui concourent à la même fonction (pilotage d'un robot).

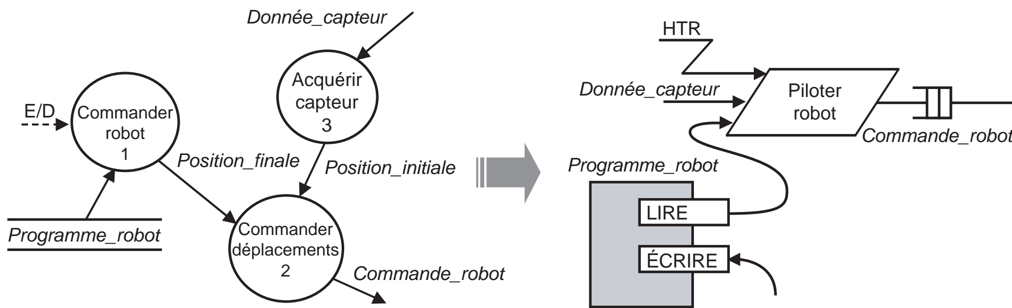


Figure 3.25 – Exemple d'un regroupement de tâches d'un diagramme multitâche DARTS basé sur une cohésion fonctionnelle.

Il est important de noter que certaines tâches remplissent parfois plusieurs de ces critères à la fois et donc la justification du regroupement est encore plus évidente.

En appliquant ces critères de cohésion au diagramme DARTS de l'application « système de freinage automobile » de la figure 3.20, nous obtenons un diagramme multitâche plus compact, puisque nous passons d'une architecture six tâches à quatre tâches.

Le premier regroupement concerne les deux tâches « Commander freinage » et « Contrôler application » qui remplissent à la fois le critère de cohésion de contrôle et le critère de cohésion fonctionnelle. Nous obtenons ainsi une seule tâche qui est appelée « Contrôler freinage ». Nous pouvons remarquer qu'il paraît important que cette tâche, issue du regroupement de plusieurs tâches, ne porte pas le même nom d'autant qu'elle ne correspond plus à un processus fonctionnel du diagramme SA-RT initial. Notons que ce regroupement économise deux synchronisations ; en revanche, la boîte aux lettres de type FIFO à écrasement a été remplacée par une boîte aux lettres de type FIFO sans écrasement afin de réaliser une synchronisation forte.

Le deuxième regroupement effectué concerne les deux tâches « Lire bouton ABS » et « Afficher état bouton ABS » qui remplissent à la fois le critère de cohésion temporelle et le critère de cohésion fonctionnelle. Nous obtenons alors une seule tâche qui est appelée « Lire et afficher demande ABS ». Dans ce cas aussi, le diagramme fait l'économie d'une synchronisation.
Nous obtenons ainsi un diagramme DARTS plus simple d'un point de vue multitâche. Cette clarification de l'architecture logicielle permet alors une compréhension plus aisée qui peut conduire à une validation plus complète.

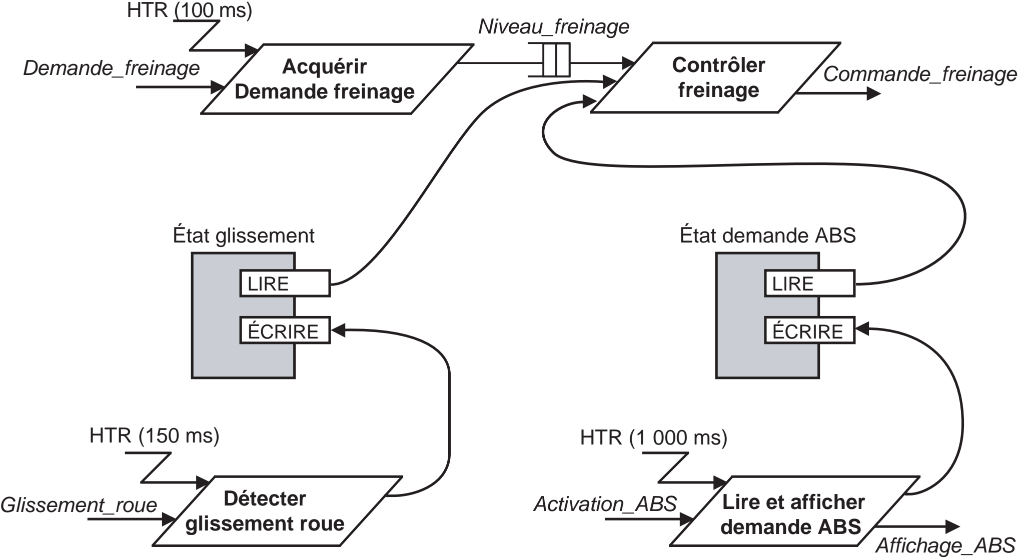


Figure 3.26 – Deuxième conception du diagramme multitâche DARTS obtenue à partir du regroupement de tâches du diagramme DARTS de la figure 3.20.

Ainsi, nous avons mis en œuvre la méthode de conception DARTS. À partir de la spécification réalisée à l'aide de la méthode d'analyse SA-RT, la transformation consiste à obtenir une architecture multitâche avec une démarche la plus automatisée possible. Pour cela, nous avons décomposé cette conception DARTS en quatre phases que nous pouvons rappeler :

– Phase 1 : Création des tâches.
– Phase 2 : Détermination du typage et de l'activation des tâches.
– Phase 3 : Mise en place des synchronisations et des transferts de données.
– Phase 4 : Regroupement de tâches.

© Dunod – La photocopie non autorisée est un délit.

3.3 Exemples de conception avec la méthode DARTS

Nous allons mettre en œuvre cette méthodologie DARTS pour les exemples plus complexes que l'exemple décrit jusqu'à présent « système de freinage automobile ».

3.3.1 Exemple : gestion de la sécurité d'une mine

Pour l'exemple décrit « gestion de la sécurité d'une mine », nous allons considérer le diagramme préliminaire spécifié dans le chapitre 2 et représenté sur la figure 2.32. Le diagramme préliminaire de cette application, qui comporte quatre processus fonctionnels et un processus de contrôle, sert de base pour la traduction en diagramme multitâche DARTS.

Après la mise en œuvre des quatre phases de la conception DARTS, nous obtenons un modèle multitâche possible de l'application, présenté sur la figure 3.27. Cette représentation multitâche intègre cinq tâches correspondant aux différents processus du diagramme préliminaire SA-RT. Nous pouvons noter immédiatement que le processus de contrôle a été traduit par une tâche qui est restée après la phase de regroupement. En effet, cette tâche présente une complexité algorithmique de bon niveau qui est mise en exergue par le diagramme état/transition de la figure 2.33.

Les deux processus fonctionnels d'acquisitions de données ont été traduits par deux tâches matérielles « Acquérir capteur méthane » et « Acquérir capteur eau » déclenchées par l'horloge temps réel. Elles possèdent des périodes très différentes liées à la

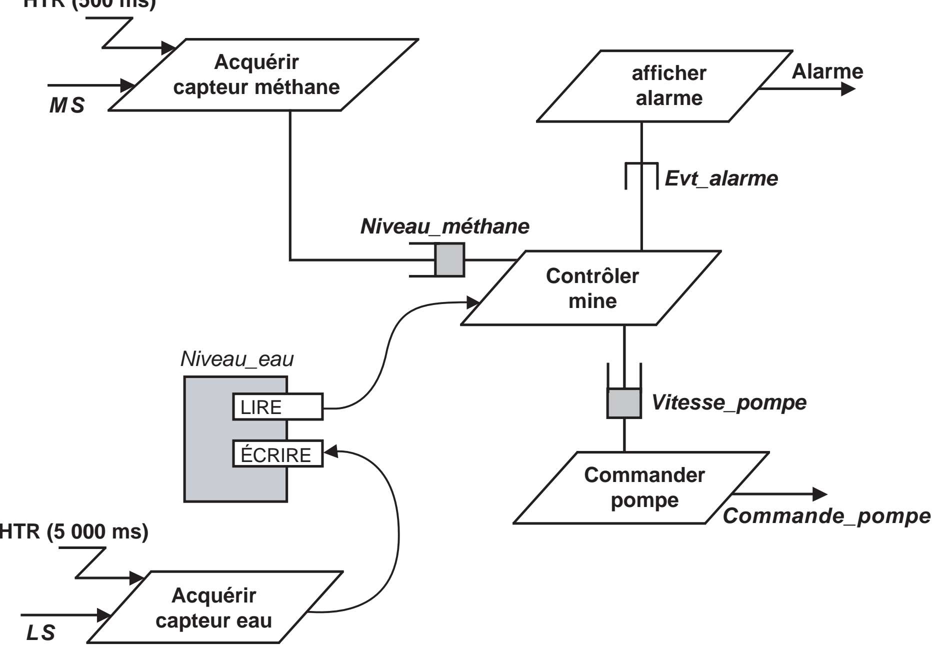


Figure 3.27 – Diagramme multitâche DARTS obtenu à partir du diagramme préliminaire de l'application « gestion de la sécurité d'une mine » de la figure 2.32.

dynamique des grandeurs physiques associées : 500 ms pour la mesure d'un taux de gaz (variation du méthane dans l'air : évolution rapide) et 5 s pour la mesure d'un niveau (eau dans un puisard : évolution lente). Cette différence de vitesse d'exécution nécessite l'emploi d'un module de données « *Niveau_eau* » pour le passage de paramètres afin de désynchroniser ces tâches. La tâche la plus rapide va cadencer le pilotage général de l'application en se synchronisant avec les tâches suivantes de régulation : « Contrôler mine », « Commander pompe » et « Afficher alarme ».

Remarque

La tâche d'acquisition de données, dont la période est la plus petite, va dicter la cadence principale d'exécution de l'application, c'est-à-dire synchroniser la régulation du procédé.

Il est intéressant de noter les traductions des synchronisations et des communications entre le diagramme SA-RT et le diagramme DARTS. En effet, les événements du diagramme SA-RT « *Niveau_LLS* » et « *Niveau_HLS* » ont été remplacés par un module de données qui sauvegarde la comparaison par rapport aux niveaux de consignes et aussi le niveau de l'eau. Ainsi, le flot de données direct entre les processus fonctionnels « Acquérir capteur eau » et « Commander pompe » se retrouve sous la forme d'une boîte aux lettres à une place à écrasement entre les deux tâches « Contrôler mine » et « Commander pompe ». De la même manière, les trois événements « *Consignes_respectées* », « *MS_L1_dépassée* » et « *MS_L2_dépassée* » sont remplacées par une boîte aux lettres à une place à écrasement entre les deux tâches « Acquérir capteur méthane » et « Contrôler mine ». Enfin, l'événement « *E/D* » de commande du processus fonctionnel « Afficher alarme » est traduit par une synchronisation « *Evt_alarme* » entre les deux tâches « Contrôler mine » et « Afficher alarme ».

La traduction du diagramme SA-RT de cette application « gestion de la sécurité d'une mine » a été réalisée de façon assez simple avec une très faible part à la phase 4 de conception avec des modifications de la traduction semi-automatique effectuée dans les trois premières phases. L'exemple suivant montre le contraire, où la phase 4 va modifier profondément le diagramme DARTS issu de la traduction semi-automatique du diagramme SA-RT.

3.3.2 Exemple : pilotage d'un four à verre

Pour l'exemple décrit « pilotage d'un four à verre », nous allons considérer le diagramme préliminaire spécifié dans le chapitre 2 et représenté sur la figure 2.36. Le diagramme préliminaire de cette application, qui comporte six processus fonctionnels et un processus de contrôle, sert de base pour la traduction en diagramme multitâche DARTS.

Après la mise en œuvre des quatre phases de la conception DARTS, nous obtenons un modèle multitâche possible de l'application, présenté sur la figure 3.28. Cette représentation multitâche intègre cinq tâches qui ne correspondent pas exactement aux différents processus du digramme préliminaire SA-RT. Nous pouvons noter immédiatement que le processus de contrôle n'a pas été traduit par une tâche. En effet, cette tâche aurait représenté une complexité algorithmique très faible comme le montre le diagramme état/transition de la figure 2.37.

© Dunod – La photocopie non autorisée est un délit.

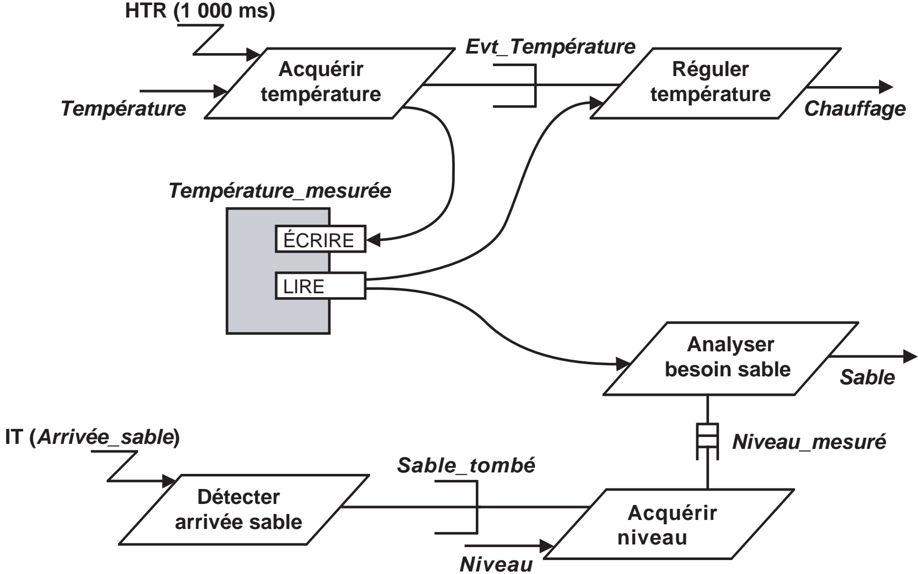


Figure 3.28 – Premier diagramme multitâche DARTS obtenu à partir du diagramme préliminaire de l'application « pilotage d'un four à verre » de la figure 2.36.

Nous pouvons noter immédiatement le regroupement qui a été effectué entre les processus fonctionnels « Analyser température » et « Chauffer four » pour donner une seule tâche appelée « Réguler température » sur des critères de cohésion fonctionnelle et séquentielle.

Deux des trois processus fonctionnels d'entrée de données ont été traduits par deux tâches matérielles « Acquérir température » et « Détecter arrivée sable ». La première tâche est une tâche périodique à scrutation déclenchée par l'horloge temps réel de période 1 s. En revanche, la deuxième tâche est une tâche matérielle déclenchée par une interruption « *Arrivée_sable* ». Cette tâche est utilisée pour déclencher de façon logicielle, par une synchronisation « *Sable_tombé* », la tâche d'acquisition « Acquérir niveau » traduction d'un processus fonctionnel d'entrée de données. Nous avons ici un exemple rare de processus d'acquisition qui est traduite par une tâche logicielle.

Nous avons une traduction très classique des transferts de données. Ainsi, le flot de données direct « *Niveau_mesuré* » entre les processus fonctionnels « Acquérir niveau » et « Analyser besoin sable » se retrouve sous la forme d'une boîte aux lettres du même nom entre les deux tâches aussi de même nom conformément à la règle 3.1. De la même manière, le transfert de la donnée « Température_mesurée » sous la forme d'une zone de stockage, partagée par les trois processus fonctionnels « Acquérir température », « Analyser température » et « Analyser besoin sable », est traduit par un module de données du même nom conformément à la règle 3.2 qui est accédé par les trois tâches « Acquérir température », « Réguler température » et « Analyser besoin sable ».

Nous pouvons noter la relation entre les deux tâches « Acquérir température », et « Réguler température » qui correspond aux modèles du cas I de la figure 3.21. Aussi, cette relation aurait pu être traduite par les autres modèles possibles (cas II et III de la figure 3.21).

Pour terminer l'étude de cette conception, nous allons analyser le comportement de cette architecture multitâche. Ainsi, l'exécution de la séquence des tâches « Détecter arrivée sable », « Acquérir niveau » et « Analyser besoin sable » est conditionnée par l'arrivée de l'interruption « *Arrivée_sable* » de déclenchement de la tâche initiale « Détecter arrivée sable ». Si aucun approvisionnement en sable ne s'effectue, alors l'interruption ne peut se produire et cette chaîne d'exécution est bloquée, car seule la tâche finale « Analyser besoin sable » peut modifier l'approvisionnement en sable. Afin de pallier ce problème, une solution basée sur un chien de garde est mise en place. La tâche « Analyser besoin sable » va créer un chien de garde destiné à la réveiller dans le cas où elle resterait inactive pendant une durée supérieure à 5 s. Si ce dysfonctionnement se produit, un événement lié à ce chien de garde est généré et peut être utilisé pour résoudre le blocage. Cela explique la présence sur la figure 3.29 de la tâche matérielle « Alarme » déclenchée par le chien de garde CG de durée 5 s. Cette tâche ne correspond à aucun processus fonctionnel du diagramme SA-RT. Lors de son exécution, la tâche « Alarme » va générer une synchronisation identique à celle générée par la tâche « Détecter arrivée sable ». Ainsi, la séquence va être débloquée, une nouvelle mesure du niveau du sable, qui aura diminué, conduira à une modification de la vitesse d'approvisionnement.

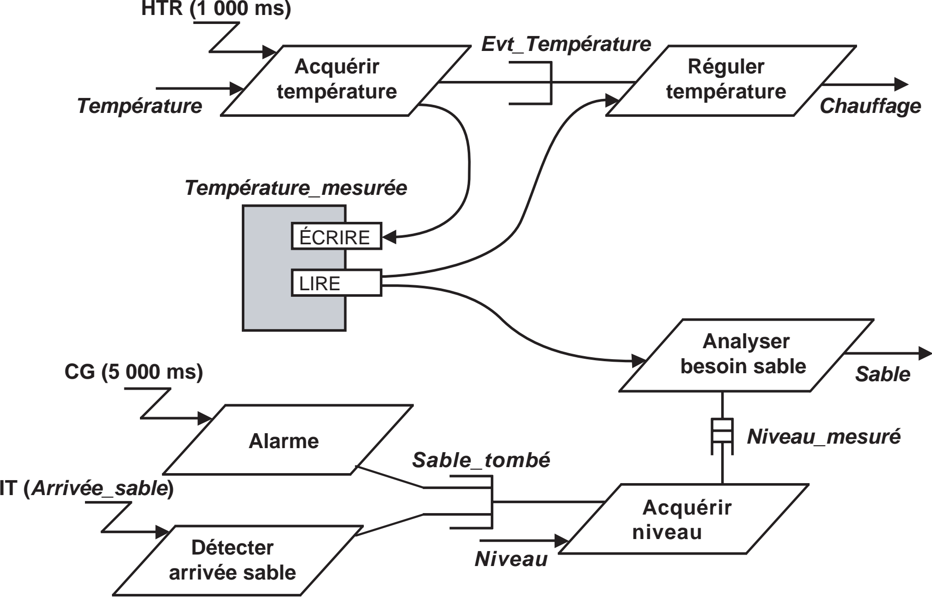


Figure 3.29 – Deuxième diagramme multitâche DARTS obtenu à partir du diagramme préliminaire de l'application « pilotage d'un four à verre » de la figure 2.36 avec la présence d'une tâche déclenchée par un chien de garde CG.

© Dunod – La photocopie non autorisée est un délit.

Nous obtenons ainsi un diagramme final de conception DARTS de six tâches dont deux n'ont pas d'équivalent direct dans le diagramme flots de données SA-RT de cette application. Une première tâche a été créée suite à un regroupement de tâche et une deuxième tâche a été mise en place pour répondre à un blocage du système par une solution de type chien de garde.
La modification du diagramme multitâche initial, issu d'une traduction automatique du diagramme flots données SA-RT, a été dans cet exemple de la « pilotage d'un four à verre » plus importante que dans l'exemple précédent « gestion de la sécurité d'une mine ». Le dernier exemple plus complexe « Commande d'un moteur à combustion » va montrer un travail de conception encore plus profond.

3.3.3 Exemple : Commande d'un moteur à combustion

Pour ce dernier exemple « commande d'un moteur à combustion », nous allons considérer le diagramme de préliminaire spécifié dans le chapitre 2 et représenté sur la figure 2.40. Le diagramme préliminaire de cette application, qui comporte neuf processus fonctionnels et un processus de contrôle, sert de base pour la traduction en diagramme multitâche DARTS.
Après la mise en œuvre des quatre phases de la conception DARTS, nous obtenons un modèle multitâche de l'application, présenté sur la figure 3.30. Cette représentation multitâche intègre huit tâches qui ne correspondent pas exactement aux différents processus du digramme préliminaire SA-RT. Nous pouvons noter immédiatement que le processus de contrôle n'a pas été traduit par une tâche. En effet, cette tâche aurait présenté une complexité algorithmique faible comme le montre le diagramme état/transition de la figure 2.41.
Nous pouvons aussi noter immédiatement le regroupement qui a été effectué entre les processus fonctionnels « Commander injection » et « Commander entrées gaz » pour donner une seule tâche appelée « Commander injection mélange » sur des critères de cohésion fonctionnelle et temporelle.
Les cinq processus fonctionnels d'acquisitions de données ont été traduits par cinq tâches matérielles portant les mêmes noms « Acquérir accélérateur », « Acquérir paramètres moteur », « Acquérir vitesse » « Acquérir capteurs pollution » et « Communiquer bus CAN » déclenchées par l'horloge temps réel. Elles possèdent des périodes très différentes liées à la dynamique des grandeurs physiques mesurées, soit respectivement 100 ms, 5 ms, 200 ms, 2 s et 50 ms.

Remarque

Nous pouvons noter que les périodes des différentes tâches d'acquisition sont choisies d'une part en fonction de la dynamique des grandeurs physiques à mesurer et d'autre part afin de minimiser la charge processeur. Réaliser une seule tâche d'acquisition, cadencée à la fréquence la plus grande, satisferait *a fortiori* la période de scrutation au sens de l'acquisition correcte de données (théorème de l'échantillonnage), mais conduirait à un besoin processeur fortement surdimensionné.

Comme cela a été souligné précédemment, la tâche la plus rapide va cadencer le pilotage principal de l'application, c'est-à-dire la commande du moteur (allumage) en fonction de la demande de vitesse. Cette fonction principale est réalisée en se synchronisant avec les tâches suivantes de régulation : « Élaborer commande moteur », « Commander allumage » et « Commander injection mélange ». De plus, nous

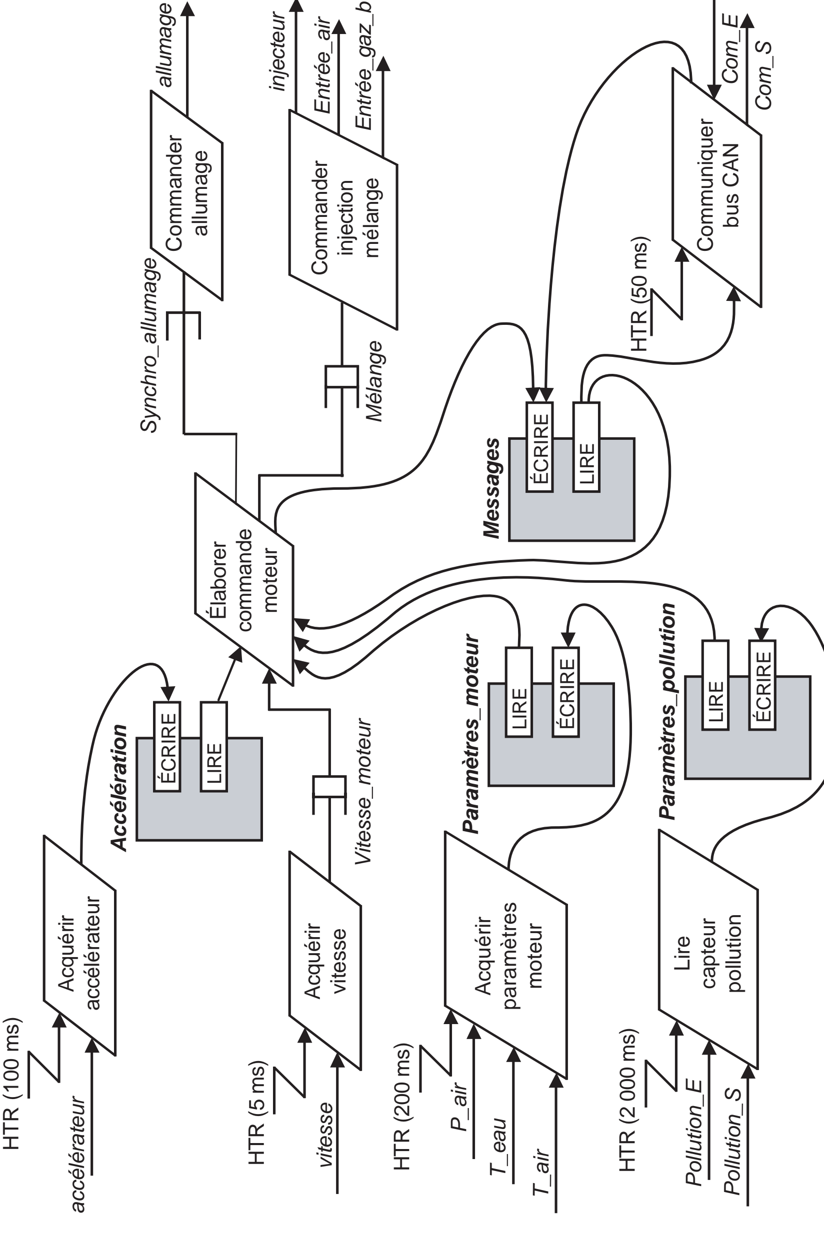


Figure 3.30 – Diagramme multitâche DARTS obtenue à partir du diagramme préliminaire de l'application « Commande d'un moteur à combustion » de la figure 2.40.

© Dunod – La photocopie non autorisée est un délit.

pouvons noter que la tâche matérielle « Communiquer bus CAN » est à la fois une tâche d'entrées et de sorties. Cette synchronisation forte est implémentée à l'aide de deux boîtes aux lettres à une place « *Vitesse_moteur* » et « *Mélange* », et d'une synchronisation « *Synchro_allumage* ».

Cette différence de vitesse d'exécution entre ces tâches d'acquisitions nécessite l'emploi de quatre modules de données pour le passage de paramètres afin de désynchroniser ces tâches : « *Accélération* », « *Paramètres_moteur* », « *Paramètres_pollution* » et « *Messages* ».

En conclusion de cet exemple, il important de souligner que la méthode DARTS est une méthode de conception non formelle qui permet d'exprimer sous une forme graphique définie une architecture multitâche. Dans ce sens, les diagrammes multitâches DARTS présentés en solution des exemples traités sont une des conceptions possibles de l'application ; mais il existe évidemment de nombreuses autres solutions possibles qui peuvent aussi évoluer en fonction de l'implémentation.

4 • ARCHITECTURES SYSTÈMES

Avant de présenter les outils et méthodes utilisés lors du développement de programmes de contrôle-commande dans différents langages de programmation, il est nécessaire de présenter l'architecture matérielle et logicielle sur laquelle ils s'appuient. En effet, le chapitre 3 présente une méthode de conception multitâche, qui est mise en œuvre à partir du chapitre 6 : une implémentation multitâche repose la plupart du temps sur un système d'exploitation supportant le multitâche, reposant lui-même, au plus bas niveau, sur une architecture matérielle permettant l'entrelacement temporel de plusieurs programmes. Les concepts sous-jacents, regroupés communément dans des ouvrages d'architecture et de systèmes d'exploitations, sont nécessaires à une bonne compréhension des outils de développement. Ce chapitre donne donc un bref aperçu des problèmes posés par le multitâche et des solutions communément proposées : il sert de base d'architecture matérielle et logicielle des systèmes informatisés pour le lecteur non familiarisé avec ce domaine.

4.1 Architecture matérielle

4.1.1 Définitions de base

Le chapitre 3 met l'accent sur la nécessité de programmer les applications de contrôle-commande en utilisant plusieurs tâches s'exécutant en parallèle. Cependant, les systèmes informatisés fonctionnent de façon séquentielle : une unité de calcul et de traitement exécute les instructions composant un programme, les unes à la suite des autres. Dans les applications de contrôle-commande, les unités de traitement peuvent être des microprocesseurs ou des microcontrôleurs.

■ Processeur

Un **microprocesseur** est une unité de traitement optimisée pour le calcul. Généralement plus rapide qu'un microcontrôleur, c'est le cœur des micro-ordinateurs. Il est capable d'effectuer des **entrées/sorties** (moyens de communication entre l'ordinateur et le monde extérieur : capteurs, actionneurs, périphériques divers) via des circuits spécialisés (ports série, ports parallèle, USB, etc.) et surtout, dans le cadre des applications de contrôle-commande, via des cartes spécialisées enfichables nommées **cartes d'acquisition**. Par exemple, on trouvera des cartes d'acquisition pouvant s'enficher dans des ports au format PCI, PCMCIA ou autre (voir § 4.1.4, p. 128).

© Dunod – La photocopie non autorisée est un délit.

Un **microcontrôleur** est une unité de traitement optimisée pour les entrées sorties. Généralement, un microcontrôleur est associé directement à plusieurs types d'entrées/sorties, et il ne nécessite pas l'apport de cartes enfichables supplémentaires. Il est de ce fait plus compact qu'un microprocesseur muni de cartes d'acquisition, mais moins performant en terme de calculs.

Les unités de calculs sont caractérisées par une fréquence d'horloge en hertz (Hz). Cette fréquence correspond au nombre de **cycles** par seconde effectués par l'unité de calcul. Chaque instruction de bas niveau exécutée par l'unité de calcul nécessite de un à quelques cycles d'horloge.

La fréquence d'horloge des microprocesseurs actuels se situe aux alentours de quelques gigahertz (GHz), alors que les microcontrôleurs sont cadencés (ce terme vient du fait que l'horloge interne donne la cadence) à quelques dizaines de mégahertz (MHz).

Dans la suite, le terme générique **processeur** sera utilisé pour désigner un microcontrôleur ou un microprocesseur. Il est à noter que de plus en plus de processeurs dupliquent certaines parties centrales afin d'être capables d'exécuter plusieurs instructions en parallèle. Cependant, afin de simplifier la présentation, on peut les voir comme un regroupement de deux (ou plus) processeurs séquentiels.

■ Mémoire

Les instructions d'un programme, ainsi que les données et le contexte d'exécution du programme sont stockés dans la **mémoire centrale** (mémoire vive comme la RAM pour *Random Access Memory*, ou mémoire FLASH plus lente que la RAM mais rémanente).

Une mémoire se caractérise par sa taille, la taille des **mots mémoires** (taille de données que le processeur et la mémoire échangent à chaque accès) et son **temps d'accès**. Le temps d'accès est donné en fréquence, soit en nombre d'accès par seconde. Généralement, la mémoire centrale est relativement lente par rapport au processeur. Dans le cas des microcontrôleurs, la fréquence d'accès à la mémoire centrale est souvent la même que la fréquence du microcontrôleur : pour charger ou déposer un mot en mémoire, le processeur n'a pas à attendre plus d'un cycle. Dans le cas des microprocesseurs, la mémoire centrale est souvent 5 à 10 fois plus lente que le microprocesseur, cela implique qu'une communication avec la mémoire peut durer 5 à 10 cycles d'horloge du microprocesseur. Ainsi, lors du déroulement d'un programme le chargement d'une instruction à partir de la mémoire centrale, ou l'accès à une donnée, peut durer plusieurs cycles processeur. Cependant, comparée à la mémoire de masse, comme le disque dur par exemple, avec un temps d'accès de l'ordre de 10 millisecondes, une mémoire centrale, pouvant fonctionner à plusieurs centaines de mégahertz (temps d'accès de l'ordre de quelques nanosecondes) fait figure de mémoire extrêmement rapide.

Dans un système informatisé, la taille est caractérisée en **octets**. Un octet est une entité composée de 8 **bits** (bit = *binary digit*, chiffre binaire). Toute information (instruction, donnée, information sur l'état interne, etc.) est représentée en binaire sur un certain nombre d'octets (voir § 4.1.3). L'octet est la plus petite entité adressable : chaque octet possède une adresse en mémoire centrale.

Fonctionnement d'un processeur

Les instructions d'un programme, mémorisées sous forme d'octets en mémoire centrale, sont stockées séquentiellement. L'exécution d'un programme par un processeur consiste donc, schématiquement, à charger une instruction à partir de la mémoire, l'exécuter, aller chercher l'instruction suivante et ainsi de suite. Un programme est donc caractérisé par un **point d'entrée** : l'adresse de sa première instruction.

Les instructions sont composées d'un certain nombre d'octets (variable en fonction de l'instruction). Chaque octet est adressable et peut être obtenu grâce à son adresse en mémoire.

Le transfert d'un mot entre la mémoire centrale et le processeur s'effectue à l'aide du **bus mémoire**. La fréquence du bus correspond à la fréquence de la mémoire (généralement, la fréquence maximale d'une mémoire centrale est supérieure ou égale à celle du bus mémoire : c'est la fréquence du bus mémoire qui sert de référence à la vitesse de la mémoire). Un bus mémoire est un bus parallèle : schématiquement, il est capable de faire passer en parallèle un nombre de bits correspondant au mot mémoire. Le nombre maximal d'octets par seconde (débit maximal) échangés entre le processeur et la mémoire correspond donc à *fréquence du bus mémoire* × *taille du mot mémoire en octets*. Typiquement, la largeur d'un mot mémoire est de 8, 16, 32 ou 64 bits.

Les **registres** sont des mémoires internes au processeur permettant de stocker les données manipulées : les calculs ne sont pas effectués directement à partir du contenu de la mémoire centrale, mais les données de la mémoire sont d'abord chargées dans des registres, à partir desquels elles sont manipulées par l'unité de calcul. Le nombre, le nom la fonction, et la taille des registres varient en fonction du processeur, cependant certains registres ont des fonctions communes d'un processeur à un autre. Ainsi, un registre particulier, nommé accumulateur, sert sur certains processeurs à stocker le résultat de chaque calcul arithmétique ou logique. Sur d'autres processeurs, divers registres généraux peuvent servir à cette fonction. La taille de ce ou ces registres est importante car elle donne la taille maximale des données manipulées en une seule instruction. Cette taille s'appelle le **mot machine**, et correspond généralement à la taille du mot mémoire. Pour les microcontrôleurs, on trouve fréquemment des mots machines de 8 ou 16 bits et rarement 32 bits, alors que pour les microprocesseurs, on trouve plutôt des mots machine de l'ordre de 32 ou 64 bits.

Le processeur comprend notamment (figure 4.1) :

- un séquenceur de commandes, permettant d'initier et de contrôler le déroulement d'une instruction. C'est cet élément qui cadence le fonctionnement du processeur à la fréquence d'horloge du processeur ;
- un registre nommé **compteur ordinal** dont le rôle est de donner l'adresse en mémoire de la prochaine instruction à exécuter ;
- un registre d'instruction qui stocke l'instruction en cours d'exécution ;
- un registre d'adresse permettant de requérir ou modifier des données ou instructions se situant à une certaine adresse en mémoire centrale ;
- une **unité arithmétique et logique** (UAL), se chargeant de l'exécution des instructions comme un calcul en nombres entiers, une opération logique, ou une

© Dunod – La photocopie non autorisée est un délit.

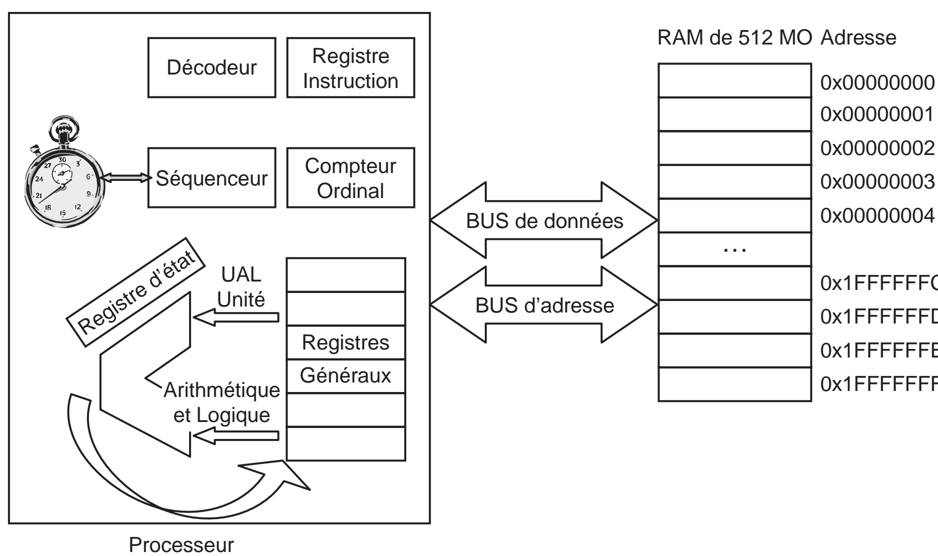


Figure 4.1 – Représentation simplifiée d'un processeur et de la mémoire centrale : les adresses mémoire sont notées en représentation hexadécimale (base 16).

opération de manipulation binaire (opérations arithmétiques, logiques, manipulation des représentations binaires) ;

– des **registres généraux** permettant entre autres choses de stocker les données utilisées lors des calculs, les résultats, etc. ;
– un **registre d'état** (***flags***), dont un des rôles est de communiquer des informations d'état de l'unité arithmétique et logique.

■ Exemple d'exécution d'un programme

Afin d'illustrer le fonctionnement d'un processeur, voyons comment serait traité le programme simple ci-après par un processeur de type PENTIUM® :

```
si i>j alors i:=i-1
sinon i:=i+5
fin si
```

Ce langage n'est pas directement compréhensible par un processeur. Il faut d'abord le compiler en un fichier exécutable contenant du code directement compréhensible par le processeur : ce code s'appelle le **code machine**. Le code machine est quasiment incompréhensible (c'est une suite de nombres), mais il existe un langage, très proche du langage machine, nommé l'**assembleur**, qui permet de l'exprimer de façon lisible. Contrairement aux langages de haut niveau, l'assembleur est hautement dépendant du processeur sous-jacent puisqu'il se base directement sur les registres et instructions spécifiques du processeur. Il en résulte que les assembleurs des différentes familles de processeurs sont très différents les uns des autres.
Souvent, le code machine ainsi que les adresses en mémoire sont représentés en notation **hexadécimale** (base 16, voir § 4.1.3, p. 120).

Le tableau 4.1 présente la traduction du programme simple en langage de haut niveau en code machine. Afin de comprendre ce code, la correspondance en assembleur est donnée pour chaque instruction.

Tableau 4.1 – Traduction binaire et assembleur d'un programme simple.

Sens	Adresse de l'instruction en notation hexadécimale	Code machine en notation hexadécimale	Instruction en assembleur
i est stocké dans un registre général (eax)	0040101E	A1 E4 A6 40 00	mov eax,dword ptr [i (40A6E4h)]
j est stocké dans un autre registre général (ecx)	00401023	8B 0D E0 A6 40 00	mov ecx,dword ptr [j (40A6E0h)]
Préparation du registre de pile	00401029	83 C4 10	add esp,10h
Comparaison de i et j, le résultat, se trouvent sous la forme d'un bit à 0 ou 1 dans le registre d'états, est utilisé par l'instruction suivante	0040102C	3B C1	cmp eax,ecx
Si i < j sauter à l'adresse 0x401033	0040102E	7E 03	jle 401033h
eax :=eax-1	00401030	48	dec eax
Sauter à l'adresse 0x401036	00401031	EB 03	jmp 401036h
eax :=eax+5	00401033	83 C0 05	add eax,5
Le registre eax est copié à l'adresse de i	00401036	A3 E4 A6 40 00	mov dword ptr [i (40A6E4h)],eax

Lorsque le processeur doit exécuter cette portion de programme, le compteur ordinal possède l'adresse (en notation hexadécimale) 0x0040101E. Cette adresse est disposée dans le registre d'adresse et communiquée via le bus à la mémoire centrale. Cela a pour effet de copier le contenu de cette case mémoire dans le registre d'instruction. Une instruction est composée d'un **CODOP** (code opération) et d'**opérandes**. Le CODOP de cette instruction est 0xA1, qui signifie « copier le contenu d'une adresse mémoire 32 bits dans le registre général nommé EAX », et l'opérande est l'adresse de la case mémoire à recopier. Cette instruction est décodée, puis séquencée, pendant que le compteur ordinal est incrémenté de la longueur de l'ins-

© Dunod – La photocopie non autorisée est un délit.

truction (il passe donc à 0x00401023). Certaines instructions nécessitent plusieurs accès mémoire, et le décodeur d'instruction pourra se charger de requérir les éventuels opérandes manquants lors du premier accès. Une requête à la mémoire centrale a lieu afin de lire le contenu du mot machine contenant la valeur de *i* (4 octets à partir de l'adresse 0x0040A6E4, correspondant à la variable *i* dans le langage de haut niveau) afin de le stocker dans un registre général du processeur, en l'occurrence le registre 32 bits nommé EAX. La taille d'un mot machine étant de 32 bits sur les microprocesseurs de type PENTIUM®, un seul accès à la mémoire est nécessaire pour recopier *i* dans le registre EAX. La prochaine instruction peut alors être lue à partir de la mémoire afin d'être recopiée dans le registre d'instruction, et exécutée à son tour.

La seconde instruction est aussi une instruction de déplacement de donnée entre la mémoire et un registre du processeur (son CODOP est donné sur les 2 octets 0x8B0D). Notons que tous les calculs (ici comparaison de *i* et *j*) nécessitent que les opérandes soient présents dans un registre du processeur.

L'instruction `cmp eax,ecx` compare les registres contenant respectivement les valeurs de *i* et *j*. Cette instruction est exécutée par l'unité arithmétique et logique, et son résultat est accessible via le registre d'état, dont le bit « *less or equal* » est mis à 1 si le résultat de la dernière comparaison était « *inférieur ou égal* ».

Les deux premières instructions sont donc de type échange registres ↔ mémoire centrale, la troisième est une opération arithmétique, et la quatrième instruction est un saut conditionnel `jle 00401033h`. Ce saut est conditionné par le résultat du dernier calcul arithmétique (en l'occurrence une comparaison) et se base donc sur le registre d'état. Ici, si la dernière comparaison a donné le résultat « inférieur ou égal », la prochaine instruction à exécuter se trouve à l'adresse 0x00401033, sinon elle se trouve à l'adresse suivante (déjà contenue dans le compteur ordinal). Les sauts conditionnels ont pour effet de modifier le fonctionnement normal du compteur ordinal.

L'instruction suivante, c'est-à-dire la décrémentation du registre contenant *i*, n'est donc exécutée que si le résultat du test est négatif (on peut remarquer qu'il a été inversé par rapport au langage de haut niveau). Dans ce cas, l'instruction suivante (`jmp`) est exécutée, ce qui permet de sauter les instructions du bloc d'instructions correspondant au cas `sinon`.

Dans le cas où le test a été positif, le cas `alors` est sauté pour aller directement au cas `sinon`.

Enfin, dans les deux cas, l'instruction se trouvant à l'adresse 0x00401033 est exécutée, c'est-à-dire que le contenu du registre EAX est transféré à l'adresse mémoire correspondant à la variable *i*.

Les optimisations ayant lieu au cœur des processeurs actuels rendent difficile une description globale du cheminement des informations. En effet, il existe différentes optimisations permettant d'accélérer la vitesse moyenne de traitement :

– le préchargement des instructions (appelé généralement *prefetch*) permet au bus d'être employé pour le chargement des prochaines instructions pressenties. Dans ce cas, il est possible que toutes les instructions possibles soient préchargées et stockées dans un tampon d'instructions afin d'être plus rapidement accessibles ;

– le *pipeline* permet de commencer l'exécution des prochaines instructions avant même que l'instruction courante ne soit terminée ;
– la **mémoire cache** : il y a souvent un rapport de vitesse de 5 à 10 entre la vitesse des microprocesseurs et la vitesse de la mémoire centrale. Cela implique qu'à chaque transfert d'information entre la mémoire et le processeur, le processeur doit attendre les instructions ou données dont il a besoin. Or, la plupart des programmes contiennent des boucles (donc utilisent souvent les mêmes instructions) et accèdent souvent aux mêmes données. Cela est appelé le **principe de localité**. Afin d'accélérer les accès à la mémoire, une mémoire cache est ajoutée entre le processeur et la mémoire : très rapide (sa fréquence d'accès est la même que celle du processeur ou bien seulement deux fois plus lente), de petite taille (de l'ordre du mégaoctet). La mémoire cache conserve les éléments les plus récemment accédés, ce qui, en vertu du principe de localité accélère grandement la vitesse de traitement ;
– de plus en plus de microprocesseurs du commerce dupliquent certains circuits afin de pouvoir effectuer des traitements parallèles. Ainsi, certains processeurs utilisent une double unité de calcul, voire même ont un cœur double, ce qui leur permet presque d'être équivalents à deux processeurs.

Cet exemple permet d'introduire différents éléments de base de l'architecture matérielle, sans toutefois entrer dans les détails. Il montre, de par l'utilisation de la base hexadécimale, qu'il est nécessaire d'avoir quelques notions sur les bases typiquement utilisées en informatique : les bases binaire et hexadécimales sont donc présentées au paragraphe 4.1.3. En effet, les adressages sont typiquement donnés en base 16 qui est une représentation condensée très pratique du binaire. De même, la maîtrise de la base binaire, et donc hexadécimale (pour la représentation condensée), se montre indispensable pour toute personne ayant à programmer un système de contrôle-commande. En effet, ce type de système fait souvent appel à de la programmation de bas niveau (au niveau octet, voire même bit à bit). Afin de maîtriser les opérations de base bit à bit, il est nécessaire d'avoir des notions en algèbre booléenne, dont une application directe est l'algèbre binaire.

De plus, il est important pour un concepteur ou programmeur d'applications de contrôle-commande de maîtriser des notions de base en terme de types d'entrées/sorties afin de décider de l'emploi de tel ou tel type de matériel (microcontrôleur, microprocesseur).

Enfin, cet exemple démontre que même les instructions de haut niveau les plus simples se décomposent le plus souvent en plusieurs instructions de bas niveau (niveau assembleur et instructions machine). Ce point est primordial au regard des applications multitâche comme le sont les applications temps réel. Nous présenterons les difficultés que cela engendre dans le paragraphe 4.2.

4.1.2 Algèbre de Boole

L'algèbre booléenne est l'algèbre définie sur le domaine {vrai, faux} ou de façon équivalente sur le domaine {1,0}, ce qui explique son emploi dans toutes les architectures informatisées (système binaire). Georges Boole, mathématicien anglais (1815-1864) proposa cette algèbre prisée aussi bien par les logiciens que par les

© Dunod – La photocopie non autorisée est un délit.

informaticiens d'aujourd'hui. Son nom est devenu célèbre grâce à son algèbre et au type booléen présent dans de nombreux langages de programmation.

Toute opération booléenne est définie dans le domaine {vrai, faux}. Cependant, étant donné que le but de cette partie est de permettre de comprendre les différents types de manipulation binaire, il est important que le lecteur sache que vrai est assimilé à 1 en binaire, et faux à 0.

On définit usuellement trois opérateurs (appelés **connecteurs**) de base : le **complément** (ou « **non** ») noté $\neg$, la **conjonction** (ou « **et** ») notée $\wedge$, et la disjonction (ou « **ou** ») notée $\vee$. Tous les connecteurs de l'algèbre booléenne peuvent être obtenus à partir de ces trois connecteurs (il est aussi possible de la définir à l'aide de deux connecteurs).

La valeur d'une formule booléenne, composée de connecteurs et de variables booléennes (dont la valeur peut être vrai ou faux) peut être représentée à l'aide d'une table de vérité. Chaque ligne de la table donne une valeur possible aux variables, et la valeur obtenue de la formule étudiée. La table de vérité donne donc toutes les valeurs possibles d'une formule en fonction des variables. Par exemple, pour la variable $a \in$ {vrai, faux}, la table de vérité de $\neg a$ est donnée dans le tableau 4.2. Ce tableau présente aussi les tables de vérités des principaux connecteurs : « et », « ou », « implication », « ou exclusif » appelé **xor**, « équivalence ». Cette table est présentée sous la forme algébrique en utilisant les connecteurs booléens classiques, puis reprise sous forme algébrique proche du binaire. En effet, le « et » booléen a des propriétés similaires à la multiplication, puisque le « faux » est absorbant, de même que le 0 pour la multiplication. Le « vrai » est absorbant pour le « ou », ce qui fait penser à une somme dans laquelle on ne considérerait que les résultat 0 ou différent de 0 (dans ce cas, 1). Le « ou exclusif » est vrai si un et un seul des opérandes est vrai, cela signifie qu'il est vrai si ses deux opérandes sont différents l'un de l'autre (le domaine n'ayant que deux valeurs possibles). Le « ou exclusif » correspond à l'usage du mot « ou » dans le langage usuel. Ainsi, le « ou » correspondant à l'affirmation « je me rendrai à la réunion en train ou en avion » est un ou exclusif. *A contrario*, le « ou » logique classique, dit « ou inclusif », laisse la possibilité aux deux variables d'être vraies. L'équivalence est vue comme l'égalité de ses deux opérandes.

On dit que deux formules sont équivalentes si elles possèdent la même table de vérité.

L'implication est donnée ici à titre indicatif. La formule $a \Rightarrow b$ est équivalente à la formule $(\neg a) \vee b$, ce qui signifie que le seul cas où elle est fausse correspond au cas où a est vrai, et b est faux. Ainsi, l'affirmation « si j'avais des pouvoirs magiques, alors je soulèverais la tour Eiffel », qui est une implication, ne peut pas être infirmée tant que la personne qui prétend cela n'a pas effectivement de pouvoirs magiques.

Il suffit de considérer la façon dont on démontre une implication en mathématiques : faire l'hypothèse que a est vrai, et démontrer, dans ce cas seulement, que b est vrai.

La comparaison entre les opérateurs algébriques somme et produit ne s'arrête pas à une sémantique proche des opérateurs booléens « et » et « ou ». En effet, les opérateurs booléens possèdent les mêmes propriétés de distributivité et d'associativité, plus d'autres liées à la taille du domaine des booléens. Le tableau 4.3 donne quelques propriétés algébriques, ainsi que les représentations graphiques des opérateurs logiques dans la norme américaine (MIL STD 806) et la norme européenne (IEC 617).

Tableau 4.2 – Tables de vérité des connecteurs les plus usités.

a	¬a
faux	vrai
vrai	faux

a	b	a ∧ b	a ∨ b	a ⊕ b	a ⇔ b	a ⇒ b
faux	faux	faux	faux	faux	vrai	vrai
faux	vrai	faux	vrai	vrai	faux	vrai
vrai	faux	faux	vrai	vrai	faux	faux
vrai	vrai	vrai	vrai	faux	vrai	vrai

a	b	ab	a + b	a ? b	a = b	a ⇒ b
0	0	0	0	0	1	1
0	1	0	1	1	0	1
1	0	0	1	1	0	0
1	1	1	1	0	1	1

Les propriétés algébriques peuvent être vérifiées très simplement à l'aide de tables de vérité (tableau 4.3, pages suivantes).

Les principales propriétés à retenir sont les propriétés absorbantes du « faux » sur « et » (*i.e.* 0 . a = 0) et les propriétés absorbantes du « vrai » sur « ou » (*i.e.* 1 + a = 1). En effet, pour les manipulations de nombres binaires, notamment la création de masques binaires (voir § 4.1.4), très utile lors d'accès au matériel par des primitives bas niveau, ces règles sont fondamentales.

4.1.3 Représentation de l'information

Bien que les premières machines à calcul de l'ère électromécanique fussent en base décimale, depuis l'ère de l'électronique, tous les ordinateurs utilisent la base binaire pour effectuer des calculs. La raison en est simple : une tension à deux valeurs possibles, (0, *n* volts) ou bien (– *n*, + *n* volts), permet l'utilisation intensive de transistors (portes électroniques s'ouvrant ou se fermant en fonction d'une faible tension d'entrée). Les transistors sont les éléments de base des processeurs modernes, ils permettent d'implémenter aisément toutes les fonctions d'algèbre booléenne, qui servent aussi à effectuer des calculs en arithmétique binaire. En effet, chaque chiffre

© Dunod – La photocopie non autorisée est un délit.

Tableau 4.3 – Propriétés algébriques usuelles des booléens.

Priorités des opérateurs		
La priorité des opérateurs est définie comme suit (supérieur signifie est plus prioritaire que) : $\neg > \wedge > \vee > \Rightarrow > \Leftrightarrow$	$c \vee a \wedge b \Leftrightarrow d$ se lit $(c \vee (a \wedge b)) \Leftrightarrow d$	$c + ab = d$ se lit $(c + (ab)) = d$
Commutativité		
Les connecteurs « et », « ou », « xor », « équivalent » sont commutatifs.	$a \wedge b \Leftrightarrow b \wedge a$ $a \vee b \Leftrightarrow b \vee a$ $a \oplus b \Leftrightarrow b \oplus a$ $(a \Leftrightarrow b) \Leftrightarrow (b \Leftrightarrow a)$	$ab = ba$ $a + b = b + a$ $a \oplus b = b \oplus a$ ou (a ? b) = (b ? a) $(a = b) = (b = a)$
Associativité		
« et » et « ou » sont associatifs ; attention « xor » n'est pas associatif. Par défaut, l'associativité choisie est à gauche.	$a \wedge (b \wedge c) \Leftrightarrow (a \wedge b) \wedge c$ $a \vee (b \vee c) \Leftrightarrow (a \vee b) \vee c$ $a \oplus (b \oplus c) \neq (a \oplus b) \oplus c$	$a(bc) = (ab)c$ $a + (b + c) = (a + b) + c$ $(a \neq (b \neq c)) \neq ((a \neq b) \neq c)$
Éléments neutres et absorbants		
Vrai est absorbant pour le « ou » mais neutre pour le « et ». Faux est absorbant pour le « et » et neutre pour le « ou ».	$a \vee \text{vrai} \Leftrightarrow \text{vrai}$ $a \wedge \text{vrai} \Leftrightarrow a$ $a \vee \text{faux} \Leftrightarrow a$ $a \wedge \text{faux} \Leftrightarrow \text{faux}$ $a \vee a \Leftrightarrow a$ $a \wedge a \Leftrightarrow a$	$a + 1 = 1$ $a \,.\, 1 = a$ $a + 0 = a$ $a \,.\, 0 = 0$ $a + a = a$ $a \,.\, a = a$
Idempotence et complémentation		
Le « ou » et le « et » sont idempotents. Règles de complémentation.	$a \vee a \Leftrightarrow a$ $a \wedge a \Leftrightarrow a$ $a \vee \neg a \Leftrightarrow \text{vrai}$ $a \wedge \neg a \Leftrightarrow \text{faux}$	$a + a = a$ $a \,.\, a = a$ $a + \neg a = \text{vrai}$ $a \,.\, \neg a = \text{faux}$
Distributivité		
Le « et » est distributif sur le « ou ». Le « ou » est distributif sur le « et ».	$a \wedge (b \vee c) \Leftrightarrow a \wedge b \vee a \wedge c$ $a \vee b \wedge c \Leftrightarrow (a \vee b) \wedge (a \vee c)$	$a(b + c) = ab + ac$ $a + bc = (a + b)(a + c)$

Tableau 4.3 – Propriétés algébriques usuelles des booléens (suite).

Lois de De Morgan		
L'impact du « non » sur le « et » et le « ou »	$\neg(a \wedge b) \Leftrightarrow \neg a \vee \neg b$ $\neg(a \vee b) \Leftrightarrow \neg a \wedge \neg b$	$\neg(ab) = \neg a + \neg b$ $\neg(a + b) = \neg a \neg b$
Équivalences de quelques formules		
	$a \vee a \wedge b \Leftrightarrow a$ $a \wedge (a \vee b) \Leftrightarrow a$ $a \vee \neg a \wedge b \Leftrightarrow a \vee b$ $a \wedge (\neg a \vee b) \Leftrightarrow a \wedge b$ $a \Rightarrow b \Leftrightarrow \neg a \vee b$	$a + ab = a$ $a(a + b) = a$ $a + (\neg ab) = a + b$ $a(\neg a + b) = a + b$ $a \Rightarrow b = \neg a + b$

	Norme MIL STD 806	Norme IEC 617
IDENTITÉ	a → a	a [1] a
NON	a → ¬a	a [1] ¬a
ET	a, b → ab	a, b [&] ab
OU	a, b → a+b	a, b [?1] a+b
XOR (OU exclusif)	a, b → $a \oplus b$	a, b [=1] $a \oplus b$
NAND	a, b → ¬(ab)	a, b [&] ¬(ab)
NOR	a, b → ¬(a+b)	a, b [?1] ¬(a+b)
EGAL	a, b → $\neg(a \oplus b)$	a, b [=1] $\neg(a \oplus b)$

étant un bit (*binary digit*) valant 0 ou 1, absence ou présence de courant, l'algèbre booléenne peut être utilisée pour les calculs numériques.

Cette section a pour objet d'introduire l'arithmétique binaire, ainsi que quelques notions de changements de bases. Ces concepts sont fondamentaux pour la programmation de bas niveau (programmation des cartes d'acquisition notamment).

© Dunod – La photocopie non autorisée est un délit.

■ Changements de base

Un nombre se représente dans une base b par une suite de chiffres b_i compris entre 0 et $b - 1$. Tout nombre se décompose de façon unique en chiffres dans une base b : $b_k \cdot b^k + b_{k-1} \cdot b^{k-1} + \ldots + b_1 \cdot b^1 + b_0 \cdot b^0$. Le nombre s'écrit dans la base b : $b_k b_{k-1} \ldots b_1 b_0$.

Ainsi, en système décimal, le nombre 3 412 correspond à $3 \times 10^3 + 4 \times 10^2 + 1 \times 10^1 + 2 \times 10^0$. Tout nombre x, que l'on a l'habitude de manipuler sous sa forme décimale, correspond à une quantité, qui se décompose de façon unique dans toute base b sous la forme d'une suite de chiffres $b_k b_{k-1} \ldots b_1 b_0$, avec $b_i = \left(\left\lfloor \frac{x}{b^i} \right\rfloor \equiv b \right)$ où $\lfloor a \rfloor$ est la partie entière inférieure de a et $a \equiv b$ est a modulo b (*i.e.* reste de la division de a par b). Ainsi, cette quantité peut se décomposer en base 1 en 3 412 « bâtons ».

Pour passer une quantité d'une base à une autre, il suffit d'utiliser la division euclidienne « $a = bq + r$ » : lorsque l'on divise un nombre par une base b, on obtient comme reste le chiffre des unités. Ainsi, si l'on souhaite effectuer des divisions euclidiennes successives de 3 412 par la base 10, on obtient $3\,412 = 341 \times 10 + 2$. Le chiffre des unités est donc 2, et il y a 341 dizaines. $341 = 34 \times 10 + 1$, il y a donc une dizaine et 34 centaines. $34 = 3 \times 10 + 4$, il y a donc 4 centaines et 3 milliers. $3 = 0 \times 10 + 3$, il n'y a donc que 3 milliers. 3 412 se décompose donc en base 10 par 3 milliers, 4 centaines, 1 dizaine, et 2 unités, ce qui est évident.

Si l'on souhaite convertir 3 412 dans une base $b > 1$ quelconque, il suffit donc d'effectuer des divisions euclidiennes successives, la première donne le chiffre des unités, la seconde celui des « b-zaines », …, la $n^{\text{ième}}$ celui des « b^{n-1}-zaines ». La figure 4.2 représente les conversions de 3 412 en base 10, 2, 8 et 16.

Par convention, dans la suite, les nombres en binaire seront préfixés par la lettre *b* (sauf non ambiguïté), les nombres en octal seront préfixés par la lettre *o*, les nombres en hexadécimal seront préfixés par « *0x* », alors que les nombres en décimal ne seront pas préfixés du tout.

Certains changements de base se voient simplifiés lorsque l'on passe d'une base b à une puissance de b et vice-versa. Les bases fréquemment utilisées en informatique, en plus du binaire qui est l'unité physique utilisée, sont l'**octal** (base 8) et l'**hexadécimal** (base 16). En effet, un nombre représenté en binaire est très long, et difficilement lisible. Or $8 = 2^3$ et $16 = 2^4$. Chaque chiffre octal correspond donc à 3 chiffres binaires, et chaque chiffre hexadécimal correspond à 4 chiffres binaires. Pour passer de la base 2 à la base 8, il suffit donc de regrouper les bits par 3 (en partant bien entendu des bits de poids faible !!!), alors que pour passer de la base 8 à la base 2, il suffit de représenter chaque chiffre octal par 3 bits. Il en va de même pour la base 16, sauf que les regroupements sont de 4 bits. La figure 4.3 donne un exemple de conversion binaire-octal et binaire-hexadécimal.

L'une des illustrations du changement de base par division euclidienne peut être faite à l'aide d'un changement de base direct entre deux bases non usuelles : par exemple, passage de la base 8 à la base 16 (figure 4.4). Cela peut sembler troublant, car la division euclidienne se fait directement en base 8 (les règles d'addition et de soustraction sont différentes de celles de la base 10 que chacun a l'habitude de manipuler,

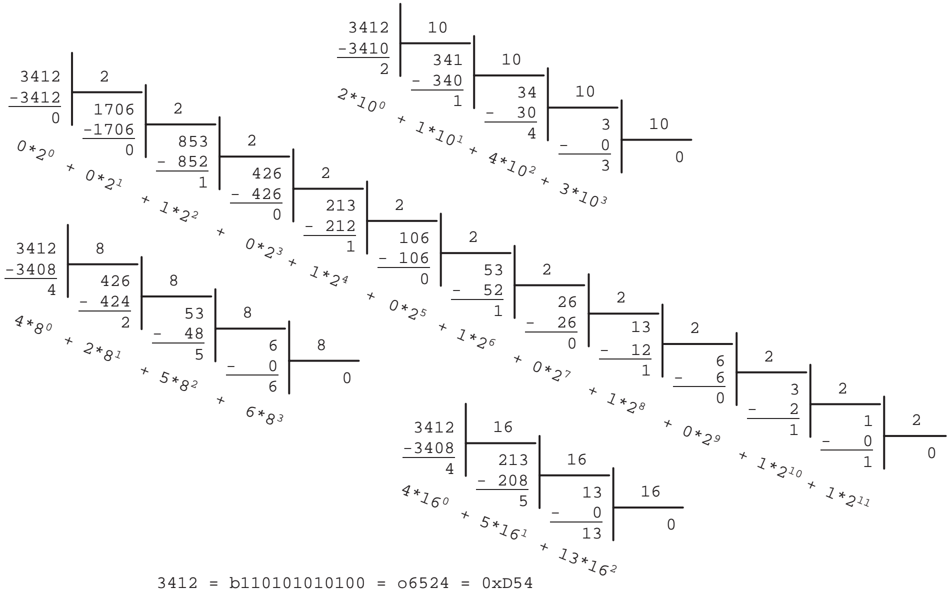


Figure 4.2 – Conversion de 3 412 en base 10, 2, 8, et 16.

```
1425 =b 10 110 010 001
        2   6   2   1      ⇨ 1425 =o2621
1425 =b 101 1001 0001
         5    9    1       ⇨ 1425 =0x591
```

Figure 4.3 – Illustration de changement de base :
binaire-octal et binaire-hexadécimal.

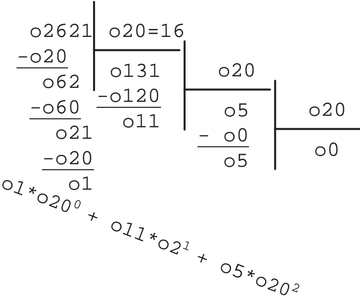


Figure 4.4 – Changement de base direct entre base 8 et 16.

© Dunod – La photocopie non autorisée est un délit.

ainsi par exemple $o5 \times o2 = o12$). La division euclidienne pour convertir un nombre octal en hexadécimal consiste donc à diviser un nombre donné en base 8 par 16, la division a lieu bien entendu en base 8, c'est donc une division par $o20$. Notons cependant qu'au vu des propriétés des bases 8 et 16 par rapport à la base 2, il est plus rapide de transformer le nombre octal en binaire, puis de transformer ce nombre par regroupements de 4 bits en hexadécimal. Cependant, le calcul montre bien que $o2621 = 0x591$.
L'un des avantages du binaire est le faible nombre d'opérations de base nécessaires à l'exécution des opérations arithmétiques. En binaire, il n'y a que des tables de 0 et de 1 à manipuler. L'addition est extrêmement simple (figure 4.5).

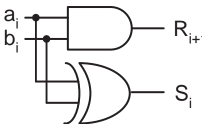
```
        1 1 1
 57    b111001
+24   +b011000
 81   b1010001
```

Figure 4.5 – Addition de deux nombres entiers en binaire.

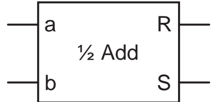
Représentation binaire des entiers positifs

La taille de la représentation d'un entier en machine est bien entendu bornée. Le plus souvent, un entier peut être représenté sur 8 bits (type caractère), 16 bits (type entier court), 32 bits (type entier long). Souvent, comme c'est le cas en langage C, le type entier (sans qualificatif supplémentaire) correspond à la taille du mot machine, soit le plus souvent 32 bits sur un microprocesseur, souvent moins sur un microcontrôleur. Le choix de la taille de la représentation d'un entier influe bien entendu sur le domaine possible de l'entier (voir tableau 4.4) puisqu'à l'aide de n bits, on peut représenter 2^n valeurs différentes. Notons que, par convention, le bit de poids faible est numéroté 0, et que les bits sont numérotés dans l'ordre croissant jusqu'au bit de poids fort, numéroté n–1 pour un nombre de n bits.
Afin d'illustrer le rapport profond qui lit l'algèbre booléenne avec la représentation binaire, la figure 4.6 donne le circuit logique qui pourrait être utilisé pour additionner deux nombres binaires. Soient a et b deux entiers représentés sur n bits respectivement par $a_{n-1} \dots a_1 a_0$ et $b_{n-1} b_{n-2} \dots b_1 b_0$. La réalisation de $a + b = c$ (représenté par $c_{n-1} c_{n-2} \dots c_1 c_0$) est présentée sur la figure 4.6. Dans la réalité, les connecteurs logiques sont réalisés à l'aide de transistors, et tout le cœur des processeurs se base sur l'algèbre booléenne.
Dans le langage Ada, la taille de représentation des nombres peut varier en fonction de l'implémentation et de l'architecture matérielle (taille du mot machine) sous-jacente. Cependant, un programmeur Ada peut définir lui-même le domaine d'un entier.
Les techniques utilisées pour représenter les entiers signés et les nombres fractionnaires (points fixes et flottants) sont présentées en annexe A.

Demi-additionneur : $(a+b)*^{i}2 = R*2^{i+1} + S*2^{i}$
(non prise en compte de la retenue précédente)



Encapsulation du ½ additionneur
(correspond au ½ additionneur à gauche)



Additionneur : $(a+b)*2^{i} + R*2^{i} = R*2^{i+1} + S*2^{i}$
(prise en compte de la retenue précédente)

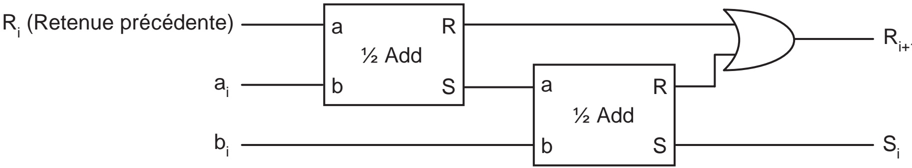


Figure 4.6 – Addition de deux entiers réalisée à l'aide de l'algèbre booléenne.

Tableau 4.4 – Intervalles de représentation des entiers.

Taille (en bits)	Domaine $[0..2^n–1]$	Nom du type
8	[0..255], [0..0xFF]	C : unsigned char LabVIEW : U8
16	[0..65535], [0..0xFFFF]	C : unsigned short LabVIEW : U16
32	[0.. 4294967295], [0..0xFFFFFFFF]	C : unsigned long LabVIEW : U32

■ Opérations arithmétiques et logiques

Des exemples d'additions ont été présentés dans la section précédente. La multiplication utilise les mêmes règles qu'en numération décimale. La division, quant à elle, est une division entière et utilise les mêmes règles qu'en numération décimale.
Il est à noter que dans toute base *b*, une multiplication par la base correspond à décaler les chiffres vers la gauche, afin d'introduire un 0 comme chiffre de poids faible. Ainsi, $32 \times 10 = 320$. Il en est de même pour le binaire : multiplier par 2 consiste à décaler à gauche. La division entière par la base consiste à enlever le chiffre de poids faible : $321/10 = 32$. Cela correspond à un décalage des chiffres vers la droite.
Lorsque l'on doit programmer à bas niveau, il arrive fréquemment que l'on ait à créer des octets de toute pièce (par exemple créer un octet dont le $i^{\text{ème}}$ bit et le $j^{\text{ème}}$ bit sont à 1, les autres étant à 0 afin de configurer une carte d'acquisition) ou bien à

© Dunod – La photocopie non autorisée est un délit.

Tableau 4.5 – Quelques repères en binaire et hexadécimale.

Propriété algébrique	Utilisations type	Exemples
$16 = 2^4$	Passage aisé binaire ↔ hexadécimal	Octet dont le bit 5 (6e bit) est à 1 : b0010 0000 = 0x20 Octet dont tous les bits sont à 1, sauf le bit 3 : b1111 0111 = 0xF7
« 0 » absorbant et « 1 » neutre sur le « et »	Vérifier qu'un bit est à 1 dans un octet	Le bit 2 (3e bit) de l'octet k est à 1 si et seulement si : k et 0x04 ≠ 0 Le bit de poids fort de l'octet k est à 1 si et seulement si : k et 0x80 ≠ 0 Le bit i de l'octet k est à 1 si et seulement si : k et (1 décalé à gauche de i) ≠ 0
	Vérifier qu'un bit est à 0 dans un octet	Le bit 2 de l'octet k est à 0 si et seulement si : k et 0x04 = 0 Le bit de poids fort de l'octet k est à 0 si et seulement si : k et 0x80 = 0 Le bit i de l'octet k est à 0 si et seulement si : k et (1 décalé à gauche de i) = 0
	Accéder à une partie d'un nombre	Obtenir le 2e octet d'un nombre entier k codé sur 32 bits : (k décalé à gauche de 8) et 0xFF
	Mettre à 0 un bit dans un nombre	Mettre à 0 le bit 5 de l'octet k : k :=k et non(0x20) Mettre à 0 le bit i de l'octet k : k :=k et non(1 décalé à gauche de i)
« 1 » absorbant et « 0 » neutre sur le « ou »	Mettre à 1 un bit dans un nombre	Mettre à 1 le bit 5 de l'octet k : k := k ou 0x20 Mettre à 1 le bit i de l'octet k : k := k ou (1 décalé à gauche de i)

tester certains bits (celui-ci est-il à 1, celui-là est-il à 0 ?). Pour cela, on utilise les opérateurs binaires qui sont directement issus de l'algèbre booléenne. Il est donc bon d'avoir quelques repères binaires et booléens (voir tableau 4.5). Rappelons (voir § 4.1.2) que le 0 est absorbant pour le « et », et neutre pour le « ou », et que le 1 est absorbant pour le « ou » et neutre pour le « et ». Rappelons de plus que la représentation binaire est peu lisible, et que dans un programme, il est généralement plus pratique d'exprimer les valeurs binaires en hexadécimal (1 chiffre hexadécimal = 4 bits).

Enfin, voici une petite technique qui permet de trouver rapidement la représentation binaire d'un nombre, à condition que celui-ci ne soit pas trop grand. Pour cela, il suffit de connaître les puissances de 2, et de faire mentalement quelques soustractions. Soit un nombre décimal, par exemple 155, que l'on veut représenter en binaire (noter qu'au minimum, ce sera un octet non signé, car 155 > 127). On choisit la plus grande puissance de 2 inférieure ou égale à 155, soit 128. On place un 1 qui correspond à 128 (point n'est nécessaire de préciser qu'il s'agit de 2^7), on retranche 128 à 155, il reste donc 27 à exprimer en binaire. On passe alors toutes les puissances de 2 de façon décroissante : à la suite du 1 correspondant à 128, on note 0 pour 64, 0 pour 32, 1 pour 16 (reste 11), 1 pour 8 (reste 3), 0 pour 4, 1 pour 2 (reste 1), et 1 pour 1. On obtient donc la représentation binaire b10011011.

■ Récapitulatif sur la représentation des données

Dans tous les systèmes informatisés, toute information, qu'elle soit instruction ou bien donnée, est représentée sous forme binaire. Il en va de même pour les caractères : les caractères sont représentés en mémoire sous la forme d'un code numérique. Ce code numérique dérive de la table de caractères normalisée par l'ANSI, qui a défini, sur 7 bits, 128 caractères de base non accentués. Cette table adoptée dans les années 1960, appelée **ASCII** (*American Standard Code for Information Interchange*), a eu plusieurs dérivés, ainsi, les codes **ASCII étendus** ajoutent un bit à la représentation ASCII, ce qui permet d'obtenir 256 caractères. Cela est loin d'être suffisant pour représenter tous les caractères internationaux, donc les 128 caractères ajoutés (codes 128 à 255) dépendent d'une table chargée en fonction du pays. Le même principe est repris dans l'implémentation Unicode de la norme ISO/IEC 10646. Cette implémentation, reprise dans la plupart des logiciels actuels, propose des tables de caractères sur 8, 16 et même 32 bits. Lorsque sous MS Windows® on lance une application console et que les accents ne sont pas bien affichés, cela est souvent dû au fait que l'application ne prend pas en compte la bonne page d'ASCII étendu. Lorsque l'on navigue sur internet et qu'une page asiatique n'est pas convenablement affichée, cela est dû au fait que la plupart des navigateurs européens se basent sur la page Unicode Latin-1, qui regroupe les caractères d'Europe de l'ouest en utilisant une représentation sur 16 bits.

Les informations en mémoire peuvent correspondre, en fonction du contexte, à des instructions/opérandes, ou données quel que soit leur type. La figure 4.7 présente un récapitulatif des différents formats utilisés pour représenter les données de base.

© Dunod – La photocopie non autorisée est un délit.

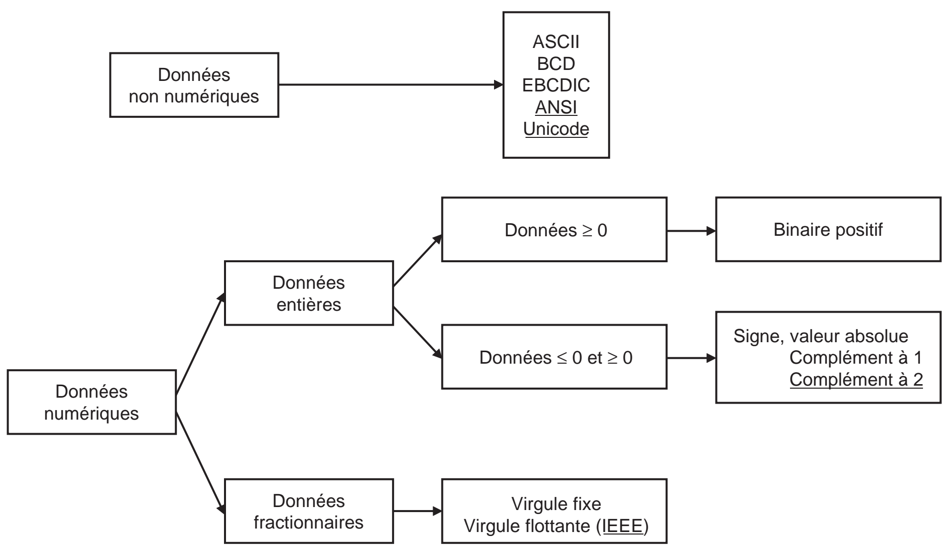


Figure 4.7 – Différents formats de représentation de données.

4.1.4 Notions sur les entrées/sorties

Le dernier point d'architecture matérielle abordé concerne les différents types d'entrées/sorties permettant à une application de communiquer avec le monde extérieur. Lorsque le processeur doit effectuer une entrée/sortie, les temps d'accès au matériel sont sans commune mesure avec le temps de cycle processeur. Par conséquent, afin d'éviter au processeur de perdre du temps de calcul lors de la réalisation d'une entrée/sortie, le mécanisme des interruptions est utilisé.

■ Interruptions matérielles

Lorsque le processeur doit lire des informations de l'extérieur, il peut le faire de deux façons. La plus simple, mais la moins efficace, consiste à faire de la **scrutation** (aussi appelée **attente active** ou ***polling***) : le processeur lit en boucle ce qui arrive sur une entrée, sans savoir avant de lire s'il y a des données « intéressantes ». La plus efficace, mais qui n'est pas toujours possible, consiste à utiliser les **interruptions**. Le processeur peut être en train d'effectuer un calcul quelconque, mais lorsque des données sont prêtes, le processeur est interrompu dans son traitement, et un traitement (généralement très court) appelé **routine de traitement d'interruption** (en anglais ***ISR*** pour *Interrupt Service Routine*) est effectué. Ainsi, le processeur ne gaspille pas de temps à vérifier la présence d'informations intéressantes : il en est prévenu par interruption. Généralement, c'est un dispositif physique dédié, appelé contrôleur de bus d'entrées/sorties, qui est chargé de générer des interruptions.

Il en va de même pour l'envoi de données vers un bus d'entrées/sorties : le débit autorisé par le bus pour le transfert d'informations est tellement faible par rapport à la vitesse du processeur qu'il serait obligatoire d'attendre de nombreux cycles

processeurs entre l'envoi de chaque mot mémoire. Grâce aux interruptions, on utilise généralement un système de ***buffer*** (zone de stockage d'informations) de données à émettre : le contrôleur d'entrées/sorties se charge donc d'émettre les données à la vitesse du bus, et prévient par interruption le processeur lorsque le *buffer* est vide ou quasi-vide afin que le processeur fournisse de nouvelles données à émettre. Ainsi, le processeur n'a pas à attendre que le bus d'entrées/sorties soit libre, et il peut se consacrer à d'autres traitements.

Étant donné que le traitement effectué par un processeur est séquentiel, comment une interruption peut-elle être traitée ? L'état d'un programme est totalement caractérisé par l'état des registres (notamment du compteur ordinal) et bien entendu sa mémoire propre en mémoire centrale (instructions, données…). Si l'ISR n'affecte pas la mémoire propre du programme en mémoire centrale, et si les registres sont sauvegardés, il est possible de changer sur interruption la valeur du compteur ordinal pour la « brancher » sur l'adresse de début des instructions de l'ISR, la routine s'exécute alors, et à la fin de la routine, l'état du processeur est restauré. Le programme en cours d'exécution est donc interrompu, mais à la fin du traitement de l'interruption, le processeur est remis dans le même état qu'au moment de l'interruption : il continue donc son exécution comme s'il n'avait pas été interrompu (figure 4.8).

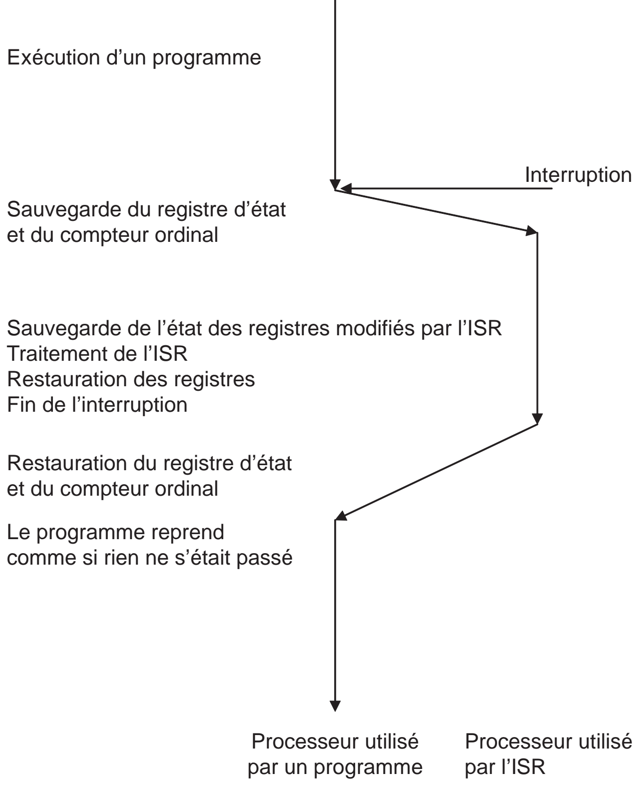


Figure 4.8 – Prise en compte d'une interruption matérielle.

© Dunod – La photocopie non autorisée est un délit.

Il peut être possible d'ignorer complètement une interruption : cela s'appelle **désarmer** une interruption. L'interruption pourra à nouveau être prise en compte lorsqu'elle sera **armée**.
Pour retarder le traitement d'une interruption (par exemple lors du traitement d'une autre interruption), on peut **masquer** une interruption. Dans ce cas, les requêtes d'interruptions sont si possible mémorisées : ces requêtes seront traitées lorsque l'interruption sera **démasquée**.

■ **Bus d'entrées/sorties**

Au niveau logiciel, la possibilité d'utiliser les interruptions pour éviter l'attente active se traduit par la possibilité d'effectuer des lectures bloquantes. De plus, la plupart du temps, les envois sur les bus d'entrées/sorties sont *bufferisés* et **suspensifs** pour le programme les émettant. Cela signifie qu'un programme qui envoie des données sur un bus d'entrées/sorties (par exemple, un programme qui effectue un affichage sur un terminal alphanumérique) envoie des données dans le *buffer*, puis se suspend (le processeur peut être utilisé à autre chose) jusqu'à ce qu'il y ait à nouveau de la place dans le *buffer*, etc., jusqu'à épuisement des données à émettre. Généralement, les entrées/sorties utilisant des bus sont suspensives pour l'émission de données et bloquantes pour la réception, grâce à l'utilisation des interruptions matérielles.
Le nombre des interruptions matérielles est limité sur un système informatisé (par exemple, il y en a 16 sur les PC). Les interruptions sont utilisées par les bus d'entrées/sorties et autres éléments d'entrées/sorties :

– **bus série** (norme RS-232) : l'un des **bus externes** (permettant de brancher un élément externe au système informatisé) les plus utilisés encore aujourd'hui pour communiquer avec du matériel d'acquisition/commande « intelligent ». La communication série (transfert bit à bit) est très répandue aussi bien au niveau des microprocesseurs, qui cohabitent le plus souvent avec des puces capables de gérer des entrées/sorties série, qu'au niveau des microcontrôleurs, qui intègrent très souvent ce type de communication. Ce bus est très souvent utilisé avec un protocole bidirectionnel de type ASCII (ce sont des caractères qui sont échangés), permettant au processeur d'envoyer des commandes ou des informations de configuration, et de recevoir des informations de statut ou des données d'acquisition. Le type de matériel utilisant la communication série est très large : centrales d'acquisition de températures sur thermocouples, capteurs GPS, centrales inertielles intégrant accéléromètres, gyromètres et magnétomètres, écrans alphanumériques, etc. C'est un bus relativement simple à programmer, car il existe de nombreuses bibliothèques logicielles permettant de tirer parti de ce bus capable de transporter des flots de données jusqu'à 128 000 bits/s ;
– **bus parallèle** : presque obsolète, ce bus externe permet une transmission de 8 bits en parallèle à des débits de 3 Mo/s. Il y a quelques années, c'était un bus très utilisé pour communiquer avec des imprimantes ;
– **bus USB** (*Universal Serial Bus*) : ce bus externe héritier du bus série permet, dans sa version 1.0, des débits de l'ordre de 1,5 Mo/s et dans sa version 2.0 des débits aux alentours de 60 Mo/s. Dans le monde de la micro-informatique, ce bus a supplanté les bus série et parallèle pour permettre la communication avec

des éléments variés qui peuvent être relativement gourmands en débits de données (souris, clavier, webcams, imprimantes, scanners, modems, capteur GPS…). Il a l'avantage de pouvoir transporter l'alimentation vers le matériel lorsque celui-ci consomme peu d'énergie. Cependant, bien qu'il soit possible qu'il s'impose dans quelques années sur le marché des éléments temps réel, il n'a pas encore fait de réelle apparition dans ce domaine, et peu de microcontrôleurs intègrent un bus USB ;

– **bus *FireWire*** (norme IEEE 1394) : ce bus série externe concurrence l'USB 2.0 dans sa version *a*, offrant des débits de l'ordre de 50 Mo/s. Ce bus est très utilisé pour l'acquisition vidéo numérique (le format de compression vidéo utilisé, le format DV, est peu compressé pour éviter les pertes de qualité). Dans sa norme *b*, offrant des débits de l'ordre de 400 Mo/s, il concurrence le bus SCSI pour le branchement de disques durs/lecteurs DVD ou CD externes. Il est pour l'instant peu présent sur les microcontrôleurs et dans le monde des systèmes temps réel ;
– **bus SCSI** (*Small Computer System Interface*) : en perpétuelle amélioration, ce bus externe parallèle fournissant des débits allant de 5 Mo/s (SCSI-1) à 320 Mo/s (Ultra-4-SCSI), permet notamment la connexion d'un ordinateur à des périphériques de stockage externe (disques durs externes, etc.) ;
– **bus PCMCIA** (*Personal Computer Memory Card International Association*) : ce bus externe parallèle permettant des débits de l'ordre de 130 Mo/s, présent surtout sur les ordinateurs portables, permet l'utilisation de périphériques compacts (équivalent des périphériques PCI des ordinateurs de bureau). De format presque identique (modulo un petit adaptateur), le format Compact Flash est de plus en plus utilisé pour des éléments compacts facilement embarquables (comme des GPS pour PC de poche) ;
– **bus ISA** (*Industry Standard Architecture*) : presque obsolète, ce **bus interne** (lorsqu'il est présent, se trouve sur la carte mère des ordinateurs utilisant un microprocesseur) permet de connecter différents types de cartes internes (vieille carte son, vieille carte d'acquisition…) peu gourmandes en débit de données ;
– **bus PCI** (*Peripheral Component Interconnect*) : très utilisé pour brancher des cartes internes (cartes d'acquisition, cartes son, etc.) dans les ordinateurs, ce bus interne parallèle fournit des débits de l'ordre de 1 Go/s ;
– **bus AGP** (*Advanced Graphic Port*) : utilisé exclusivement pour connecter des cartes vidéo (carte se chargeant de l'affichage graphique), ce bus interne parallèle est l'un des bus d'entrées/sorties les plus rapides avec des débits qui augmentent continûment (la version 8x permet un débit de 2,1 Go/s) ;
– **bus ATA** (*Advanced Technology Attachment*) : généralement appelé **bus IDE** (*Integrated Drive Electronics*), ce bus parallèle interne est généralement utilisé pour communiquer avec les éléments internes de stockage (disque dur, lecteur/graveur de CD ou DVD, etc.). En perpétuelle évolution, ce bus permet des débits de 133 Mo/s dans sa version Ultra DMA/133.

En théorie, le débit maximal (bande passante) d'un bus parallèle devrait être obtenu par la *fréquence du bus* × *largeur du bus*, cependant, sur certains types de bus (comme ISA), les informations de gestion du bus prennent une partie non négligeable de la bande passante. Dans l'autre sens, de plus en plus de bus (AGP, SCSI…) permettent

© Dunod – La photocopie non autorisée est un délit.

Tableau 4.6 – Récapitulatifs sur les bus d'entrées/sorties.

Nom	Norme	Largeur en bits	Fréquence	Débit maximal théorique	Applications typiques
			bus externes		
Série	RS 232	1		240 Ko/s	Tout type de capteurs intelligents, modem…
Parallèle		8		3 Mo/s	Presque plus utilisé
USB	*Universal Serial Bus*	1		60 Mo/s	Vidéo, imprimante, souris, clavier…
FireWire	IEEE 1394	1		50 à 400 Mo/s	Vidéo, disques externes…
SCSI	*Small Computer System Interface*	8/16/32	4,77 à 80 MHz	320 Mo/s	Disque externe…
PCMCIA	*Personal Computer Memory Card International Association*	16	33 MHz	130 Mo/s	Mémoire flash, Cartes d'acquisition, réseau…
			bus internes		
ISA	*Industry Standard Architecture*	16	8 MHz	8 Mo/s	Obsolète, servait au branchement de cartes internes (son, acquisition…)
PCI	*Peripheral Component Interconnect*	32/64	133 MHz	1 Go/s	Cœur reliant les différents bus d'entrées/sorties au microprocesseur
AGP	*Advanced Graphic Port*	32	66×8	2,1 Go/s	Carte vidéo
ATA/IDE	*Advanced Technology Attachment / Integrated Drive Electronics*	16	66	133 Mo/s	Disque interne

l'envoi de plusieurs données par cycle d'horloge. C'est d'ailleurs le cas pour les bus spécialisés dans le transfert de données entre le processeur et la mémoire centrale.

Finalement, le bus d'entrées/sorties le plus utilisé avec des éléments d'acquisition externes est le bus série. Quel que soit le bus employé, il est important de conserver en mémoire que le fait d'accéder à un périphérique via un bus d'entrées/sorties (que ce soit en lecture ou en écriture de données) est suspensif pour un programme.

Le tableau 4.7 donne quelques mesures communément utilisées dans les systèmes informatisés, permettant de mieux appréhender le tableau 4.6.

Tableau 4.7 – Mesures de fréquences et de taille.

Mesures de fréquence		
kHz	10^3 cycles/s	Fréquences audibles (< 15 kHz)
MHz	10^6 cycles/s	Fréquence du bus de données (plusieurs centaines de MHz)
GHz	10^9 cycles/s	Fréquence d'un microprocesseur (quelques GHz)
THz	10^{12} cycles/s	Fréquence ondulatoire des ondes lumineuses (370 à 750 THz)
PHz	10^{15} cycles/s	Fréquence ondulatoire des rayons X (> 30 PHz)

Mesures de capacité						
k	kilo	10^3	ko	2^{10}	1'024	Taille d'un petit fichier ASCII
M	méga	10^6	Mo	2^{20}	1'048'576	Taille d'un fichier MP3 (musique compressée)
G	giga	10^9	Go	2^{30}	1'073'741'824	Taille d'une heure de film au format MPEG2 (télévision numérique)
T	téra	10^{12}	To	2^{40}	1'099'511'627'776	Taille des informations stockées dans une petite université
P	péta	10^{15}	Po	2^{50}	1'125'899'906'842'624	Taille des informations stockées par une société spécialisée dans le stockage

■ Entrées/Sorties numériques et analogiques

Sur les microcontrôleurs, on trouve des interfaces d'entrées/sorties que l'on peut connecter presque (modulo des **boîtiers de conditionnement** et des relais en puissance) directement à du matériel d'acquisition ou commande (capteurs ou actionneurs). On trouve le même type d'interface pour les ordinateurs : elles se présentent le plus souvent sous la forme de cartes d'extension (notamment au format ISA, PCI ou PCMCIA) nommées **cartes d'acquisition**. Deux types de signaux peuvent être manipulés par ce genre d'interface : les **signaux numériques** et les **signaux analogiques**.

© Dunod – La photocopie non autorisée est un délit.

☐ Entrées/sorties numériques

Les entrées/sorties numériques sont de type tout ou rien (2 états possibles qui correspondent électriquement par exemple à – 5 V, + 5 V). Ce type d'entrée peut être branché (via une adaptation de puissance et ou tension) à des capteurs ou actionneurs de type tout ou rien (**TOR**) (électrovanne TOR, capteur de présence, alimentation…). On appelle **ligne** une entrée ou sortie numérique (physiquement correspondant à deux connecteurs : la masse numérique et la porteuse). L'état d'une ligne est caractérisé par un bit (0 ou 1). Comme la plus petite entité adressable en mémoire est l'octet, les lignes sont regroupées par 8, formant ainsi un octet appelé **port numérique**. Il est possible de définir des ports de taille multiple de 8 (ports de 16 ou 32 bits par exemple). Chaque ligne d'un port peut être configurée de façon logicielle en entrée (lecture sur un capteur) ou en sortie (écriture vers un actionneur).

Généralement, l'utilisation d'entrées/sorties numériques se fait de la façon suivante (figure 4.9) :

– Configuration du port numérique à l'aide d'un octet de configuration de **direction des lignes**. Le plus souvent, le bit *i* (*i* = 0..*taille du port*-1) permet de configurer la ligne *i* (i = 0..*taille du port*-1). Par exemple, en fonction de la carte d'acquisition ou du microcontrôleur, un bit à 1 peut correspondre à mettre la ligne correspondante en sortie, alors qu'un bit à 0 configure la ligne en entrée. Dans ce cas, un octet de configuration valant 0x73 (b01110011) configurerait les lignes 6,5,4,1,0 en sortie et les lignes 7,3,2 en entrée. L'octet de configuration 0x0 placerait tout le port en entrée, alors que 0xFF (ou bien -1 grâce à la représentation en complément à 2 de ce nombre, qui n'est constituée que de 1 en binaire) configurerait le port en sortie. La configuration a généralement lieu une seule fois en début de programme.

– Lecture d'un port : les lignes se lisent toujours par port (d'au moins un octet donc). Par exemple, si le port est d'un octet, la lecture renvoie un octet, dont la représentation binaire correspond à l'état des lignes. Ainsi, si on lit l'octet 0xA3 (b10100011) sur un port préalablement configuré en entrée, cela signifie que les lignes 7,5,1,0 sont à 1 et que les lignes 6,4,3,2 sont à 0. Si le port n'est pas totalement configuré en entrée, les bits lus correspondant aux lignes configurées en sorties ont des valeurs non significatives.

– Écriture sur un port : de même que les lectures, les écritures concernent toujours un port entier. Il existe cependant une différence à noter. En effet, supposons qu'un port, configuré totalement en sortie, ait toutes les lignes reliées à des actionneurs. La sortie est maintenue de façon matérielle à une certaine valeur (1 ou 0), qui est la dernière valeur écrite. Supposons alors que l'on veuille modifier l'état d'un seul actionneur, donc d'une seule ligne. Il faut alors connaître l'état complet du port, modifier le bit correspondant à l'actionneur, puis écrire cet octet sur le port de sortie. Ce fonctionnement, bien que possible, n'est pas très pratique : en effet, les applications de contrôle-commande sont souvent multitâches, et il arrive fréquemment que le composant logiciel commandant un actionneur ne sache pas du tout dans quel état se trouvent les autres actionneurs. L'idée généralement retenue pour éviter d'avoir à s'occuper de ce problème est d'utiliser un **masque**. Le principe est assez proche du ou des octets de configuration de port. Supposons

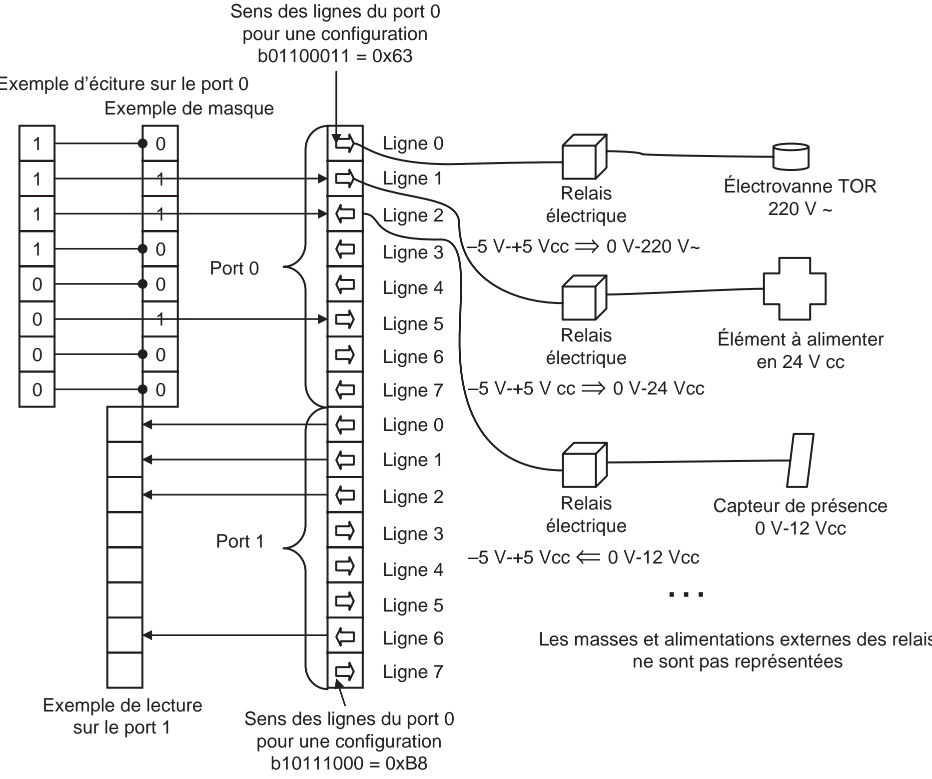


Figure 4.9 – Lignes et ports numériques.

que le port soit d'un octet, on crée un octet masque : pour chaque bit à 1, on considère qu'il y a un trou dans le masque et que les bits correspondants à un trou dans le masque ont une action physique sur le port. Si le bit est à 0, on considère que le masque est opaque et que la ligne correspondante ne peut pas être affectée. L'écriture sur un port se base donc sur deux paramètres, chacun de la même largeur que le port : un masque, et une commande. Tout se passe comme si seuls les bits de la commande correspondant à un trou (bit à 1) dans le masque pouvaient arriver jusqu'aux lignes de sortie numérique. Par exemple (figure 4.10), si un port d'un octet est configuré en écriture, et si l'on veut mettre à 1 la ligne numéro 6 sans toucher aux autres lignes, il suffit de créer un masque valant 0x40 (1 décalé de 6 à gauche), et d'appliquer une commande valant 0x40, ou même 0xFF (octet composé de 8 bits à 1). Grâce au masque, seule la ligne 6 est affectée, et mise à 1. Si l'on souhaite mettre la ligne 6 à 0, on utilise le même masque (0x40), et on applique la valeur 0, seule la ligne 6 est affectée et mise à 0.

Le dispositif de masque est généralement purement logiciel. En fonction du niveau de programmation disponible : soit via une bibliothèque fournie, soit par adressage direct à la main, il peut être nécessaire d'implémenter soi-même la technique du masque.

© Dunod – La photocopie non autorisée est un délit.

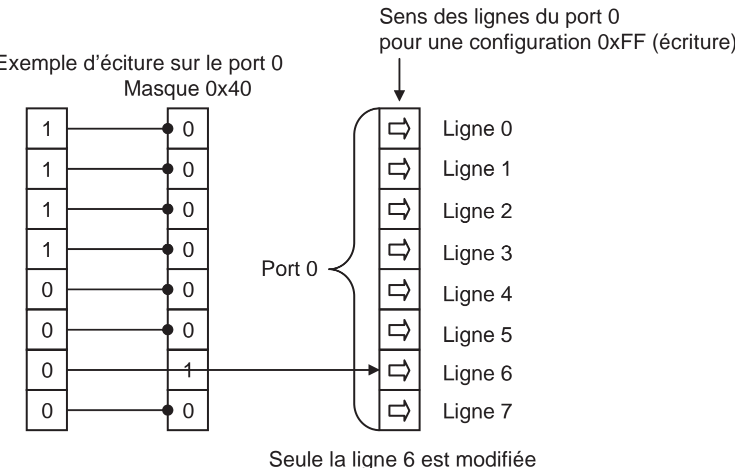


Figure 4.10 – Exemple de masque binaire numérique.

Il est à noter que le fait d'écrire sur une ligne configurée en entrée n'a généralement aucun effet, et que le fait de lire sur une ligne configurée en sortie donne des valeurs non significatives.

La communication logicielle avec les ports d'entrées/sorties s'effectue au plus bas niveau par utilisation des registres de la carte d'acquisition. Une carte d'acquisition plaque ses registres sur des adresses spécifiques dites d'entrées/sorties. Ces adresses sont caractérisées par une adresse de base, et une taille des registres plaqués sur la mémoire. Par exemple si une carte d'acquisition utilise les adresses 0x210 à 0x213, cela signifie que son adresse de base est 0x210, et qu'elle plaque 4 octets de ses registres à partir de l'adresse da base. Ainsi, il est possible que le registre plaqué sur l'adresse 0x210 corresponde aux 8 lignes du port 0, 0x211 aux 8 lignes du port 1, 0x212 aux 8 lignes du port 2, et 0x213 à un octet de configuration de la carte. Le fait d'écrire sur un de ces octets a pour effet de modifier l'état de la carte. Le fait de lire un de ces octets a pour effet d'interroger la carte. Ces actions sont suspensives pour un programme : dans ce cas, qu'il cherche à lire ou écrire sur une adresse d'entrées/sorties, il peut être suspendu jusqu'à la fin de l'entrée/sortie. La lecture n'est pas bloquante : cela signifie que même si aucune valeur n'a changé depuis la dernière lecture sur la carte, la lecture a lieu. L'utilisation basique qui est faite des entrées numériques est donc de type scrutation : il est nécessaire de lire les entrées en boucle.

Il existe cependant des types d'entrées capables de générer une interruption sur **front montant** ou **descendant** (changement d'état 0 à 1 ou 1 à 0) : cela permet de faire des lectures bloquantes, c'est-à-dire de ne pas consommer de temps processeur hors traitement des événements qui changent l'état des entrées. Il en est ainsi des entrées dites de type ***trigger***, des entrées de type **compteur**, ou bien entrées supportant le ***pattern matching*** (capables de générer une interruption sur une différence d'entrée).

Les entrées de type *trigger* sont généralement utilisées pour synchroniser les acquisitions sur une source externe, par exemple un *trigger* (qui se traduit par un signal carré, soit passage de 0 à 1 puis passage de 1 à 0) périodique, ou bien un *trigger* déclenché lorsqu'un événement extérieur survient.

Les entrées/sorties de type compteur permettent de compter des fronts (montants et ou descendants) de façon bloquante (utilisation des interruptions).

Les entrées supportant le *pattern matching* (permettant de déclencher une interruption lorsque les entrées changent), se trouvent sur du matériel assez coûteux. Elles sont utilisées lorsque le temps de prise en compte des événements extérieurs doit être très court. En effet, l'un des inconvénients du *polling* est qu'il est difficile de garantir la prise en compte d'un événement en dessous d'une période de scrutation, au minimum de l'ordre de la milliseconde.

☐ Entrées/sorties analogiques

Lorsqu'un système doit lire des valeurs externes pouvant varier dans le domaine continu (thermocouple, capteur de position axiale…) ou bien pouvoir commander des actionneurs comme des électrovannes réglables ou bien l'accélération variable d'un moteur, il lui faut utiliser des entrées ou sorties analogiques. Les **entrées/sorties analogiques** permettent de passer de la continuité du monde réel à l'univers discret des systèmes informatisés et réciproquement.

Une entrée analogique permet de brancher un élément externe pouvant délivrer une tension (généralement, les entrées analogiques permettent de lire des tensions de l'ordre de 0-10 V ou – 10 V-+ 10 V). Un circuit spécifique, nommé **Convertisseur Analogique Numérique** (CAN) est utilisé pour convertir la tension d'entrée en valeur numérique. La précision de cette valeur numérique dépend de plusieurs facteurs :

– la **résolution**, c'est-à-dire le nombre de bits utilisés pour représenter la tension lue sous forme numérique (en binaire). Pour n bits de résolutions, 2^n valeurs de tension différentes peuvent être fournies par une entrée analogique. Si le domaine est – 10 V-+ 10 V, alors 2^n valeurs sont utilisées pour représenter une gamme de 20 V. La finesse de la lecture est alors, en plus de la précision physique de la carte, de 20 V/2^n. Typiquement, la résolution est de l'ordre de 12, 16, ou 24 bits. Avec une résolution de 12 bits, deux valeurs successives représentables ont un écart de 4,88 mV. En 16 bits, on obtient 300 µV alors qu'en 24 bits, deux valeurs successives sont séparées de 1 µV. Généralement, il est possible d'ajuster de façon logicielle le domaine d'entrée, ainsi, si l'on sait que le signal lu se situe entre 0 et 5 V, la précision de la conversion sera multipliée par 4 par rapport à – 10 V-+ 10 V (car deux valeurs successives sont séparées de 5 V/2^n) ;

– la **gamme**, caractérisé par la tension minimale et maximale pouvant être lue, typiquement – 10 V-+ 10 V ;

– le **gain** permet d'amplifier le signal d'entrée : avec un gain de 10 et une gamme de 0-10V, le domaine de lecture peut se situer entre 0 et 1 V, ce qui décuple la précision de la lecture par rapport à un gain de 1 ;

– le **bruit**, pouvant provenir de différents facteurs extérieurs.

© Dunod – La photocopie non autorisée est un délit.

Les entrées analogiques sont caractérisées aussi par la fréquence d'échantillonnage du convertisseur analogique numérique. En effet, celui-ci est partagé entre un certain nombre d'entrées analogiques (par exemple 2, 4, 8, 16…). Les entrées analogiques sont donc multiplexées sur un convertisseur qui a une certaine fréquence de conversion. Si l'on souhaite utiliser plusieurs entrées analogiques partageant le même convertisseur, la fréquence maximale d'échantillonnage est donnée par *fréquence d'échantillonnage du convertisseur analogique numérique / nombre d'entrées*. On trouve aisément des entrées analogiques permettant d'échantillonner un signal à quelques centaines de milliers d'échantillons par seconde. Le matériel permettant d'atteindre des fréquences de l'ordre de quelques méga-échantillons par seconde est plus coûteux. Certaines entrées analogiques peuvent être programmées pour déclencher une interruption sur certains seuils de tension.
Malheureusement, beaucoup de capteurs (thermocouples par exemple) délivrent un courant trop faible pour être lu de façon fiable directement par des entrées analogiques. On utilise donc souvent du matériel de **conditionnement**, dont le rôle est de filtrer, amplifier ou réduire la tension en entrée, et/ou isoler du courant d'entrée qui le plus souvent ne doit pas excéder 20 mA.
De façon symétrique au convertisseur analogique numérique, le **Convertisseur Numérique Analogique** (CNA) permet à une sortie analogique de prendre une valeur numérique en entrée, et de restituer en sortie une tension analogique. Comme les sorties numériques, les sorties analogiques maintiennent une tension tant qu'on ne change pas leur valeur d'entrée (si l'on applique 4,33 V en sortie, cette tension est maintenue jusqu'à ce que l'on décide d'appliquer une autre tension). Par conséquent, un convertisseur est utilisé pour chaque sortie, et le prix des sorties analogiques est fortement influencé par le nombre de sorties (contrairement aux entrées qui partagent le même convertisseur analogique numérique).
Une sortie analogique est caractérisée par :

– le **délai** de changement de valeur ;
– la fréquence de commande, fréquence maximale de changement de valeur ;
– la résolution, dont le principe est similaire aux entrées.

À cause du fonctionnement discret des systèmes informatisés, il faut avoir conscience qu'on ne peut générer de façon parfaite un signal, par exemple un sinus : il est possible de générer des paliers successifs qui, de loin, forment un sinus.
Du matériel de conditionnement (notamment en tension ou en courant, les sorties analogiques étant souvent limitées entre 4 et 20 mA) est souvent utilisé entre les sorties analogiques et les actionneurs commandés.
La figure 4.11 présente un schéma simplifié de carte d'acquisition.

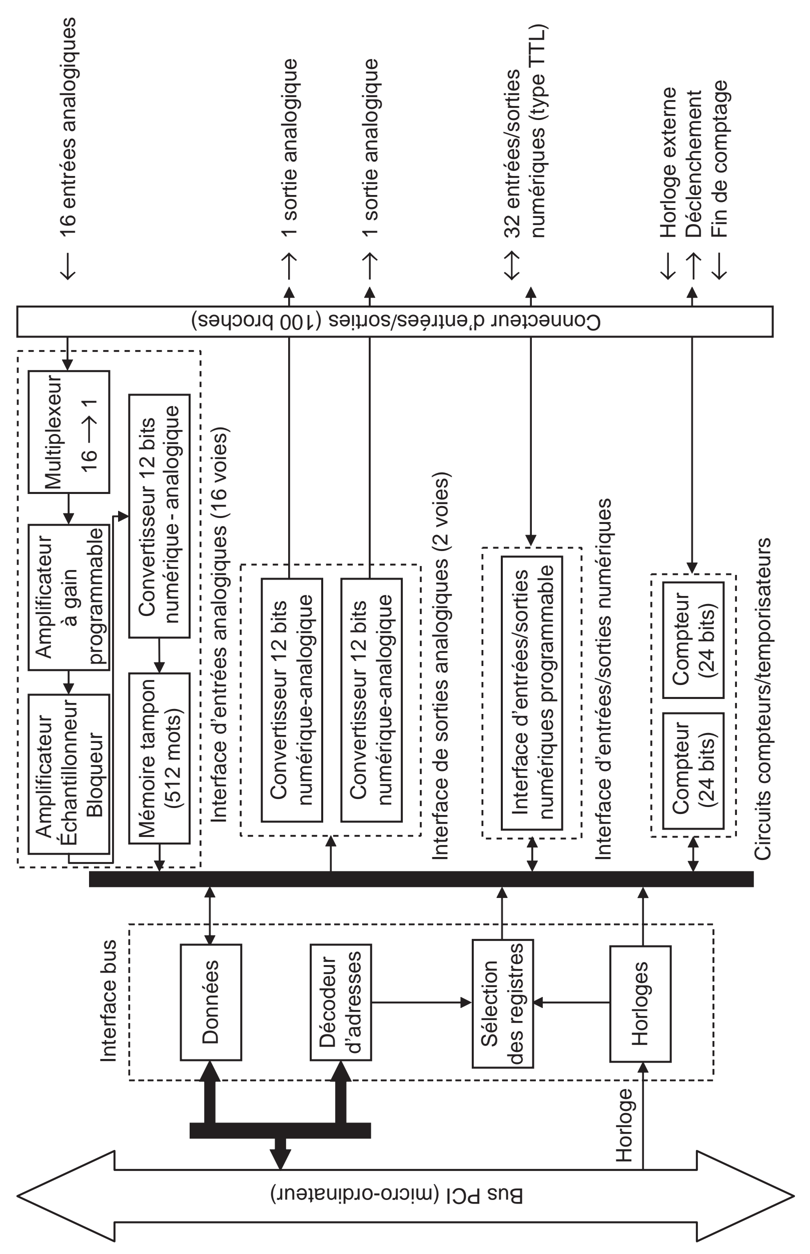


Figure 4.11 – Schéma simplifié de carte d'acquisition.

© Dunod – La photocopie non autorisée est un délit.

4.2 Architecture logicielle

Sans le système d'exploitation (SE, *Operating System*, ***OS***), le système informatisé n'est qu'une boîte inutilisable. Le système d'exploitation fait l'interface entre le matériel et le logiciel (figure 4.12) : il présente aux programmes une machine virtuelle, relativement indépendante du matériel sous-jacent.

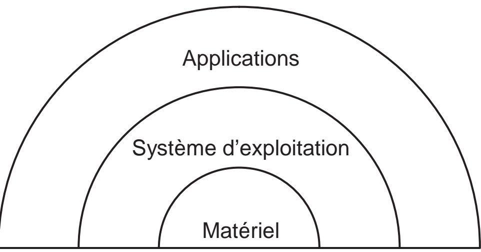


Figure 4.12 – Rôle du système d'exploitation.

Ce chapitre n'a pas pour objectif de présenter de façon exhaustive les rôles d'un système d'exploitation. Il s'attache à décrire quelques éléments indispensables pour la bonne compréhension des problèmes liés au multitâche. Il présente donc la notion de processus, d'ordonnancement, et de synchronisation de processus.

4.2.1 Processus

L'un des rôles primordiaux des systèmes d'exploitation multitâches (c'est le cas aujourd'hui de la quasi-totalité des systèmes d'exploitation) est d'assurer l'exécution de plusieurs programmes en parallèle.

■ Caractérisation des processus

Un programme en cours d'exécution s'appelle un **processus**. Un processus est une instance de programme (il peut y avoir plusieurs processus d'un même programme, par exemple, plusieurs processus du même traitement de texte). À chaque fois qu'un programme est exécuté, un processus est créé. Il se voit attribuer de la mémoire, tout ou partie de son code et de ses données est chargé en mémoire centrale, et il est caractérisé par le système d'exploitation grâce à un **Bloc de Contrôle de Processus** (BCP) contenant diverses informations (numéro d'identification, mémoire allouée, fichiers ouverts, temps d'exécution, état…). Les états d'exécution d'un processus sont représentés sur la figure 4.13.

Un processus en exécution s'exécute séquentiellement sur le processeur. Lorsqu'il fait une instruction bloquante (une entrée/sortie par exemple), il se retrouve ***bloqué*** et ne peut plus utiliser le processeur jusqu'à ce que l'événement attendu ait lieu. Dans ce cas, il se retrouve dans l'état ***prêt*** (cela signifie qu'il attend de pouvoir s'exécuter sur le processeur). Lorsque le système d'exploitation le décide, un processus en exécution peut être **préempté**, c'est-à-dire passer de l'état ***exécuté*** à l'état *prêt*. Remarquons qu'il existe d'autres états possibles, comme l'état ***suspendu***, dans lequel

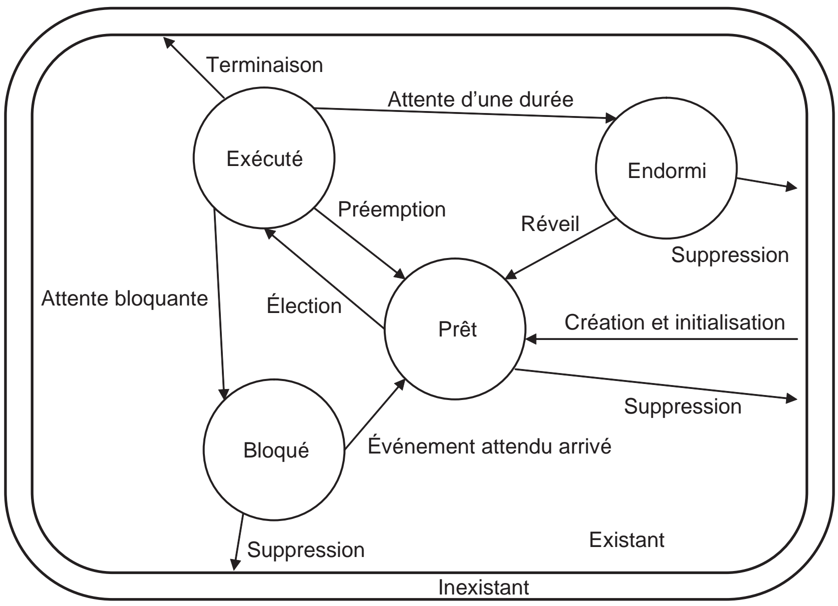


Figure 4.13 – États d'un processus.

on peut plonger un processus, jusqu'à sa reprise qui le remet dans l'état *prêt*, ainsi que l'état *endormi*, dans lequel un processus peut se trouver lorsqu'il décide d'attendre pendant un certain temps.

Techniquement, la préemption se déroule de la façon suivante (figure 4.14) : une interruption particulière, l'**horloge temps réel** (**HTR**), est utilisée par le système d'exploitation. Périodiquement (la durée d'une période s'appelle un **quantum**), l'interruption horloge a lieu. L'ISR qui a lieu fait partie du système d'exploitation : elle sauvegarde alors dans le bloc de contrôle de processus l'état d'exécution du processus (l'état des registres), puis restaure dans le processeur l'état des registres de l'un des processus prêt que le système d'exploitation choisit. Ce qui caractérise un processus en exécution s'appelle un **contexte**. La préemption consiste donc en un changement de contexte. La commutation de contexte est réalisée par une routine spécifique du système d'exploitation appelée ***dispatcher***. Souvent, dans les microprocesseurs, le *dispatcher* est implémenté de façon matérielle.

À chaque quantum de temps, le système d'exploitation prend donc la main grâce à une ISR, interrompant ainsi le processus exécuté, et peut décider de le préempter, c'est-à-dire d'**élire** un autre processus. Dans les systèmes d'exploitation, l'ordre de grandeurs du quantum de temps est de l'ordre de 10 à 100 ms, alors que dans les systèmes temps réel, on peut arriver à 1 ms, voire même en dessous.

Il faut remarquer que le temps d'exécution du système d'exploitation et du *dispatcher* peut ne pas être négligeable au regard du quantum de temps choisi. Le pourcentage du temps processeur utilisé par le système d'exploitation pour gérer les processus s'appelle le **surcoût processeur** ou ***overhead***. Sur la figure 4.14, cet *overhead* est visible sur la ligne SE du diagramme de Gantt. Typiquement, sur un système d'exploi-

© Dunod – La photocopie non autorisée est un délit.

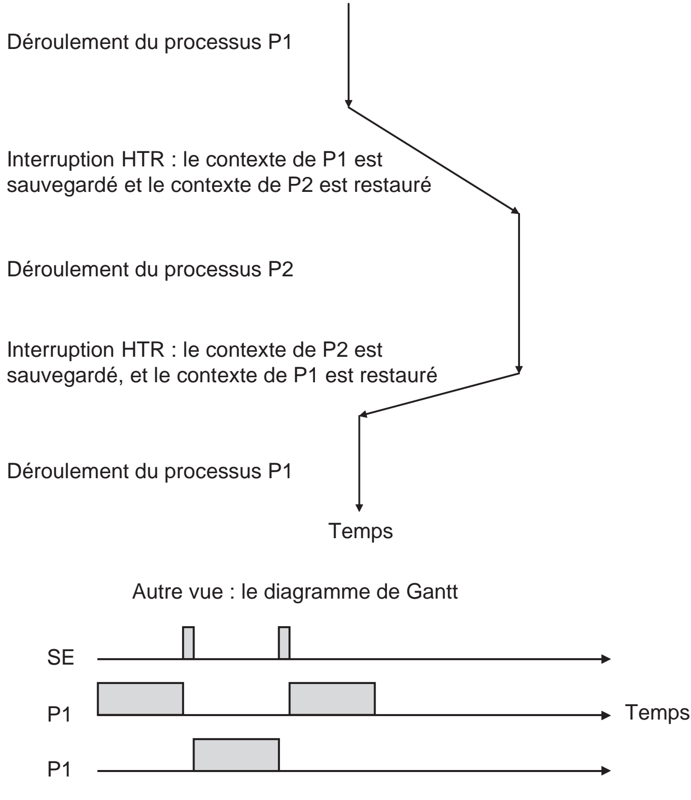


Figure 4.14 – Principe de l'entrelacement temporel.

tation classique, il y a plusieurs centaines d'interruptions par seconde (gestion de l'horloge, du réseau, des périphériques et bus d'entrées/sorties…) pour un *overhead* inférieur à 1 % du temps processeur.

■ Ordonnancement de processus

La stratégie utilisée pour choisir parmi les processus prêts le prochain processus à exécuter s'appelle l'**ordonnancement.** L'ordonnancement consiste simplement à ordonner les processus par **priorité**. Le *dispatcher* se charge de placer le contexte du processus le plus prioritaire sur le processeur. Il existe plusieurs stratégies d'ordonnancement spécifiques pour le temps réel qui seront abordées par la suite. Dans ce chapitre, nous présentons quelques stratégies de bases, utilisées par les systèmes d'exploitation.

Grâce à la préemption et à l'ordonnancement, tout se passe comme si les processus pouvaient s'exécuter en parallèle. Dans la réalité, le système d'exploitation réalise un entrelacement temporel des processus.

Notons que les exemples d'ordonnancement présentés ci-après sont très académiques, puisque les processus sont censés ne pas s'endormir, se suspendre, ou même effectuer d'actions suspensives ou bloquantes. En effet, lorsqu'un processus exécute une instruc-

tion suspensive ou bloquante, cette instruction fait un appel au système d'exploitation, qui peut alors changer l'état du processus et élire un autre processus prêt.

Il est important de retenir que seuls les processus prêts concourent pour l'obtention du processeur. Nous citons ci-après quelques politiques d'ordonnancement répandues :

– **FIFO** (***First In First Out***) : premier arrivé premier servi, l'un des algorithmes les plus simples, puisqu'il exécute les processus dans leur ordre d'arrivée. Dans le cas où les processus ne se suspendent pas et n'attendent pas, cet algorithme ne nécessite aucune préemption, car lorsqu'un processus commence son exécution, il n'est pas interrompu jusqu'à sa terminaison.

– **Algorithme à priorités** : chaque processus est muni d'une priorité. À chaque quantum, c'est le processus prêt de plus forte priorité qui est élu. En cas d'égalité, d'autres règles peuvent s'appliquer, comme FIFO par exemple.

– **Algorithme du tourniquet** (***round robin***) : les processus prêts ont droit chacun leur tour à un quantum. Lorsqu'un cycle d'attribution du processeur aux processus est terminé, un autre cycle du tourniquet commence.

On peut remarquer sur la figure 4.15 que la définition de l'algorithme du tourniquet laisse libre de gérer le réveil des processus de différentes manières : ils peuvent être placés en début de tourniquet (comme sur la figure), ou en fin de tourniquet. Notons que sur la figure 4.15, le surcoût processeur est négligé.

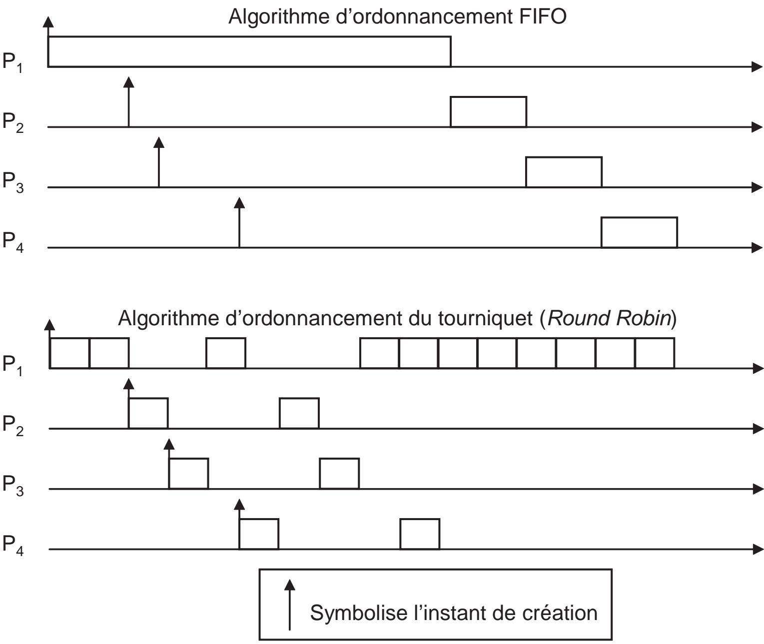


Figure 4.15 – Illustration des algorithmes FIFO et tourniquet.

© Dunod – La photocopie non autorisée est un délit.

Les critères d'évaluation des algorithmes d'ordonnancement utilisés dans les systèmes d'exploitation classiques sont les suivants :

– l'**équité** caractérise le fait que le processeur est réparti de façon équitable entre les processus. Ainsi, l'algorithme d'ordonnancement FIFO n'est pas équitable, car si un processus a une durée très longue, les processus lancés après devront attendre longtemps avant de pouvoir s'exécuter. Si l'on se place dans le contexte d'un système d'exploitation classique, cela signifie que l'utilisateur ne peut pas exécuter de nouveau processus tant qu'un processus n'est pas terminé. Au contraire, le tourniquet paraît très équitable puisqu'il partage le processeur de façon égale entre les processus. Enfin, les algorithmes à priorité ne sont pas équitables pour les processus peu prioritaires au regard des processus prioritaires ;

– le **temps d'attente moyen/maximal** représente la durée moyenne/maximale pendant laquelle un processus reste dans l'état *prêt* (en attente du processeur). Il est évident que FIFO et les algorithmes à priorité n'étant pas équitables, ils ne proposeront pas un temps d'attente maximal très intéressant pour un grand nombre de cas (il est bien sûr possible de générer des cas particuliers en choisissant les dates de réveil et les durées des processus de sorte à observer un bon temps d'attente moyen ou maximal). Pour le tourniquet, le temps d'attente maximal d'un processus est borné : si l'on nomme q la durée d'un quantum, n le nombre maximal de processus prêts à un instant, et d la durée d'une préemption par le système d'exploitation, le pire temps d'attente d'un processus est $(n-1)q + n \cdot d$ (figure 4.16). Pour les algorithmes FIFO et à priorités, le temps d'attente de chaque tâche dépend des arrivées et/ou priorités des autres tâches ;

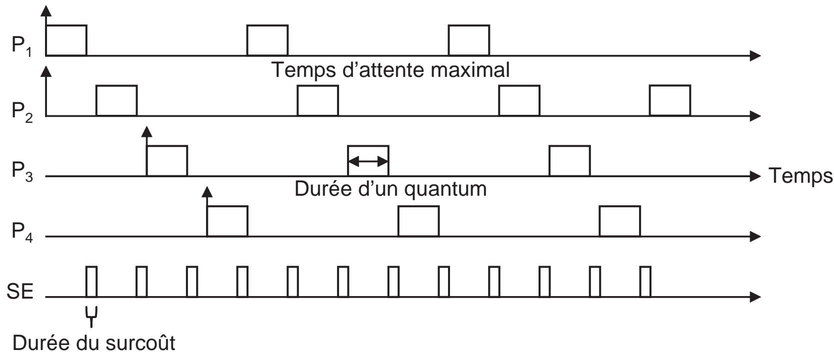


Figure 4.16 – Illustration du temps d'attente.

– le **temps de réponse** d'un processus est la durée séparant sa création de sa terminaison. Ce critère est très important pour les applications de contrôle-commande. Ce critère peut être généralisé à un ensemble de processus : dans ce cas, on parle de temps de réponse moyen (moyenne des temps de réponse des processus) ou maximal (plus grand temps de réponse parmi les processus). Généralement, le tourniquet n'est pas un très bon candidat vis-à-vis du temps de réponse moyen et maximal par rapport à FIFO (figure 4.17) car pour diminuer le temps d'attente

de chaque processus, il retarde un peu chacun d'entre eux. Les algorithmes à priorités vont bien entendu, sauf pour quelques cas particuliers que l'on peut construire, diminuer le temps de réponse des processus prioritaires, mais augmenter celui des processus moins prioritaires. Le moyen de diminuer le temps de réponse moyen consisterait à exécuter d'abord les processus ayant la durée restant à exécuter la plus courte (algorithme nommé SRPT pour *Shortest Remaining Time First*). Cependant, cela supposerait de connaître *a priori* la durée des processus (algorithme d'**ordonnancement clairvoyant**), ce qui n'est pas réaliste pour un système d'exploitation généraliste (nous verrons cependant au chapitre 8 que c'est réaliste, et même réalisé pour certains systèmes temps réel).

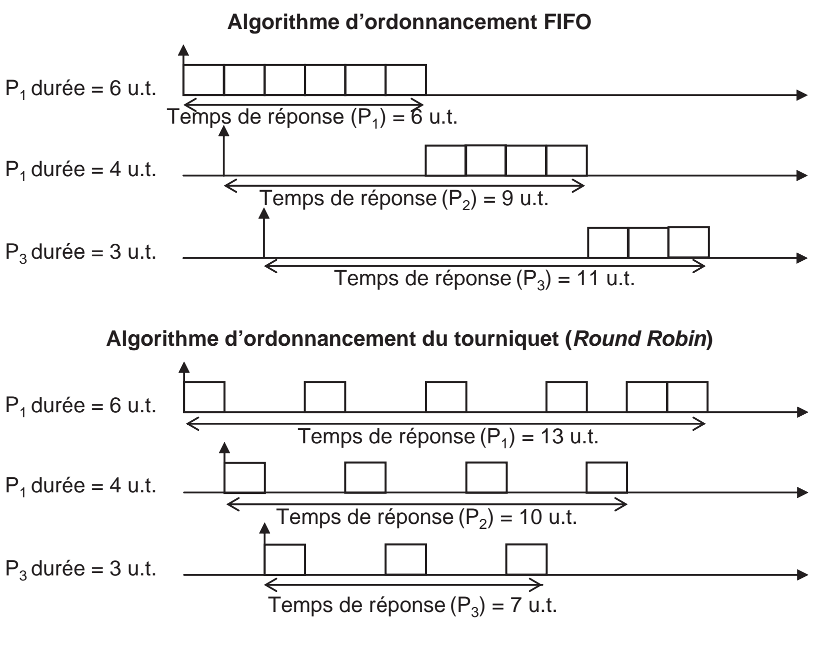


Figure 4.17 – Illustration du temps de réponse pour 3 processus.

Pour caractériser un algorithme d'ordonnancement, on peut aussi utiliser la notion de **rendement** (nombre de processus terminés par quantum de temps), la durée du quantum lui-même ou le nombre de préemptions, qui vont influencer l'*overhead*, etc.
Dans les systèmes d'exploitation généralistes, la politique d'ordonnancement la plus utilisée est basée sur une combinaison du tourniquet et des priorités, qui allient l'équité du tourniquet au rendement des algorithmes à priorités pour les processus jugés prioritaires. Cet algorithme d'ordonnancement s'appelle MLF (*Multi-Level Feedback*) et consiste à utiliser un tourniquet par niveau de priorité, et à partager le processeur suivant la règle du tourniquet entre les processus prêts de plus forte priorité. Lorsqu'un processus de faible priorité a attendu pendant un certain temps,

© Dunod – La photocopie non autorisée est un délit.

sa priorité augmente temporairement (il change de tourniquet) jusqu'à ce qu'il obtienne un quantum de temps processeur. Sa priorité redescend alors à son niveau initial. La priorité des processus est alors dynamique.

4.2.2 La gestion de la concurrence

■ Introduction

Des processus doivent pouvoir partager des ressources (matériel, variables, etc.) et s'échanger des données. Par conséquent, le système d'exploitation permet la communication et la synchronisation de processus. Pour cela il offre des primitives de communication et de synchronisation, souvent conformes ou proches de la norme POSIX (voir chapitre 6).

Le fait que les processus puissent s'exécuter en parallèle entraîne des problèmes appelés **problèmes de la concurrence**. Considérons pour illustrer cela un exemple : l'accès concurrent à une variable partagée par deux processus.

Deux processus (figure 4.18) partagent une variable *x*. Leur action est d'incrémenter *x*, puis de se terminer. Ils ont le même code très simple, que l'on peut exprimer par `x := x+1`.

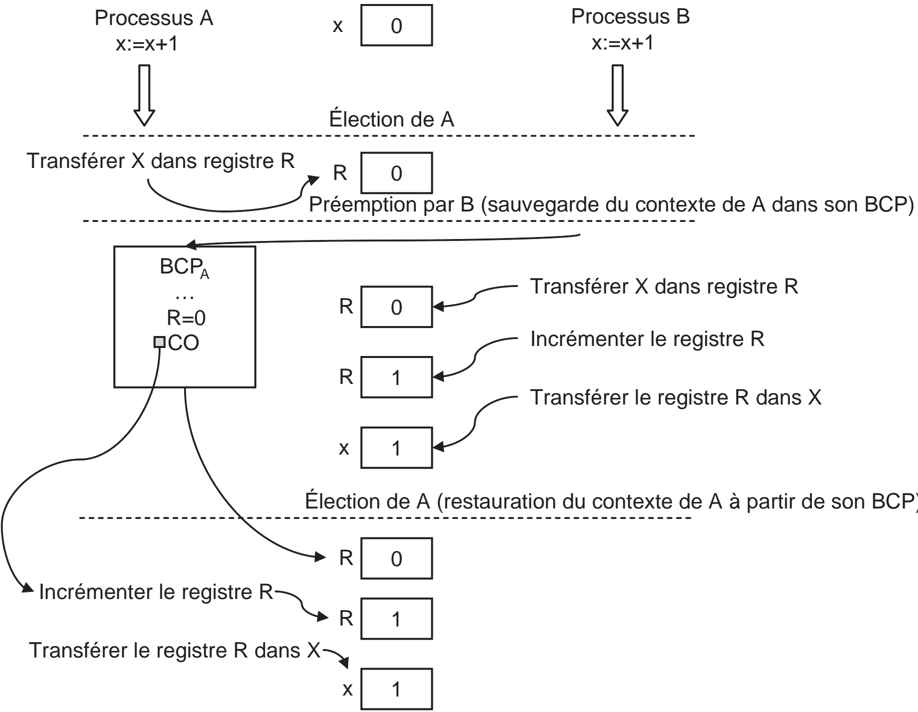


Figure 4.18 – Problème d'exclusion mutuelle.

Initialement, la variable x vaut 0. Lorsque les deux processus ont terminé leur exécution, sa valeur peut être 1 ou 2 (cela est déterminé uniquement par les préemptions). On imagine facilement comment le résultat final de x peut valoir 2, voyons comment il est possible d'obtenir 1. L'incrémentation d'une variable se décompose au minimum en trois instructions : copie de la variable dans un registre du processeur, modification du registre, puis recopie du registre à l'adresse mémoire de la variable. Supposons que le processus P_1 ait commencé son exécution : il charge x dans un registre du processeur, sa valeur est donc 0. Il est alors préempté par P_2, les registres de P_1, y compris le registre contenant la valeur 0, sont sauvegardés dans le contexte de P_1. P_2 s'exécute alors entièrement, c'est-à-dire qu'il copie la valeur de x (donc 0) dans un registre, incrémente le registre, et termine après avoir copié la valeur 1 du registre à l'adresse de x. P_1 peut alors poursuivre son exécution. Ses registres sont restaurés : le registre utilisé pour le calcul vaut donc 0, et le compteur ordinal correspond à l'instruction d'incrémentation du registre. Le registre passe donc à 1, valeur qui est copiée à l'adresse de x.

Dans cet exemple, la variable x est appelée une **ressource critique** : élément partagé par plusieurs processus. Afin de garantir que le résultat vaut toujours 2, il faut garantir que les processus ne peuvent pas s'interrompre mutuellement entre la lecture de x et son écriture (ce qui n'empêche pas un autre processus n'utilisant pas x de les interrompre). La portion de code allant de la lecture à l'écriture de x s'appelle une **section critique**. Le fait de garantir que les sections critiques de deux processus accédant à la même ressource critique ne se préemptent pas s'appelle garantir le respect de l'**exclusion mutuelle**. On dit que les deux sections critiques doivent être en exclusion mutuelle. Ainsi, si l'on respecte l'exclusion mutuelle entre les sections critiques de l'exemple de la figure 4.18, la valeur finale de x est forcément 2.

■ Exclusion mutuelle

Une possibilité de garantir l'exclusion mutuelle est de masquer les interruptions (empêcher le processeur de gérer les interruptions) : l'interruption horloge temps réel n'étant plus traitée, il n'y a alors plus de préemption. Cependant, cette solution n'est pas satisfaisante : aucun autre processus (pas même le système d'exploitation) n'a accès au processeur pendant la durée d'une section critique. Cela signifie qu'en cas de non terminaison d'un processus (code erroné, calcul très long sans aucune entrée/sortie), le réamorçage du système est obligatoire.

L'une des idées candides que l'on peut avoir consiste à utiliser une variable signifiant si la ressource est libre : par exemple 1 pour libre, 0 pour pris. Le code des processus voulant entrer en section critique serait alors :

```
Tant que libre≠1 attendre
libre :=0
-- Entrée en section critique
```

Le problème n'est que repoussé : en effet, un processus peut être interrompu par un autre entre le moment où il a constaté que Libre était à 1 et le moment où il le met à 0. On peut alors se retrouver avec plusieurs processus en section critique, car un second processus peut lui aussi penser que la ressource est libre. C'est exactement le même type de problème que sur la figure 4.18. Il n'y a aucune solution purement

© Dunod – La photocopie non autorisée est un délit.

logicielle, et il est nécessaire d'utiliser le matériel par masquage des interruptions, mais seulement pendant une durée très courte avec des fonctionnalités testées préalablement.

☐ Le sémaphore

L'un des outils logiciels (utilisant le matériel) les plus utilisés est le **sémaphore**. Un sémaphore peut être vu comme un verrou, qu'il est possible de « **prendre** » et de rendre (on dit « **vendre** »). Prendre un sémaphore est une action bloquante : si le sémaphore est libre, il devient pris, par contre s'il est déjà pris, le processus voulant le prendre passe dans l'état bloqué jusqu'à ce qu'il lui soit possible de prendre le sémaphore. Techniquement, le fait de tenter de prendre un sémaphore qui n'est pas libre a pour effet de passer le processus dans l'état bloqué, et de placer le numéro du processus dans la file d'attente du sémaphore. Un sémaphore est donc composé d'un entier et d'une file d'attente. Les actions *prendre* et *vendre* peuvent se dérouler de la façon suivante (cas d'un **sémaphore binaire**, c'est-à-dire à deux états) :

```
Un sémaphore S consiste en un entier S.val, et une file d'attente
d'identificateurs de processus S.file.
Procédure Prendre(S: in out sémaphore)
Début
Masquer les interruptions
-- La procédure prendre ne doit pas être interrompue
Si S.val=0 alors
           Ajouter le n° de processus à S.file
           Démasquer les interruptions
           Passer dans l'état bloqué et appeler le dispatcher
Sinon
           S.val ←0
           Démasquer les interruptions
FinSi
Fin
Procédure Vendre(S:in out sémaphore)
Début
Masquer les interruptions
Si il existe un processus dans S.file alors
Enlever un processus de la file
Mettre ce processus dans l'état prêt
Sinon
S.val←1
FinSi
Démasquer les interruptions
Appel au dispatcher
Fin
```

À l'intérieur des primitives, les interruptions sont masquées afin de les rendre non préemptibles : ces primitives sont **atomiques**, dans le sens où elles sont non interruptibles.

Il est important de bien comprendre que ce sont les processus qui exécutent les primitives Prendre et Vendre. Ces primitives, fournies par le système d'exploitation ou présentes dans les langages de programmation, utilisent le masquage des interruptions afin d'exécuter des parties critiques (test de la valeur du sémaphore, mise à jour de la valeur et de la file d'attente). Un processus qui exécute Prendre se bloque si le sémaphore n'est pas libre, un processus qui exécute Vendre réveille lui-

même un processus bloqué. Il n'y a donc pas d'arbitre, et les processus eux-mêmes gèrent grâce à Prendre et Vendre le passage de l'état *prêt* à *bloqué* et de *bloqué* à *prêt*.

Afin d'illustrer le fonctionnement du sémaphore, supposons que trois processus *A*, *B*, et *C* exécutent chacun le code suivant en parallèle :

```
S : Semaphore := créer_sémaphore("mutex_x",1)
-- S est un sémaphore partagé par tous les processus nommé mutex_x
-- Ce sémaphore commun protège les accès à la variable x partagée par
-- les processus
Prendre(S)
x := x+1
Vendre(S)
```

Notons que le terme *mutex* (pour *mutual exclusion*) est souvent utilisé pour dénommer les sémaphores permettant d'assurer l'exclusion mutuelle. Ici, « *mutex_x* » est un sémaphore du système d'exploitation utilisable par tous les processus : le premier processus demandant à créer ce sémaphore crée effectivement le sémaphore, avec une valeur initiale de 1. La fonction `créer_sémaphore` renvoie alors une référence à ce sémaphore, qui est utilisée en paramètre des fonctions `Prendre` et `Vendre`. Après que ce sémaphore ait été créé, il est repéré par son nom dans le système. Les appels suivants à `créer_sémaphore("mutex_x",1)` se contentent alors de renvoyer la référence du sémaphore déjà existant. C'est donc le même sémaphore du système qui sera utilisé par tous les processus utilisant un sémaphore du même nom.

Supposons que le processus *A* débute son exécution : il prend le sémaphore (sa valeur passe donc à 0) et entre en section critique. Le processus *B* est créé et préempte le processus *A* pendant sa section critique. Au moment où le processus *B* tente de prendre le sémaphore, il passe alors dans l'état *bloqué* et son numéro est stocké dans la file d'attente du sémaphore. Supposons que *C* soit alors créé : de la même façon que le processus précédent, il passe dans l'état *bloqué* au moment où il tente de prendre le sémaphore et son numéro est stocké dans la file d'attente du sémaphore. Le processus *A* reprend alors la main. Lorsqu'il sort de sa section critique, il vend le sémaphore. Cela se traduit par le réveil du premier processus en attente (si l'on considère que la file d'attente est gérée en FIFO), qui peut alors entrer en section critique. La valeur du sémaphore ne change pas, puisque cela reviendrait à l'incrémenter puis le décrémenter. Il est important de remarquer que ce processus ne peut pas savoir qu'il s'est bloqué, tout se passe comme s'il s'était endormi en attendant d'entrer en section critique, sans s'en apercevoir. *A* termine alors. Lorsque *B* termine sa section critique, il passe *C* dans l'état *prêt* : celui-ci peut à son tour entrer en section critique. *C*, lorsqu'il vend le sémaphore à la fin de sa section critique, incrémente sa valeur puisqu'il n'y a plus aucun processus en attente.

Que se passerait-il si la ressource critique était telle que deux processus, mais pas plus, pouvaient y accéder en même temps ? Il suffirait simplement d'initialiser le sémaphore à la valeur 2. Nous aurions alors un sémaphore que l'on qualifierait de **sémaphore à compte**, à la différence du **sémaphore binaire** (prenant les valeurs 0 ou 1).

Lorsque l'on utilise un sémaphore à compte, il est possible pour un processus de demander à Prendre ou Vendre plusieurs instances d'un sémaphore, les primitives

© Dunod – La photocopie non autorisée est un délit.

Prendre et Vendre sont alors modifiées pour prendre en compte un paramètre qui est le nombre d'instances concernées.
L'utilisation d'un sémaphore pour garantir l'exclusion mutuelle est donc relativement simple, puisqu'elle consiste à entourer une section critique par la prise et la vente d'un sémaphore utilisé pour protéger l'accès à la ressource. L'avantage est que le respect de l'exclusion mutuelle se fait à coût processeur faible, puisque les processus en attente d'entrée en section critique n'utilisent pas le processeur (passage dans l'état *bloqué*). L'inconvénient de cette approche est le risque d'oublier de protéger l'accès à une ressource par un sémaphore, d'oublier de le vendre, etc. Nous verrons dans la suite que le sémaphore est un outil important, très utilisé en programmation parallèle dans les langages de programmation à base de langage C.
Parfois, il est intéressant de nuancer le type d'accès à une ressource : accès en lecture seule, ou accès en écriture. Par exemple, il n'est pas gênant que des processus faisant un accès en lecture à une base de données se préemptent mutuellement pendant un accès en lecture à la base. Par contre, il faut garantir qu'un processus faisant un accès en écriture (modification d'une ligne d'une table par exemple) soit le seul en section critique : aucun « lecteur » (processus faisant un accès en lecture), et bien sûr aucun autre « écrivain » (processus modifiant la base de données) ne doit pouvoir avoir accès à la base pendant sa modification (pendant qu'un écrivain est en section critique). Ce problème est appelé **problème du lecteur/écrivain**. Sa solution est très simple lorsque le nombre de lecteurs n est connu et borné :

```
Sémaphore_bdd : sémaphore(n)
Processus lecteuri,i=1..n :
Début
Faire toujours
      Prendre(sémaphore_bdd)
      Lire la base de données
      Vendre(sémaphore_bdd)
Fait
Fin
Processus écrivaini,i=1..m :
Début
      Faire toujours
            Prendre(sémaphore_bdd,n)
            -- on prend les n instances du sémaphore
            Modifier la base de données
            Vendre(sémaphore_bdd,n)
      Fait
Fin
```

Au plus, *n* lecteurs peuvent être simultanément en section critique, mais lorsqu'un écrivain est en section critique, aucun autre écrivain et aucun lecteur ne peut y être. On peut se demander comment est gérée la demande de *n* instances d'un sémaphore lorsqu'il n'y a pas suffisamment d'instances disponibles. La valeur du sémaphore tombe alors à 0, et le processus se bloque et il est mis dans la file d'attente, avec en plus l'information du nombre d'instances manquantes à ce processus. Au fur et à mesure que des instances du sémaphore sont vendues, ce nombre d'instances manquantes diminue. Lorsqu'il tombe à 0, le processus est mis dans l'état prêt. Grâce à ce fonctionnement, si la file d'attente est gérée en FIFO, l'accès à la ressource critique base de données aura lieu en FIFO, respectant ainsi l'ordre des demandes (lorsqu'un

écrivain se bloque, comme il prend toutes les instances disponibles du sémaphore, les lecteurs voulant prendre une instance du sémaphore se retrouvent bloqués derrière l'écrivain dans la file d'attente, dans leur ordre d'arrivée).

Le sémaphore a de nombreuses utilisations possibles dans ce qu'on appelle la **synchronisation de processus**. Ainsi, un autre problème classique de synchronisation est le **problème du producteur/consommateur** : un processus produit des données et les stocke dans une zone tampon, de taille bornée (figure 4.19). Un second processus consomme ces données. Le problème est de faire en sorte que le producteur se bloque tant que le tampon est plein, et que le consommateur se bloque lorsque le tampon est vide.

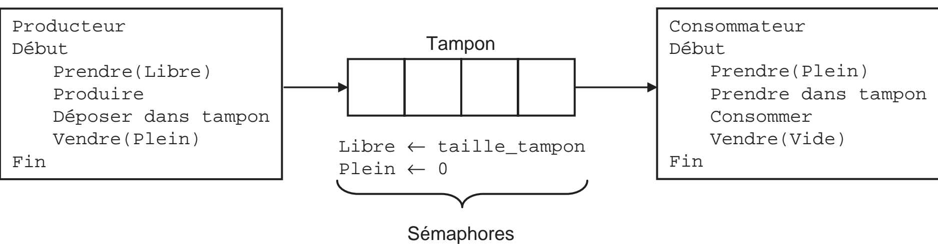


Figure 4.19 – Le problème du producteur/consommateur.

La solution au problème du producteur/consommateur illustre le fait que cela n'est pas forcément le même processus qui prend et qui vend un sémaphore.

Lorsque des processus peuvent utiliser en même temps plusieurs ressources critiques, il peut se produire un phénomène d'**interblocage** (ou ***deadlock***) : un processus détient une ressource, mais a besoin d'une autre ressource pour entrer en section critique, alors que cette ressource est déjà détenue par un autre processus qui attend lui-même une autre ressource, etc. jusqu'à former un cycle. Voici un exemple simple d'interblocage de deux processus :

```
                s1 : sémaphore(1)
                s2 : sémaphore(1)
Processus A :                         Processus B :
Faire toujours                        Faire toujours
      Prendre(s1)                           Prendre(s2)
      Prendre(s2)                           Prendre(s1)
      section critique                      section critique
      Vendre(s2)                            Vendre(s1)
      Vendre(s1)                            Vendre(s2)
Fait                                  Fait
```

On voit que si l'un des processus détient une ressource, et que l'autre processus détient l'autre ressource, alors les deux processus sont bloqués indéfiniment : c'est un interblocage. Il existe plusieurs solutions au problème d'interblocage : par exemple, la détection, l'évitement, la reprise après interblocage. On peut se faire une idée de la complexité de la détection d'un interblocage sur un exemple classique : le dîner des philosophes (figure 4.20). Plusieurs philosophes sont assis autour d'une table, pensent, et mangent de temps en temps en utilisant des baguettes. Mais les facétieux

© Dunod – La photocopie non autorisée est un délit.

cuisiniers n'ont disposé à table qu'une seule baguette par philosophe : tout se passe bien, jusqu'à ce que chaque philosophe, ayant faim au même moment, ne saisisse une baguette. Les processus sont alors en interblocage.

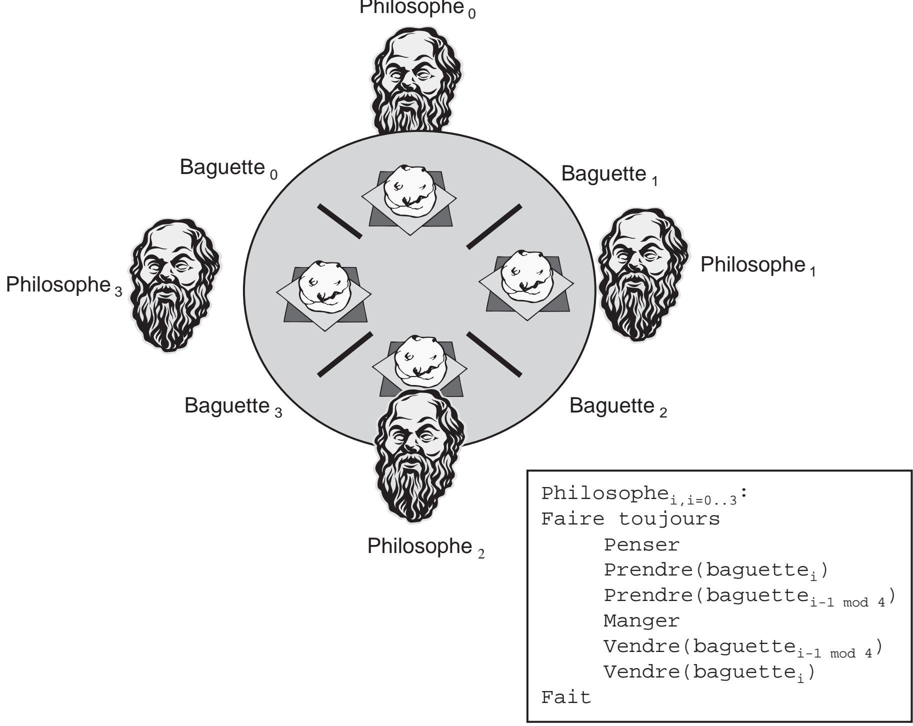


Figure 4.20 – Un exemple classique d'interblocage, le dîner des philosophes.

Une solution simple d'**évitement de l'interblocage** consiste, lorsque cela est possible, à prendre les sémaphores dans un ordre déterminé. Ainsi, chaque sémaphore peut être numéroté, et un processus ayant besoin de plusieurs ressources pour entrer en section critique, ne peut demander à prendre des sémaphores que dans l'ordre croissant de leur numéro. Sur la figure 4.20, si l'on numérote les baguettes de 0 à 3 dans le sens des aiguilles d'une montre, alors trois des philosophes prendront la baguette à leur droite, puis la baguette à leur gauche, et le quatrième prendra la gauche avant la droite, rompant ainsi le cycle fatal. Nous verrons qu'il existe une autre solution basée sur l'ordonnancement des processus.

□ Le moniteur

Nous avons vu que l'inconvénient des sémaphores était le risque d'oubli : une ressource critique (par exemple une variable partagée par plusieurs processus) est protégée par un sémaphore. Il faut donc normalement prendre le sémaphore avant tout accès, et le vendre après. Cependant, sur une application relativement importante,

on pourrait oublier de protéger un accès, ce qui aurait pour effet de compromettre l'exclusion mutuelle. Une solution alternative a été proposée par Hoare en 1973 : le **moniteur**. Le principe du moniteur est d'encapsuler une ressource, et de ne permettre son utilisation qu'à travers des **primitives** (procédures ou fonctions) qui elles-mêmes sont protégées contre la réentrance. La **réentrance** d'une primitive, une fonction par exemple, a lieu lorsqu'un processus exécute le code de la fonction, et qu'il est interrompu par un autre processus exécutant lui aussi la fonction. Au niveau du moniteur, qui peut être muni de plusieurs primitives, la notion de réentrance s'étend à l'ensemble des primitives du même moniteur. Le moniteur se retrouve tel qu'il a été défini par Hoare dans la norme POSIX (voir chapitre 5) et dans le langage Java, et sous forme d'un moniteur « amélioré », l'objet protégé, dans le langage Ada 95 (voir chapitre 6).

De façon basique, un moniteur possède des variables internes, accessibles uniquement via des primitives non réentrantes (figure 4.21), ce qui signifie que lorsqu'un processus exécute une primitive d'un moniteur, un autre processus qui voudrait utiliser une primitive du même moniteur se voit bloqué et mis dans la file d'attente du moniteur. Ainsi, la gestion d'une section critique est très simple, et il est impossible d'oublier de garantir son exclusion mutuelle : il suffit de ne rendre accessible la ressource (une variable par exemple) qu'à travers le moniteur.

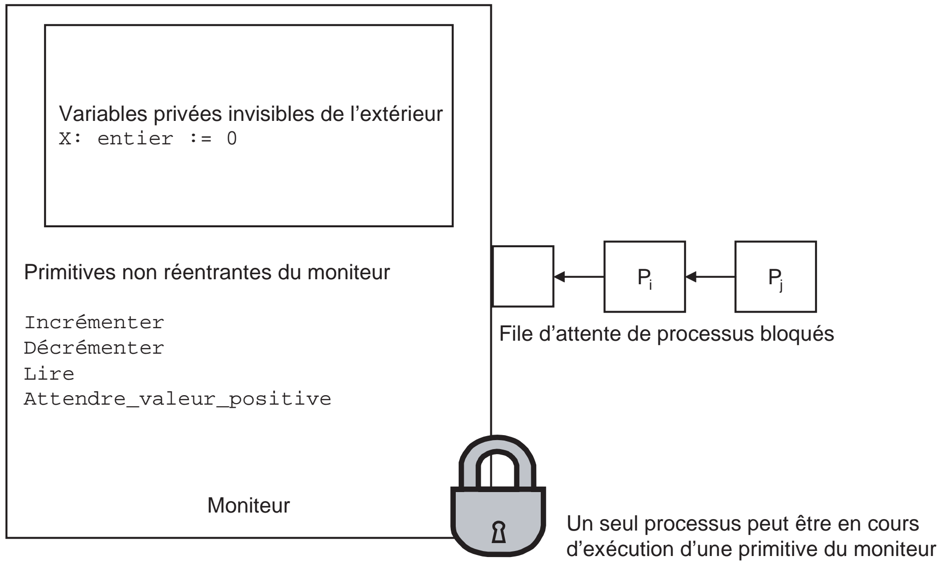


Figure 4.21 – Illustration d'un moniteur.

Par exemple, voici une variable *x* partagée par plusieurs processus :

```
Moniteur moniteur_x
      x : entier :=0
      Procédure incrémenter
      Début
            x :=x+1
```

© Dunod – La photocopie non autorisée est un délit.

```
            -- à cause de la non réentrance, il est impossible que
            -- deux processus aient accès simultanément à x, via
            -- incrémentation ou décrémentation
      Fin
            Procédure décrémenter
      Début
            x :=x-1
      Fin
      Fonction Lire renvoie entier
      Début
            Renvoyer x
      Fin
Fin du moniteur
```

Il suffit à un processus voulant incrémenter x d'appeler la procédure `moniteur_x.incrémenter`. L'exclusion mutuelle est garantie par la non réentrance.
Le moniteur est donc un outil de synchronisation plus simple à utiliser que le sémaphore pour garantir l'exclusion mutuelle. Cependant, comment faire en sorte de bloquer un processus en attendant qu'une certaine condition soit réalisée ? Pour ce faire, étudions ci-après une solution incorrecte au problème du producteur/consommateur :

```
Moniteur prod_cons
      taille:entier -- taille du tampon
      T:tampon(taille)
      nombre_pris:entier:=0 -- nombre de cases prises dans le
      tampon
      Procédure produire(e:élément)
      Début
            Tant que nombre_pris=taille faire
            -- le tampon est plein
                  rien
            Fait
            Mettre e dans T
            nombre_pris:=nombre_pris+1
      Fin
      Fonction consommer renvoie élément
      Début
            Tant que nombre_pris=0 faire
            -- le tampon est vide
                  rien
            Fait
            nombre_pris:=nombre_pris-1
            Enlever et retourner un élément de T
      Fin
Fin du moniteur
```

Cette solution n'est pas satisfaisante. En effet, d'une part, même si elle pouvait fonctionner, elle contiendrait de l'attente active. Mais surtout, elle ne peut pas fonctionner, étant donné que lorsqu'un processus se trouve dans la boucle « tant que », il bloque l'accès au moniteur, et empêche alors l'autre processus de modifier `nombre_pris`. Un processus entrant dans la boucle « tant que » reste donc indéfiniment dans cette boucle et bloque l'accès au moniteur. Le moniteur propose donc deux procédures : `wait` et `signal`. La procédure `wait` a pour effet de bloquer le processus appelant, libérant ainsi le moniteur, jusqu'à l'appel de la procédure `signal`. À l'appel de cette procédure, les processus bloqués sur `wait` seront remis dans l'état

prêt et poursuivront leur exécution. Ce sont ces procédures (voir code ci-après) qui sont utilisées lorsqu'un processus doit être bloqué sous certaines conditions à l'intérieur du moniteur, jusqu'à ce qu'un autre processus vienne le réveiller. La solution correcte au problème producteur/consommateur est donc :

```
Moniteur prod_cons
      taille:entier -- taille du tampon
      T:tampon(taille)
      nombre_pris:entier:=0 -- nombre de cases prises dans le tampon
      Procédure produire(e:élément)
      Début
            Tant que nombre_pris=taille faire
            -- le tampon est plein
                  wait -- le processus s'endort jusqu'au prochain
                       -- signal
            Fait
            Mettre e dans T
            nombre_pris:=nombre_pris+1
            signal
            -- réveille un éventuel consommateur bloqué
      Fin
      Fonction consommer renvoie élément
      Début
            Tant que nombre_pris=0 faire
            -- le tampon est vide
                  wait
            Fait
            nombre_pris:=nombre_pris-1
            Enlever et retourner un élément de T
            signal
            -- réveille un éventuel producteur bloqué
      Fin
Fin du moniteur
```

Lorsqu'un processus doit produire (resp. `consommer`) une donnée, il lui suffit donc d'appeler `prod_cons.Produire` (resp. `prod_cons.Consommer`), et il n'y a pas à se soucier de faire une utilisation correcte des sémaphores comme dans la solution à base de sémaphores.

■ Ordonnancement de processus en présence de ressources

L'ordonnancement de processus a été abordé à l'aide d'exemples très académiques au paragraphe 4.2.1, p. 140. Les processus étaient considérés comme indépendants les uns des autres. Lorsque des processus partagent une ou des ressources, ils peuvent passer dans l'état *bloqué* afin de permettre la garantie d'exclusion mutuelle, lorsqu'ils tentent d'accéder à un sémaphore ou bien à un moniteur. Il est très intéressant d'observer l'impact des sections critiques sur les algorithmes à priorité, qui sont très utilisés dans les systèmes temps réel et dans les systèmes de contrôle-commande, ainsi que dans l'algorithme MLF utilisé par nombre de systèmes d'exploitation généralistes.

Observons un effet possible de l'exclusion mutuelle sur des processus (figure 4.22). Trois processus s'exécutent à des niveaux de priorité différents. Le processus de priorité haute partage une ressource avec le processus de priorité basse, et un processus de priorité intermédiaire s'exécute indépendamment des deux autres.

© Dunod – La photocopie non autorisée est un délit.

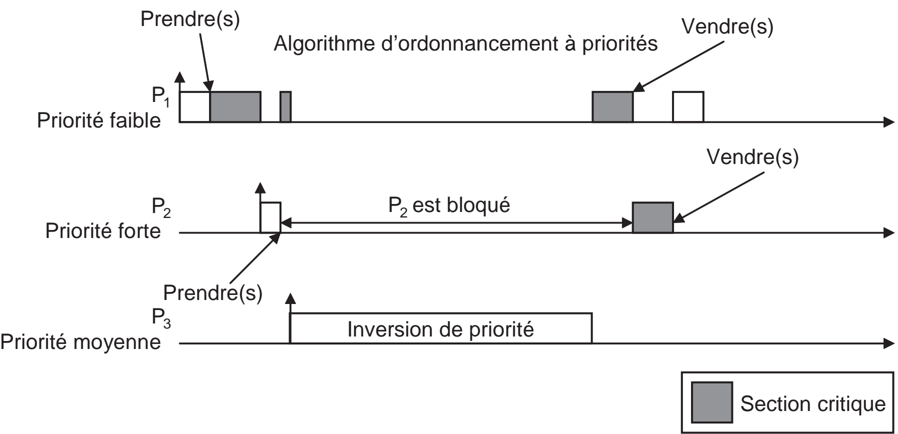


Figure 4.22 – Inversion de priorité.

Lorsque le processus de priorité basse est en section critique, le processus de priorité haute ne peut bien sûr pas entrer en section critique, il est donc bloqué : cela est inévitable. Cependant, le phénomène que l'on observe sur la figure 4.22 est que le processus de priorité intermédiaire, s'il est prêt au moment où le processus de faible priorité est en section critique, obtient le processeur alors même qu'un processus de priorité haute est en attente. Ce phénomène s'appelle une **inversion de priorité** et est à proscrire dans les systèmes pour lesquels le temps de réponse est important. La plupart des noyaux temps réel ou langages de programmation pour le temps réel proposent une solution à ce problème. Les solutions empêchant l'inversion de priorité s'appuient sur les travaux sur la « super priorité » proposée par Kaiser en 1982, et sur les protocoles à priorité héritée et plafond proposés en 1987 par Sha, Rajkumar et Lehoczky. Ces protocoles sont présentés au chapitre 8.

4.2.3 Gestion de la mémoire et mémoire virtuelle

Cette section expose la façon dont la plupart des systèmes d'exploitation gèrent la mémoire (segmentation paginée) afin de mieux affronter les problèmes posés par le parallélisme : identifier les problèmes d'exclusion mutuelle, comprendre la notion de réentrance, comprendre le mécanisme de la mémoire virtuelle qui peut sur certains systèmes d'exploitation influencer l'ordonnancement des processus, et leur durée d'exécution.

■ Segmentation de la mémoire

Chaque processus possède sa mémoire propre : ceci s'explique par le fait qu'un système d'exploitation est plus robuste si chaque processus est cantonné dans un espace mémoire clos. Au lancement d'un processus, le système d'exploitation lui alloue de la mémoire dans la mémoire centrale qu'il segmente en trois parties distinctes : le **segment de code**, dans lequel il place les instructions du processus (voir tableau 4.1), alloue un **segment de pile** (appelé ***stack***), et un **segment de données** (appelé **tas**,

ou ***heap***) contenant les variables globales ou allouées dynamiquement pendant l'exécution du processus. La figure 4.23 montre un exemple de segmentation. Cette segmentation permet divers contrôles : ainsi, il est vérifié que le compteur ordinal pointe toujours vers une instruction située dans le segment de code, on peut aussi vérifier que le segment de code n'est pas modifié par l'exécution (segment en lecture seule), et que la pile ne déborde pas sur un autre segment. Enfin, le système d'exploitation peut vérifier, sauf demande contraire de la part du processus (comme un passage en mode débogueur, communication avec un autre processus, utilisation d'adresses d'entrées/sorties, etc.), que le processus ne va jamais lire ou écrire en dehors de ses segments. Cela permet de protéger la mémoire d'un processus des actions erronées ou malveillantes d'autres processus.

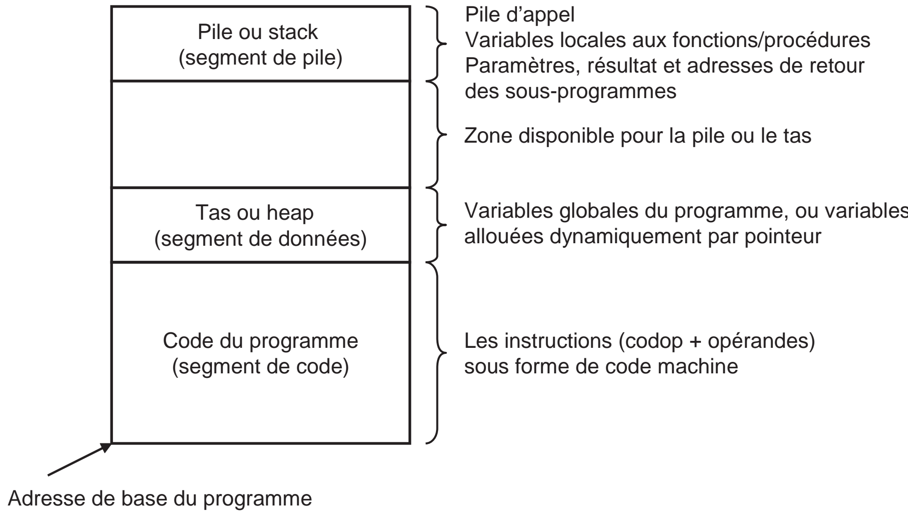


Figure 4.23 – Exemple de segmentation de la mémoire d'un processus.

Le segment de données sert à stocker les variables globales, les variables rémanentes des sous-programmes (par exemple, en langage C, celles déclarées avec le mot clé `static`), ainsi que les variables allouées dynamiquement (pointeurs).

Le segment de pile, quant à lui, sert à conserver les contextes de sous-programmes. La pile est caractérisée par l'adresse du sommet. Le fait d'empiler une valeur consiste à copier la valeur à l'adresse du sommet de pile, puis à incrémenter le sommet de pile de la taille de la valeur empilée. En réalité, on occupe souvent un nombre de mots machine entier, c'est-à-dire que sur une machine à mots de 32 bits, l'empilement d'un élément de la taille d'un octet incrémentera d'un mot, donc 4 octets, le sommet de pile : 3 octets sont perdus, mais le sommet de pile est toujours aligné sur les mots machine, ce qui permet une optimisation d'accès à la mémoire centrale. Dépiler un élément consiste à supprimer l'élément en sommet de pile, en réalité, décrémenter le sommet de pile de la taille de l'élément : l'élément est toujours présent, mais il sera écrasé au prochain empilement.

© Dunod – La photocopie non autorisée est un délit.

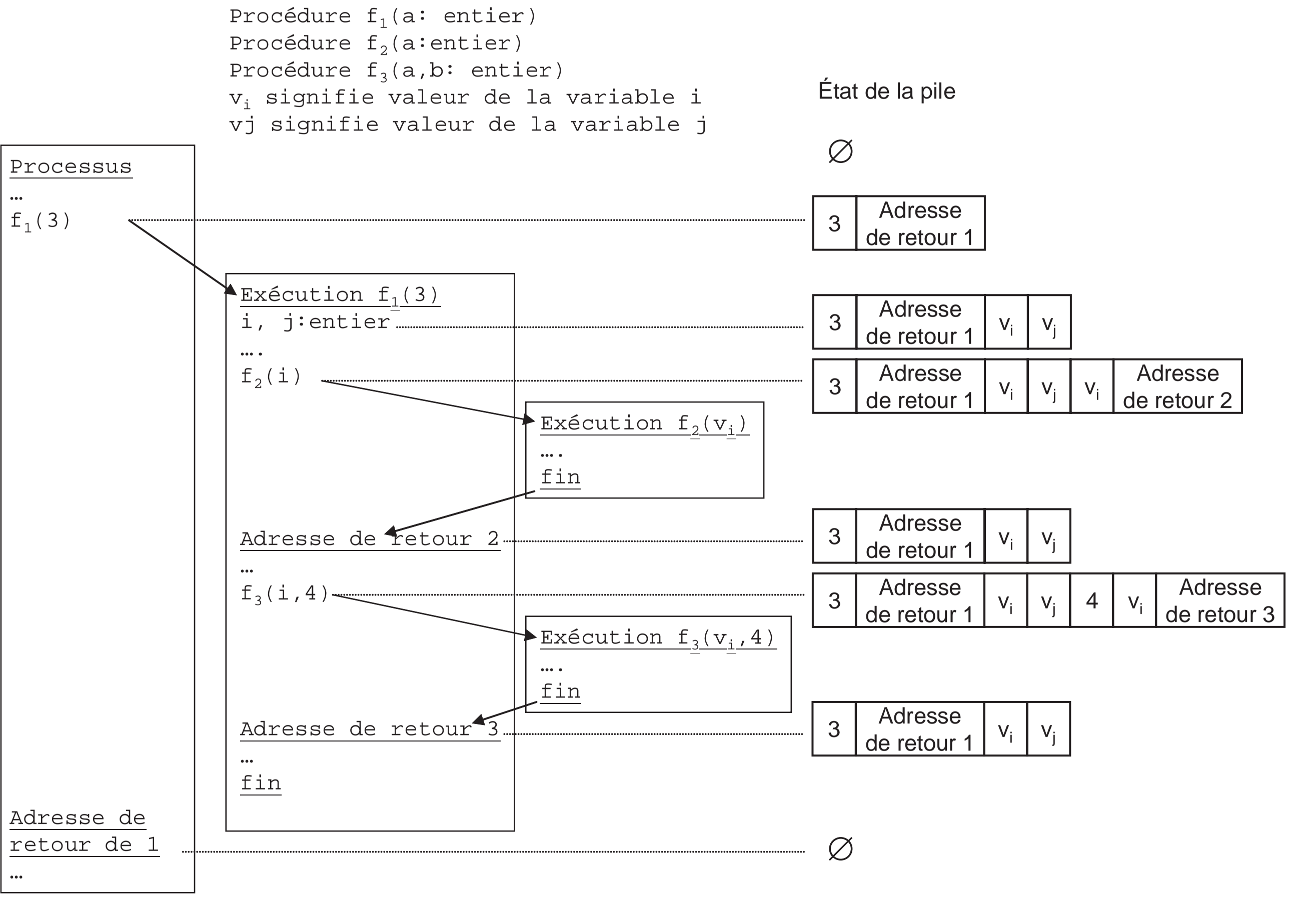


Figure 4.24 – États successifs de la pile lors de l'appel de sous-programmes.

Lorsqu'on appelle un sous-programme, les paramètres d'appel du sous-programme sont empilés, puis une instruction machine CALL avec en paramètre l'adresse du sous-programme est exécutée. Cette instruction a pour effet d'empiler la valeur du compteur ordinal (c'est-à-dire l'adresse de l'instruction suivant l'appel du sous-programme), puis de mettre l'adresse du sous-programme dans le compteur ordinal. Cela a pour effet d'exécuter le sous-programme. La première action effectuée est alors d'empiler toutes les variables locales du sous-programme. Le sous-programme sait que les paramètres qui lui ont été passés se trouvent dans la pile, à un décalage (*offset*) calculable par rapport à ses propres variables locales. À la fin du sous-programme, les variables locales sont dépilées, et une instruction RET est appelée. Cette instruction dépile le sommet de pile (qui est donc la valeur du compteur ordinal, c'est-à-dire l'adresse de l'instruction suivant l'appel du sous-programme), et le met dans le compteur ordinal. Le programme appelant continue alors son exécution. La pile conserve donc le contexte d'exécution : au fur et à mesure des appels de sous-programmes, c'est elle qui conserve le contexte dynamique (paramètres d'appel, variables locales et adresse suivant l'appel du sous-programme) de l'exécution.

Par exemple (figure 4.24), si un processus appelle un sous-programme f_1 qui lui-même appelle f_2 puis f_3, on verra la pile contenir d'abord les paramètres d'appel de f_1 et l'adresse suivant l'appel, puis les variables locales de f_1. Ensuite seront empilés les paramètres d'appel de f_2 et l'adresse suivant l'appel de f_2 dans f_1 ; puis les variables locales de f_2. À la fin de f_2, ses variables locales sont dépilées. L'adresse de retour de la fonction permet alors de replacer le compteur ordinal sur l'instruction suivant l'appel de fonction dans le code de f_1. f_1 reprend donc son exécution après avoir dépilé les paramètres d'appel de f_2. On se retrouve alors dans le même état que juste avant l'appel de f_2. Il se passe la même chose pour l'appel du sous-programme f_3. À la fin de f_1, ses variables locales sont à leur tour dépilées, on retrouve alors en sommet de pile l'adresse de retour dans le programme principal, qui est à son tour dépilée, ainsi que les paramètres d'appel à f_1. On se retrouve alors dans le contexte du programme principal.

Il est important de noter que les variables locales à un sous-programme sont fugaces lorsqu'on fait de la programmation multitâche : nous verrons d'ailleurs à travers un exemple (voir chapitre 5) l'une des erreurs possibles dues à cette non rémanence.

Pagination de la mémoire et mémoire virtuelle

Si l'on observe les adresses des instructions et variables du tableau 4.1, on trouve des valeurs de l'ordre de 0x400000, soit des adresses aux alentours de 4 Mo. Si l'on compare les adresses utilisées par plusieurs processus s'exécutant en parallèle, on pourra trouver des adresses identiques, qui ne correspondent pas aux mêmes adresses physiques. En effet, le système d'exploitation permet aux processus d'utiliser un adressage virtuel : pour un processus, tout se passe comme s'il possédait les adresses 0 à x, sachant que le système d'exploitation se charge de faire une conversion de l'adresse virtuelle en adresse physique.

Le but de la **pagination** est le suivant : un très grand espace mémoire peut être alloué à chaque processus, tel que la somme soit supérieure à la taille de la RAM. La mémoire centrale est découpée en **cadres** de taille identique (par exemple 4 Ko). La mémoire d'un processus chargée en mémoire centrale réside alors dans un certain

© Dunod – La photocopie non autorisée est un délit.

nombre de cadres. Au fur et à mesure qu'un processus demande (resp. libère) de la mémoire, une **page** de la taille d'un cadre, lui est allouée (resp. est libérée). Le nombre de pages allouées est souvent supérieur au nombre de cadres de la mémoire physique. C'est le disque dur qui permet de stocker les pages allouées en sus du nombre de cadres, à l'intérieur d'un **fichier d'échange**.

Lorsque la mémoire allouée dépasse la taille de la mémoire physique, un certain nombre de pages de mémoire allouée aux processus réside en mémoire centrale, le reste réside dans le fichier d'échange. Lorsqu'un processus a besoin d'accéder à une page qui se trouve dans le fichier d'échange, il faut libérer un cadre de la mémoire centrale afin d'y placer la page nécessaire. Cela implique une copie vers le fichier d'échange de la page contenue dans le cadre. Le cadre de la RAM libéré est utilisé pour stocker la page demandée. Ce phénomène s'appelle le ***swap***, il est visible sur un micro-ordinateur classique dans lequel plusieurs processus sont lancés : basculer de l'un à l'autre des processus, ou accéder à une fonctionnalité d'un processus qui n'a pas été utilisée depuis longtemps a pour effet visible de faire fonctionner le disque dur. Ce fonctionnement est dû à l'échange des pages entre mémoire centrale et fichier d'échange.

L'exécution est donc flexible, car il est possible d'allouer aux processus plus de mémoire que la mémoire physique, mais parfois le système est extrêmement ralenti (comparer les quelques nanosecondes nécessaires pour accéder à la RAM avec les quelques millisecondes nécessaires à un accès disque). L'adresse physique d'une même donnée peut donc varier pendant l'exécution d'un processus au fil des échanges entre mémoire centrale et fichier d'échange. On distingue donc **adresse physique** d'une donnée (adresse liée à la mémoire physique) et **adresse virtuelle** (adresse relative au numéro de page, plus adresse dans la page). La pagination permet donc l'utilisation d'une **mémoire virtuelle**.

La stratégie utilisée pour choisir les pages à placer dans le fichier d'échange repose souvent sur le nombre d'accès à une page, et/ou sur sa dernière date d'accès (principe de localité). Le but de cette stratégie est de minimiser les **défauts de pages**, c'est-à-dire le nombre d'échanges entre mémoire centrale et fichier d'échange.

La transformation adresse virtuelle/adresse physique est généralement effectuée de façon matérielle grâce à un dispositif appelé **MMU** (***Memory Management Unit***) présent dans la plupart des microprocesseurs.

Voyons comment la MMU transforme une adresse virtuelle en adresse physique.

Pour simplifier l'explication, nous supposerons que les pages sont de 4 Ko, soit 2^{12} octets, et que 32 bits sont utilisés pour exprimer une adresse, ce qui implique une mémoire adressable de 4 Go (2^{32} octets). Une adresse de 32 bits est donc décomposée en 20 bits donnant un numéro de page, et 12 bits permettant de donner l'adresse d'un octet relativement à l'intérieur de la page. En tout, le système de MMU peut alors gérer 2^{20} pages. Supposons que le système possède une mémoire centrale de 512 Mo, soit 2^{17} cadres physiques de 4 Ko.

Un processus peut se voir allouer jusqu'à 4 Go de mémoire, mais chacune des pages peut se trouver dans un cadre de la mémoire ou dans le fichier d'échange. Lorsqu'un processus demande à accéder à l'adresse 32 bits $a_{31}a_{30} \dots a_1a_0$, cette adresse est vue par la MMU en $p_{19}p_{18} \dots p_1p_0y_{11}y_{10} \dots y_1y_0$ (figure 4.25), soit 20 bits donnant un numéro de page du processus, et 12 bits donnant la position de l'octet dans la page.

Seule l'adresse physique du début de la page doit être retrouvée. Il faut alors une sorte d'annuaire par processus, appelée table des pages, qui fait la correspondance entre numéro de page logique, sur 20 bits, et adresse physique du cadre contenant la page. Si la page n'est pas en mémoire centrale, la MMU se charge de l'y placer avant de trouver l'adresse physique du cadre correspondant. Donc la MMU se charge d'effectuer la correspondance entre les 20 bits de l'adresse logique d'une page d'un processus et les 20 bits de l'adresse physique du cadre contenant la page (en effectuant si nécessaire un swap pour placer la page en mémoire centrale).

Cependant, quand on fait les comptes, on s'aperçoit qu'une page de 4 Ko peut contenir 2^{12} octets, ce qui n'est pas suffisant pour contenir les éventuelles 2^{20} adresses physiques des pages. Il y a donc souvent un index des pages à deux niveaux, avec un certain nombre de pages indexées directement, les autres nécessitant l'accès à des tables des pages de second niveau. En fonction des processus en cours d'exécution, le temps d'exécution d'un processus peut varier de façon significative à cause de la mémoire virtuelle : en effet un accès physique à la mémoire centrale coûte généra-

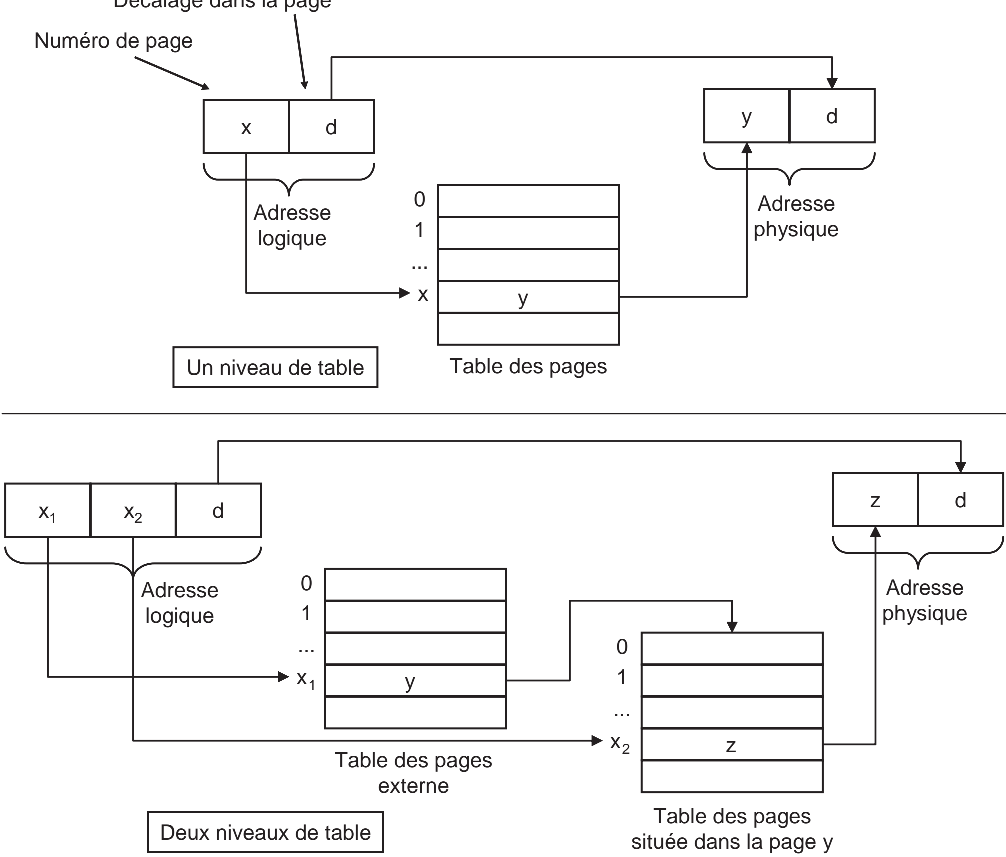


Figure 4.25 – Accès à l'adresse physique par table des pages.

© Dunod – La photocopie non autorisée est un délit.

lement 5 à 10 cycles machine, et il peut falloir 3 accès pour accéder à une variable (accès à la table des pages de premier niveau, accès à la table de second niveau, pour obtenir l'adresse physique du cadre contenant la variable), avec, suivant l'utilisation de la mémoire physique, possibilité d'avoir à *swapper* pour utiliser ces 3 pages. Dans le pire des cas, l'accès à une variable peut donc se compter en dizaines de millisecondes. Dans le meilleur des cas, les trois pages sont en mémoire centrale, et même, les valeurs nécessaires sont dans la mémoire cache (accès en un cycle machine) et l'accès à la variable ne coûte que trois cycles machine.

Sur un système d'exploitation utilisant la mémoire cache et la mémoire virtuelle, l'accès à une variable peut nécessiter un temps pouvant aller de moins d'une nanoseconde à plusieurs dizaines de millisecondes en fonction de l'utilisation de la mémoire cache et de la mémoire virtuelle par l'ensemble des processus.

4.3 Réseaux et bus de terrain

De plus en plus d'applications de contrôle-commande utilisent le réseau, que ce soit pour délocaliser les capteurs ou pour utiliser plusieurs calculateurs voire même passer par internet pour interagir avec des opérateurs distants de plusieurs milliers de kilomètres du procédé contrôlé. Cette section a pour but de donner au lecteur quelques idées sur l'existant dans le domaine des télécommunications. Bien que différents réseaux spécialisés, appelés réseaux de terrain, aient été développés dans le but de faire communiquer des éléments en temps déterministe, il faut noter l'émergence des réseaux généralistes non déterministes en temps (notamment à base de TCP/IP) dans le monde du contrôle de procédé, notamment grâce leurs performances élevées en terme de débit.

4.3.1 Modes de communication

Que les données passent sur un médium matériel (câble, fibre optique…) ou par ondes hertziennes (ondes radio, lumineuses…), il existe deux grands modes de communication :

– la **communication point à point**, où deux entités, appelées **nœuds**, communiquent directement ;

– la **communication par diffusion** où un ensemble de nœuds envoient des données visibles par tous les nœuds du réseau, et peuvent recevoir des informations directement de la part de n'importe quel nœud du réseau.

En mode point à point, différentes architectures de réseau peuvent être employées (figure 4.26). L'**arbitrage** de l'accès au médium de communication peut être centralisé (utilisation d'un arbitre d'accès) ou décentralisé. Dans tous les cas, l'allocation du médium peut être statique (déterminisme de temps d'accès) ou dynamique (moins de déterminisme, mais plus de flexibilité).

Pour la communication par diffusion (figure 4.27), l'architecture la plus employée est de type bus ou boucle pour les supports matériels, et de type ad-hoc pour les supports immatériels.

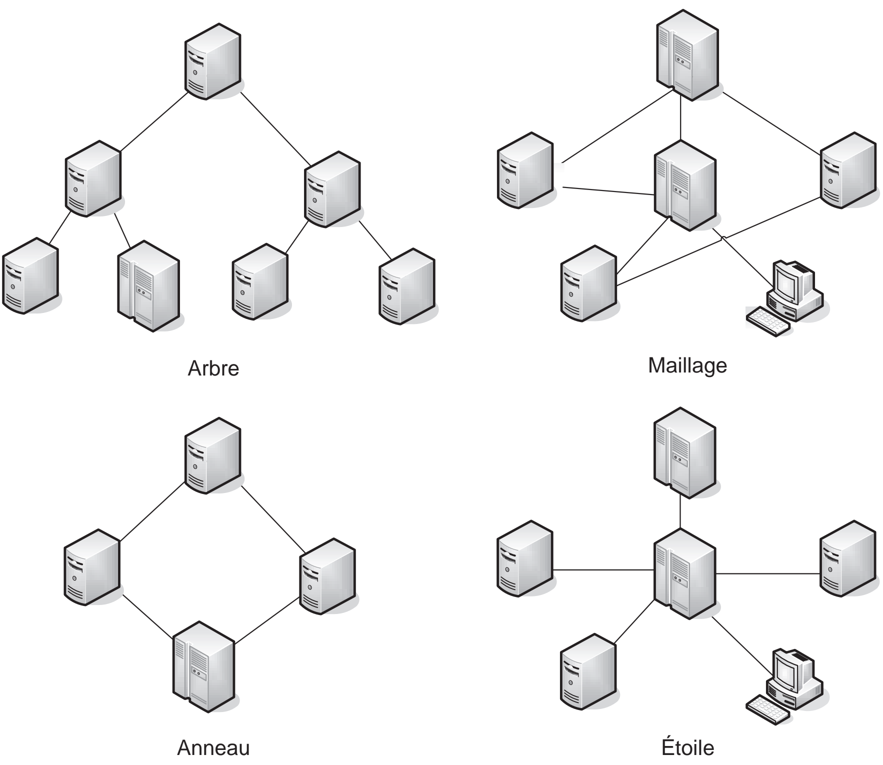


Figure 4.26 – Architectures de communication point à point.

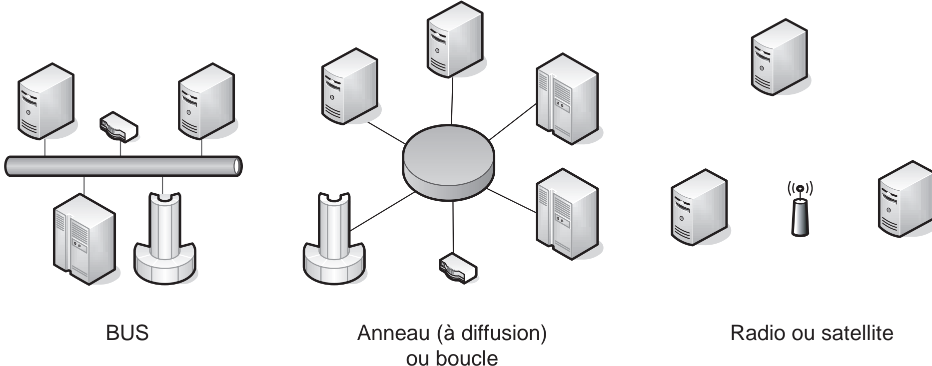


Figure 4.27 – Architectures de communication par diffusion.

Pour le **réseau internet**, ces deux types de communication sont mélangés : des **réseaux locaux** utilisent généralement de la diffusion, et un nœud particulier appelé **routeur** est connecté en point à point avec un autre routeur, lui-même souvent point d'entrée d'un autre réseau local. Les connections point à point sont souvent hiérarchiques,

© Dunod – La photocopie non autorisée est un délit.

ce qui permet à n'importe quel nœud d'un réseau local de communiquer avec n'importe quel nœud d'un autre réseau local (figure 4.28), comme nous le verrons avec le protocole TCP/IP.

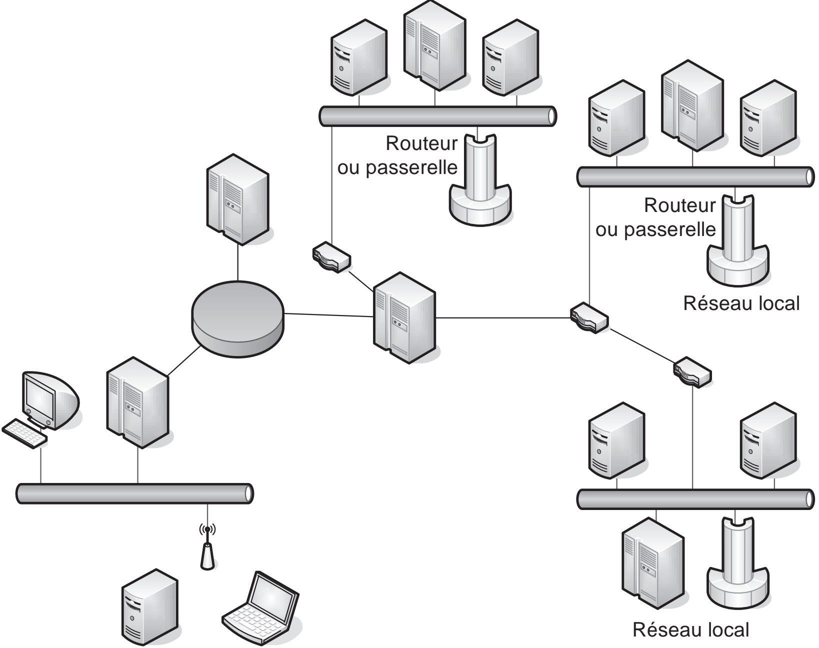


Figure 4.28 – Structure d'internet ou le mélange des genres.

4.3.2 Architecture des réseaux

Étant donné que les réseaux sont utilisés pour connecter des systèmes hétérogènes, les approches réseaux se basent sur une architecture en couche. Le principe est qu'une couche de niveau i délivre des services à une couche de niveau $i + 1$.

■ Architecture en couches

Historiquement, le plus ancien modèle en couches est toujours le plus utilisé aujourd'hui : c'est le modèle physique, MAC, LLC, IP, TCP ou UDP (figure 4.29).

L'avantage d'une architecture en couche est l'indépendance d'une couche par rapport aux autres : une couche doit délivrer un ensemble de services à la couche au-dessus d'elle, en se servant des services délivrés par la couche du dessous. De plus, une couche doit savoir dialoguer avec une même couche distante : un dialogue entre couches du même niveau s'appelle un **protocole**. Ainsi, les noms TCP, IP, ARP, etc. correspondent à des protocoles. La finalité est de faire en sorte que deux applications échangent des données, suivant un protocole d'application. Pour cela, les applications peuvent utiliser les services d'une couche TCP. Ce protocole sert à transférer des

© Dunod – La photocopie non autorisée est un délit.

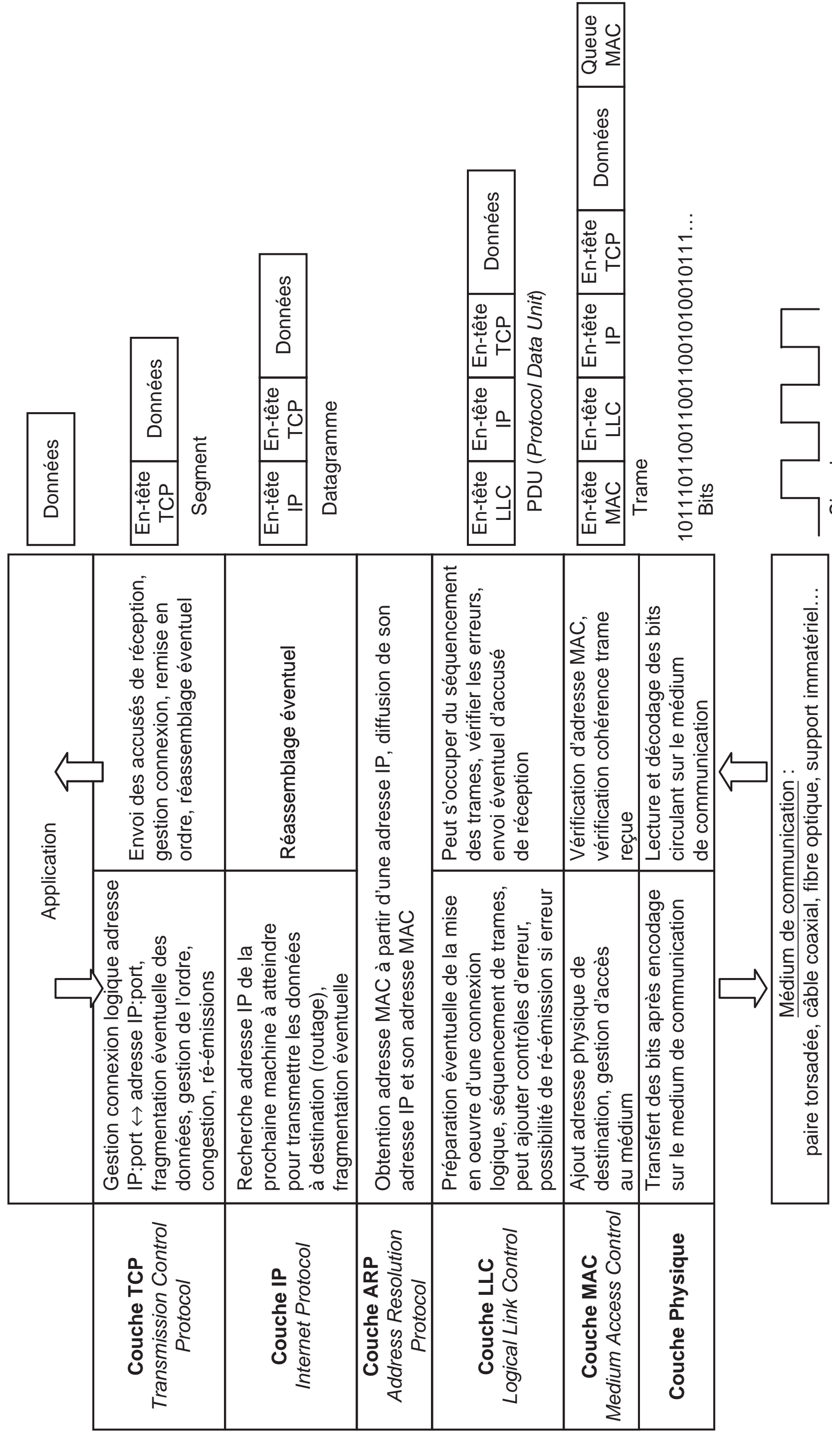


Figure 4.29 – Structure en couche du protocole TCP/IP.

données d'application de façon fiable. Afin de dialoguer avec la couche TCP du nœud destination, TCP ajoute un en-tête qui sera lu et enlevé du message lors de sa réception par la couche TCP du nœud destination.

Chaque couche ajoute ses propres données (sous forme d'un en-tête et/ou queue) aux données transmises par la couche du dessus lors de l'envoi de données. Lors de la réception de données, chaque couche se sert de l'en-tête et/ou queue placée par la couche correspondante du nœud émetteur afin de traiter les données reçues. Elle enlève ensuite ces informations qui sont spécifiques à son niveau, pour passer les données à la couche de niveau supérieur. L'avantage d'une telle approche est que, quel que soit le support physique sous-jacent (câble, fibre optique, air, vide…), les couches à partir de LLC peuvent fonctionner de la même façon. De plus, pour un programmeur, le support de transmission employé est totalement transparent.

L'ISO (*International Organization for Standardization*), l'AFNOR (Association Française de Normalisation) et le CCITT (Comité Consultatif International Téléphonique et Télégraphique, devenu depuis ITU pour *International Communication Union*) ont normalisé le **modèle OSI** (*Open Systems Interconnection*) en couche, qui est utilisé par des réseaux définis ultérieurement à **TCP/IP** (comme par exemple le protocole X.25 qui définit comment une communication par modem et ligne téléphonique se déroule en point à point). Ce modèle est illustré sur la figure 4.30. Les couches diffèrent dans leur découpage, mais on peut réaliser les associations suivantes : couche 1 = couche physique, couche 2 = MAC + LLC + $^1/_2$ ARP, couche 3

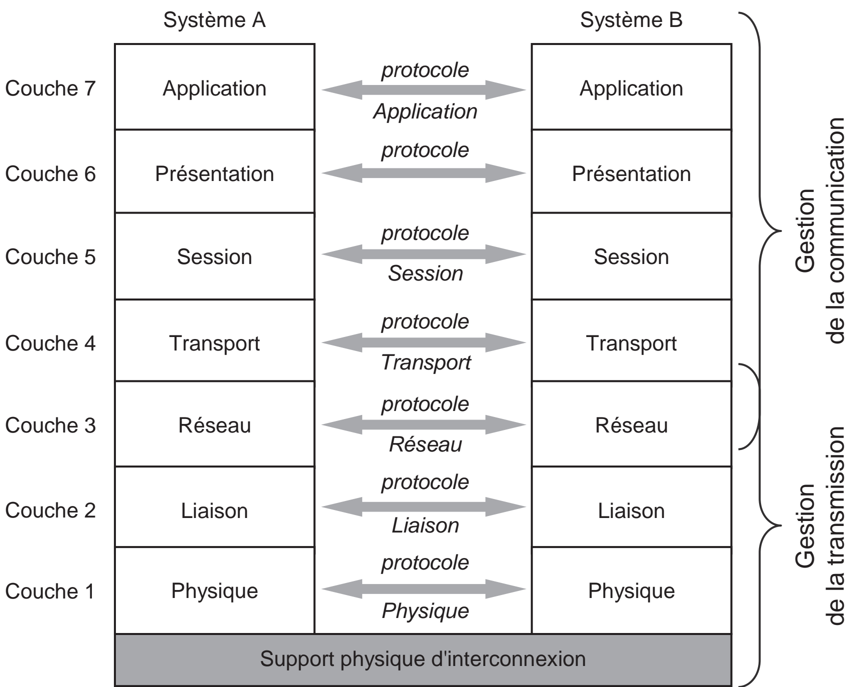


Figure 4.30 – Structure en couche du modèle OSI.

= IP + $^1/_2$ ARP, couche 4 et une partie de la couche 5 = TCP, couches 5, 6, 7 = Application.

Cet ouvrage n'a pas pour but de décrire exhaustivement le fonctionnement de chacun des protocoles existants, mais il décrit succinctement le protocole généraliste le plus employé (TCP/IP) et un des réseaux de terrain les plus utilisés dans l'industrie (CAN).

Certains paramètres peuvent influencer le choix du type de médium de communication utilisé et de mode d'accès :

– le débit binaire maximal, soit le nombre de bits par seconde que l'on peut transférer ;
– la taille maximale d'une trame en octets (une trame est une entité unitaire transférable sur le médium de communication) ;
– le médium physique utilisé, sa tolérance, et sa résistance aux éléments extérieurs, comme les perturbations électromagnétiques ou les conditions climatiques : paire torsadée, câble coaxial, fibre optique, support immatériel… ;
– la distance maximale entre deux nœuds à vue directe ;
– à diffusion (bus, support immatériel, boucle) ou point à point ;
– le type de multiplexage utilisé pour le passage des données : fréquentiel, dans lequel plusieurs messages peuvent être transmis simultanément, chaque nœud se partageant une partie du médium et/ou temporel, pour lequel les nœuds ne peuvent émettre que chacun à leur tour. La plupart des réseaux destinés à transporter des données utilisent le multiplexage temporel, afin d'utiliser intégralement la bande passante si au moins un nœud veut émettre ;
– le déterministe ou non du temps d'accès au médium ;
– le type d'accès au médium, par arbitre ou décentralisé ;
– la confidentialité des données (même cryptées, les données transmises sur support immatériel peuvent être aisément reçues par des nœuds tiers).

■ Protocoles réseaux généralistes

□ Couche 1 (physique)

Les bits peuvent être transmis sous la forme de signaux en bande de base (2 ou 3 états électriques différents servent à passer les bits), ou bien en modulation (amplitude, fréquence et/ou phase). Les différents codages utilisés sont étudiés afin d'éviter une désynchronisation d'horloge, et afin d'être tolérants aux perturbations électromagnétiques ou optiques. Cet ouvrage ne traite pas de traitement du signal, mais le lecteur intéressé pourra se référer à la bibliographie.

□ Couche 2 (LLC, MAC)

La couche 2 du modèle OSI correspond approximativement aux couches **LLC** (***Logical Link Control***) et **MAC** (***Medium Access Control***) de la pile IP. Elle s'occupe notamment de la gestion de l'accès au médium : lorsque plusieurs nœuds sont susceptibles d'émettre sur un médium partagé, des collisions de données peuvent se produire, nécessitant une ré-émission. Cela influe évidemment sur le déterminisme

© Dunod – La photocopie non autorisée est un délit.

du temps de transmission d'information entre deux nœuds. Par conséquent, cette partie présente quelques protocoles de niveau 2 afin d'informer le lecteur sur le déterminisme temporel qu'il est possible d'obtenir.
La figure 4.31 présente différents types d'accès au médium de communication.

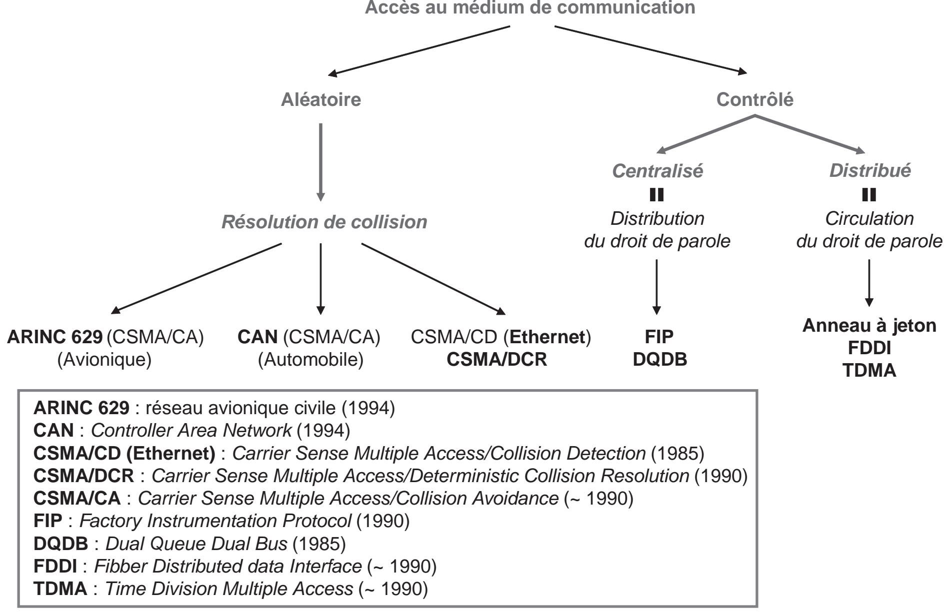


Figure 4.31 – Types d'accès au médium de communication.

Ethernet

La norme **Ethernet** (normalisée IEEE 802.3) est la norme de niveau MAC la plus répandue à l'heure actuelle sur les réseaux locaux. Cette norme fait son apparition sur le marché industriel et concurrence désormais les bus de terrain, car bien que théoriquement moins robuste, elle a un coût financier très faible, des débits moyens très élevés, et des temps moyens d'accès au médium très bas. Au niveau physique, Ethernet utilise généralement de la transmission numérique sur paires torsadées ou câbles coaxiaux. Les débits sont importants, puisque suivant le matériel utilisé, on peut échanger des données à 10 Mbits/s, 100 Mbits/s, 1 Gbits/s, etc. La communication est de type diffusion (le plus souvent, sur un bus), et chaque nœud est muni d'une adresse (sur 2 octets, ou plus, typiquement sur 6 octets) dite **adresse MAC** ou **adresse Ethernet**.
Lorsqu'un nœud émet une trame (figure 4.32) à destination d'un autre nœud, tous les nœuds du réseau la reçoivent, mais seul le nœud possédant l'adresse destination doit lire la trame.
En réalité, les bus Ethernet ont une forme d'étoile ramifiée (figure 4.33). Dans le protocole originel, toute trame émise par un nœud est diffusée à tous les nœuds

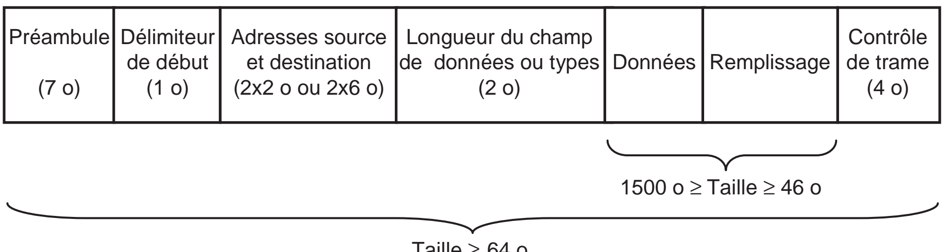


Figure 4.32 – Format d'une trame Ethernet.

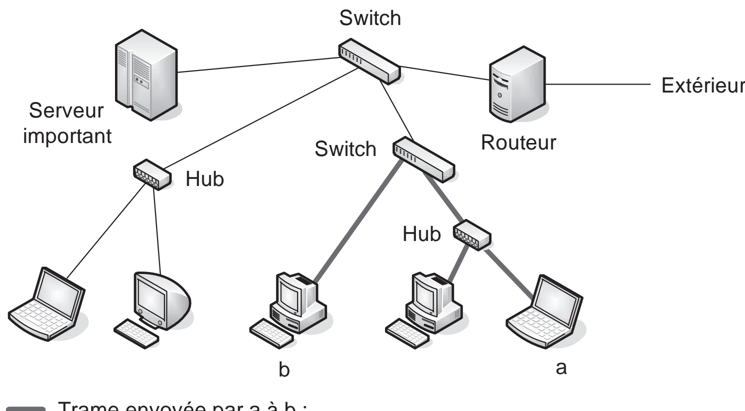


Figure 4.33 – Un « bus » Ethernet ressemble davantage à une étoile ramifiée qu'à un bus.

du réseau, ce qui revient à considérer cette topologie comme un bus à diffusion. Chaque extrémité de l'étoile ramifiée est un nœud (ordinateur, imprimante, photocopieuse, matériel d'acquisition/commande, etc.), alors que les intersections sont constituées d'éléments répéteurs passifs (les ***hubs***), ou actifs (les commutateurs, ou ***switchs***). Un *hub* relaie ce qu'il reçoit sur une entrée à toutes ses sorties, alors qu'un *switch* s'informe sur la topologie du réseau afin de ne répercuter les trames reçues que vers les branches concernées. Un *switch* est en fait une passerelle de niveau MAC.

Ethernet arbitre l'accès au medium suivant le protocole d'accès décentralisé non déterministe **CSMA/CD** (***Carrier Sense Multiple Access with Collision Detection***) : lorsqu'un nœud souhaite émettre des données, il attend que le médium soit libre (pas d'émission en cours) et émet une trame contenant ses données. Tout en continuant à émettre, il observe le réseau pendant 2 fois la durée maximale de propagation d'un message (temps d'aller/retour maximal d'un signal entre les deux nœuds les plus éloignés du réseau). Pendant cette durée appelée **tranche canal**, il vérifie que ce qu'il lit sur le réseau correspond bien à ce qu'il écrit (*i.e.* il vérifie qu'aucun autre nœud n'a pris l'initiative d'émettre presque en même temps que lui). S'il s'aperçoit qu'il y a collision, il continue à émettre sa trame avec des bits de bourrage afin d'être

© Dunod – La photocopie non autorisée est un délit.

certain que tous les nœuds concernés s'aperçoivent aussi de la collision (il y a une taille minimale de trame de 64 octets afin d'être certain que deux stations entrant en collision s'aperçoivent de la collision, et que les nœuds destination des trames s'aperçoivent aussi de la collision). Les deux nœuds entrés en collision tirent alors un nombre aléatoire entre [0 et $2^n \times$ tranche canal-1] avec n le nombre de tentatives d'émission de la trame. Au bout d'un certain temps, non déterministe, si le réseau n'est pas surchargé, un nœud pourra émettre sa trame. Si le réseau possède trop de nœuds et que le nombre de collisions est important, on parle d'**écroulement** du réseau.

La longueur maximale d'un réseau Ethernet est fonction de la tranche canal, et de la longueur minimale d'une trame.

Afin de limiter les problèmes d'écroulement lorsque le nombre de nœuds sur un réseau Ethernet est important, l'**Ethernet commuté** est apparu : il se base sur des répéteurs actifs appelés *switchs*. Leur rôle est de collecter les adresses Ethernet des nœuds connectés à chacune de ses branches, ainsi, lorsqu'une trame arrive sur un nœud à destination d'une adresse, le *switch* ne relaie la trame que sur la branche menant au nœud destination. Les *switchs* permettent de réduire les **espaces de collision** (un espace de collision est un espace dans lequel tout message est vu par tous les nœuds).

Enfin, grâce aux paires torsadées, la communication entre les *switchs* ou *hubs* et les *nœuds* peut utiliser le *full-duplex*, c'est-à-dire émettre et recevoir en même temps des trames.

Le protocole Ethernet est extrêmement flexible, performant, et bon marché mais non déterministe.

Résolution déterministe des collisions

Pour permettre l'utilisation du protocole CSMA lorsque le déterminisme est nécessaire, le protocole **CSMA/DCR** (***Deterministic Collision Resolution***), norme IEEE 802.3D, a été proposé. Il introduit un mécanisme de priorités basé sur les adresses : au lieu d'attendre un temps aléatoire, les nœuds entrant en collision se scindent en 2 groupes : les nœuds plus prioritaires qui peuvent ré-émettre, et les nœuds moins prioritaires, qui doivent attendre, ceci jusqu'à ce que seul un nœud prioritaire puisse émettre. On peut ainsi déterminer le temps maximal d'attente du médium de communication en fonction de la priorité d'un nœud. En effet, avant d'émettre, un nœud prioritaire attend au plus la durée d'émission de la trame en cours, puis tente sa chance au plus $log_2(n)$ fois (n étant le nombre de nœuds voulant émettre en même temps que lui).

Wi-Fi

Une version sans fil d'Ethernet a été normalisée sous les normes IEEE 802.11, appelées **Wi-Fi** (***Wireless Fidelity***). Ce protocole, utilisant une transmission analogique à base de déphasage (quadrature de phase), permet des débits relativement importants (55 Mbits/s pour la norme 802.11g). Les ondes radio utilisées sont sur la bande des 2,4 GHz (ou 5 GHz pour la norme 802.11a). Le mode d'accès ne peut pas être de type CSMA/CD car deux nœuds émettant vers un nœud tiers peuvent

entrer en collision sans s'en apercevoir. À la place, le protocole **CSMA/CA** (***Collision Avoidance***) peut être utilisé : les nœuds ne peuvent bien entendu émettre que lorsqu'il n'y a pas d'émission reçue. Mais l'émission se base sur un protocole avec accusé de réception de la part du nœud destination. Un nœud souhaitant émettre vers un nœud commence par émettre une trame *Ready To Send* (prêt à émettre) vers le nœud destination, contenant des informations sur le volume de données à transmettre et la vitesse de transmission, le nœud destination, s'il ne capte pas d'autre émission ou n'attend pas d'autre émission, répond alors *Clear To Send* (émission possible) et le nœud peut émettre les données. Comme tous les nœuds à portée ont pu entendre la trame *Ready To Send*, ils attendent le temps escompté de transmission avant de tenter d'émettre. Il y a deux types d'architecture pour les réseaux Wi-Fi : une architecture centralisée, nommée **infrastructure**, centralisée autour d'un nœud routeur (généralement connecté à un réseau filaire) nommé ***Access Point***. La seconde architecture est dite **Ad-Hoc** ou **IBSS** (***Independant Basic Service Set***) : dans ce cas, chaque nœud ne peut communiquer qu'avec d'autres nœuds à vue directe. Dans le cas d'une infrastructure, un autre protocole d'accès au médium peut être utilisé à la place du CSMA/CA : c'est le **protocole à arbitre centralisé** (l'*Access Point*) **PCF** (***Point Coordination Function***) basé sur une méthode de scrutation.
Les normes 802.11 fournissent donc la possibilité de pouvoir communiquer sur des supports immatériels, cependant, bien que la norme 802.11g fournisse en théorie des débits pouvant atteindre 55 Mbits/s, la distance entre les nœuds, et les obstacles (murs, cloisons…) autorisent un débit réel bien inférieur. De plus, il est important de coupler ce protocole avec des techniques de cryptage, la possibilité de capter les transmissions à grande distance (quelques centaines de mètres, voire même quelques kilomètres avec une antenne directionnelle) a permis aux hackers d'inventer un nouveau passe-temps : le *War-Driving*, consistant à rechercher les réseaux Wi-Fi mal protégés et à s'y introduire.

Arbitrage distribué à base de jeton

Certaines méthodes d'accès contrôlé au médium se basent sur un jeton (*token*). Par exemple, la norme IEEE 802.4 définit le **bus à jeton** et la norme IEEE 802.5 définit l'**anneau à jeton**, protocoles d'accès au médium dans lesquels une trame jeton passe de nœud en nœud (de façon logique pour le bus à jeton, de façon physique pour l'anneau à jeton) : un nœud ne peut émettre que lorsqu'il possède le jeton, et ce pendant un temps fini. Le temps d'attente maximal d'accès au médium est alors borné, et le débit minimal alloué à un nœud est quantifiable. Des mécanismes de priorité peuvent être introduits dans ces deux normes.
La norme IEEE 802.6, nommée **FDDI** (***Fiber Data Distributed Interface***) est très proche de la norme anneau à jeton, excepté que la transmission du jeton s'adapte à la très grande vitesse de propagation des trames sur de la fibre optique, avec un objectif orienté temps de rotation maximum sur tout l'anneau.
Les protocoles 802.4, 5 et 6 sont donc des protocoles d'accès contrôlé mais décentralisé basés sur la transmission d'un jeton (physique pour les anneaux, logique pour le bus).

© Dunod – La photocopie non autorisée est un délit.

Arbitrage centralisé

Des protocoles à accès contrôlé de façon centralisée ont été définis, mais ils sont relativement peu utilisés pour les réseaux généralistes : étant donné que l'arbitre ne peut pas savoir à l'avance quelle station voudra émettre, et de quelle bande passante elle a besoin, une bonne partie de la bande passante est perdue par le mécanisme d'interrogation des nœuds. L'arbitre interroge chaque nœud, soit chacun leur tour (scrutation) soit avec un message destiné à tous (test) afin de savoir qui veut émettre. Il alloue alors le médium aux nœuds, chacun leur tour.

Citons le protocole **TDMA** (***Time Division Multiple Access***) qui accorde un certain temps à chaque nœud, qu'il veuille émettre ou non : ce mode déterministe et périodique gaspille énormément de bande passante.

□ Couche 3 (réseau)

Quelles que soient les couches de niveau 1 et 2, elles assurent une transmission entre deux ou *n* nœuds à vue directe, c'est-à-dire que, pour la couche 2, deux nœuds peuvent communiquer s'ils partagent le même médium de communication. La couche 3, s'appuyant sur la couche 2, a pour objet le routage des informations entre des nœuds pouvant appartenir à différents réseaux locaux. Le protocole de couche 3 le plus utilisé est le protocole **IP** (***Internet Protocol***). Chaque nœud visible sur un réseau possède une adresse IP (en plus bien sûr d'une adresse de niveau inférieur, comme une adresse Ethernet par exemple). Deux entités de couche 3 peuvent communiquer à partir du moment où il existe un chemin entre eux à travers des nœuds appelés **routeurs**. La communication physique passe de nœud en nœud à vue directe, mais la couche 3 se charge de choisir un chemin afin d'acheminer des données entre deux nœuds qui peuvent se trouver sur différents réseaux locaux.

Une **adresse IP** (dans la version 4) est une adresse constituée de 32 bits (figure 4.34) notée sous la forme de 4 chiffres compris entre 0 et 255, qui sera supplantée par une adresse sur 64 bits dans la norme IP version 6. À part les adresses spécifiques privées, IP version 4 définit 5 classes d'adresses IP.

Une **adresse de classe A** est composée d'un octet définissant un réseau (par exemple, 11.0.0.0 définit un réseau de classe A), et de 3 octets définissant les adresses des nœuds du réseau (un nœud du réseau peut avoir l'adresse 11.241.23.195), ce qui permet aux gestionnaires d'un réseau de classe A d'adresser plus de 16 millions de nœuds (2^{24}-2 machine, le suffixe .0.0.0 étant réservé pour adresser le réseau entier, et le suffixe .255.255.255 à diffuser à chaque nœud du réseau).

Une **adresse de classe B** est composée de deux octets définissant le réseau (par exemple, 130.15.0.0 est un réseau de classe B) et de deux octets pour définir l'adresse d'un nœud du réseau (par exemple, 130.15.65.123), ce qui permet d'adresser 2^{16}-2 machine.

Enfin, les réseaux de **classe C** sont définis sur 3 octets (par exemple, 193.55.198.0 est un réseau de classe C), ce qui laisse la possibilité d'adresser 254 nœuds (par exemple, 193.55.198.56).

Un organisme mondial, l'**ICANN** (***Internet Corporation for Assigned Names and Numbers***) attribue des classes d'adresses IP à différents organismes continentaux,

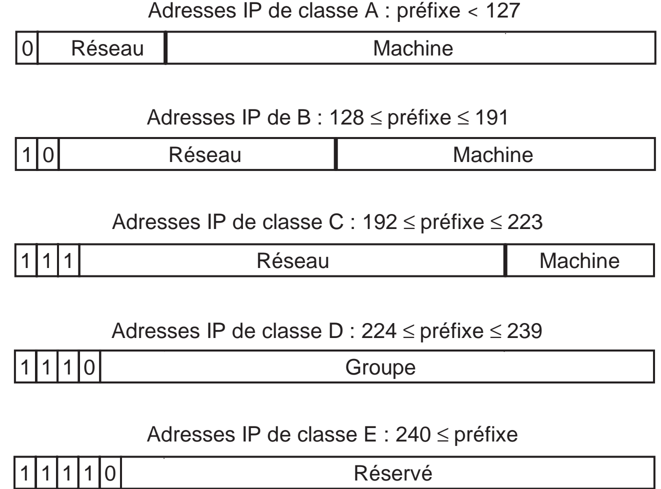


Figure 4.34 – Classes d'adresses IP.

comme le **RIPE** (***Réseaux IP Européens***) qui alloue alors des classes d'adresses aux administrateurs de réseaux.

Généralement, toutes les machines d'un même réseau local utilisent une adresse de la même classe d'adresse. En théorie, une classe d'adresse peut correspondre physiquement à un seul réseau local. Cependant, il est clair qu'il est impensable de mettre 16 millions de nœuds (cas des réseaux de classe A), voire même 65 000 (réseaux de classe B), sur un même réseau Ethernet : celui-ci s'écroulerait immédiatement sous le nombre de collisions. En pratique, une zone de collision d'un réseau Ethernet (zone de diffusion totale par *hub* par exemple) ne dépasse pas quelques dizaines ou petites centaines de nœuds (pas plus de 1 024 d'après la norme Ethernet). Donc, les réseaux de classe A et B sont souvent découpés en sous-réseaux. Pour ce découpage, on utilise un masque de sous réseau. Ce masque, de la même taille en octets que l'adresse IP, est utilisé avec un « et » binaire sur l'adresse d'un nœud. Ainsi, si un nœud possède une adresse $a_1.a_2.a_3.a_4$, et que le masque est $m_1.m_2.m_3.m_4$, alors toute machine d'adresse $a'_1.a'_2.a'_3.a'_4$ telle que $a_1.a_2.a_3.a_4$ *et* $m_1.m_2.m_3.m_4$ $= a'_1.a'_2.a'_3.a'_4$ *et* $m_1.m_2.m_3.m_4$ est censée se trouver sur le même réseau local, c'est-à-dire à vue directe. La première tâche de la couche IP est donc de déterminer si le nœud destination est à vue directe. Pour cela, IP connaît l'adresse du nœud d'émission, le masque de sous-réseau du réseau local, et l'adresse destination. Il suffit alors d'utiliser la formule « *mon adresse et masque = adresse destination et masque* » afin de déterminer si le nœud destination est directement accessible : si c'est le cas, IP obtient grâce à **ARP** (***Address Resolution Protocol***) l'adresse MAC du nœud destination, il lui suffit alors d'utiliser les services de la couche 2 pour émettre ses données, appelées **datagramme**. Si le nœud ne se trouve pas sur le même réseau, alors le datagramme est envoyé vers une machine particulière du réseau, identifiée comme passerelle ou routeur (figure 4.35), qui se charge de trouver une route jusqu'au réseau du nœud destination (généralement, le routeur est connecté à un ou plusieurs autres routeurs, le datagramme va donc traverser un certain nombre de routeurs,

© Dunod – La photocopie non autorisée est un délit.

d'abord en remontant dans la hiérarchie géographique, puis en redescendant vers la région du réseau local destination). Au final, un réseau de classe A, B, ou C ne correspond pas réellement à un réseau local : un réseau local est caractérisé par le « et binaire » entre n'importe quelle adresse de nœud a_i du réseau local et le masque de sous-réseau m_i : sur la figure 4.35, l'adresse d'un réseau local est donc donnée sous la forme $a_i\&m_i$.

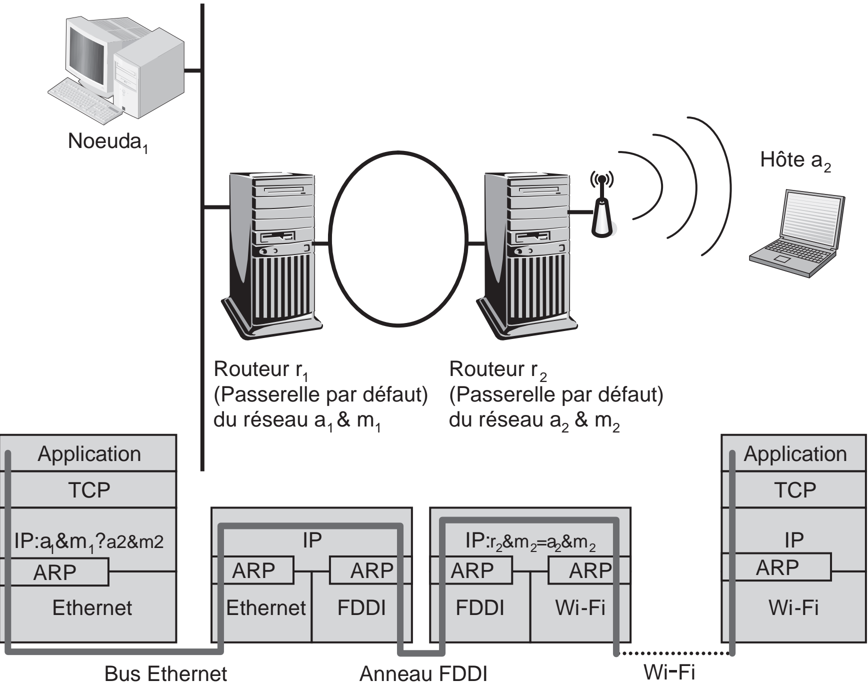


Figure 4.35 – Exemple de fonctionnement d'IP.

Au cours du trajet suivi par les données, différents supports physiques sont traversés : chacun a des contraintes sur la taille maximale des datagrammes transportés. Le datagramme original pourra donc être fragmenté pendant le voyage, puis réassemblé à l'arrivée. Il est à noter que la route empruntée par les datagrammes, ainsi que leur ordre d'arrivée et le fait même qu'ils arrivent à destination, est indéterministe. En fonction des couches physiques choisies de machine à machine, il est possible que de tel à tel nœud, la transmission soit fiable, mais que de tel à tel autre nœud, la transmission soit non fiable. Dans tous les cas, IP est un protocole réseau non fiable, puisqu'il n'y a aucune garantie d'acheminement, et généralement pas de garantie de qualité de service (vitesse de transmission…) bien que la norme IP version 6 tente d'y apporter des améliorations.

IP permet donc d'assurer le routage de datagrammes entre plusieurs réseaux hétérogènes, ceci sans aucune garantie de fiabilité. C'est pourtant à ce jour le protocole de couche réseau le plus utilisé.

Notons que beaucoup d'entreprises ou de domiciles sont équipés de réseaux locaux à **adresses IP privées**, c'est-à-dire qu'au plus un nœud, le routeur du réseau local, possède une adresse IP publique (s'il n'y a aucun nœud pouvant communiquer avec internet, on parle d'un **intranet**). Ces adresses privées sont généralement choisies dans la classe A 10.0.0.0 ou la classe B 192.168.0.0. Elles peuvent être allouées de façon fixe, ou bien de façon dynamique par exemple en utilisant le protocole **DHCP** (***Dynamic Host Configuration Protocol***), dans lequel un nœud se connectant au réseau va interroger un serveur DHCP pour obtenir dynamiquement une adresse IP, généralement privée.

☐ **Couche 4 (transport)**

La couche transport s'appuie sur la couche réseau : la couche réseau fournit des services permettant une transmission non fiable d'un nœud à un autre, quels que soient les réseaux locaux auxquels ils appartiennent. Les deux protocoles de couche transport les plus utilisés sont **TCP** (***Transmission Control Protocol***) et **UDP** (***User Datagram Protocol***).

Dans la pile TCP/IP, la couche 4 est la dernière couche avant l'application : les services de la couche transport sont donc souvent directement utilisés par les processus. Sur un nœud, plusieurs processus peuvent utiliser le réseau (figure 4.36) : par exemple, un navigateur internet, un logiciel de lecture de courrier, etc. Il faut donc distinguer les données entrantes afin de les communiquer à un processus particulier. En effet, ce sont des processus qui s'exécutent sur un nœud et qui communiquent à travers le réseau avec d'autres processus distants. L'idée est alors la suivante : il y a une adresse de niveau MAC utilisée pour la transmission physique à vue directe, une adresse IP pour la transmission de nœud à nœud, et finalement une adresse de couche transport permettant d'adresser un processus particulier sur un nœud. Pour les protocoles TCP et UDP, cette adresse est un nombre sur 16 bits appelé **port**. Un port est donc une valeur allant de 0 à 65 535, qui sert à discriminer les processus disposés à recevoir des données sur le réseau. Pour un processus, travaillant directement sur la couche TCP ou la couche UDP, le moyen de communiquer avec un processus est de connaître l'adresse IP du nœud sur lequel ce processus s'exécute, et le port utilisé par le processus. Ces informations caractérisent totalement une adresse de communication entre deux processus.

Le protocole TCP est un protocole fiable connecté, empêchant la congestion du réseau. Sa fiabilité est assurée par un système d'**accusés de réception**. Une connexion logique est constituée avant chaque échange de données. Ce protocole est asymétrique : l'un des processus se définit sur un nœud comme prêt à accepter une connexion sur un certain port : ce processus est appelé un **serveur**. Au lancement du processus, celui-ci prévient la couche TCP du système d'exploitation qu'il est à l'écoute d'un port (par exemple, c'est généralement le port 80 qui est utilisé par un serveur web). L'autre processus prendra l'initiative de demander à ouvrir une connexion avec le serveur : ce processus est alors un **client**. Sur TCP, la communication est donc de type **client/serveur**.

Ainsi, par exemple lorsqu'un processus de type navigateur internet est exécuté sur un nœud du réseau, et que l'utilisateur souhaite se connecter à un serveur web, le processus charge la couche TCP d'effectuer une connexion au port 80 du nœud

© Dunod – La photocopie non autorisée est un délit.

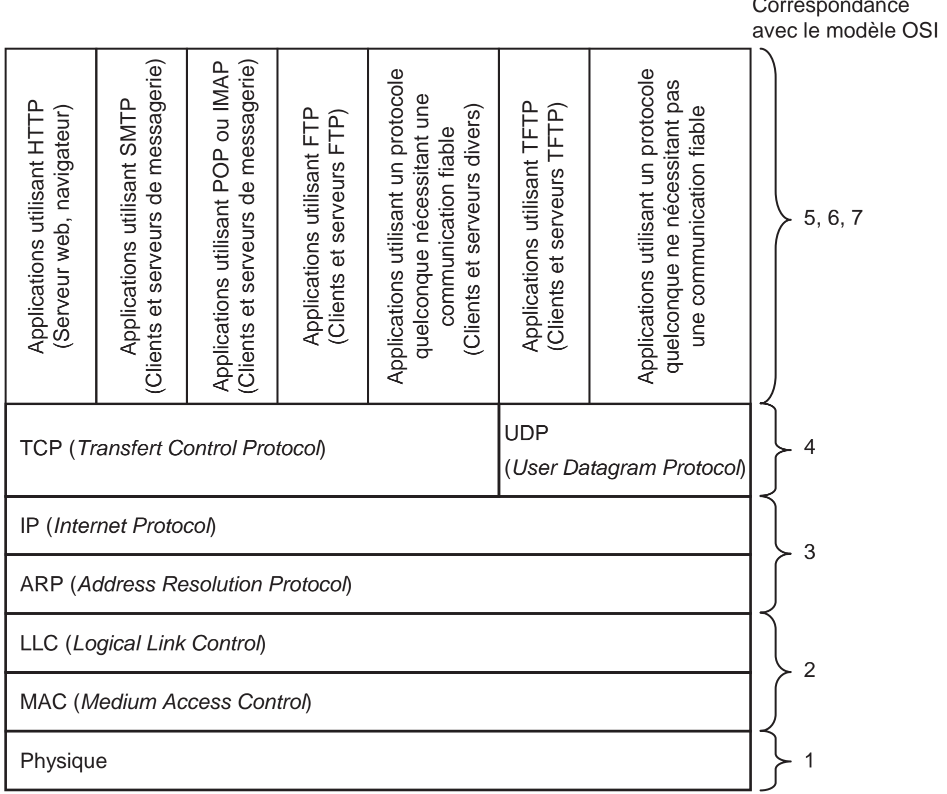


Figure 4.36 – La pile TCP/IP et UDP/IP complète.

hébergeant le serveur web voulu. Un segment de demande d'ouverture de connexion TCP, appelé SYN, est envoyé à travers les couches par la couche TCP : l'entité transmise par TCP s'appelle un **segment**. Un segment est caractérisé notamment par le port source et le port destination. Dans le cas d'une demande d'ouverture de connexion, la couche TCP peut choisir un port source disponible : il sera utilisé par le serveur web pour répondre au processus client. Le segment de demande d'ouverture de connexion est alors passé à la couche IP avec pour consigne de faire parvenir ces données (le segment passé par TCP est considéré par IP comme un ensemble de données) à l'adresse IP de destination. IP forme alors un datagramme (figure 4.29), caractérisé notamment par l'adresse IP source et l'adresse IP de destination : le champ de données du datagramme contient donc le segment TCP. Le datagramme IP est alors passé à la couche inférieure qui fait suivre aux couches jusqu'au niveau physique (voir figure 4.29) afin qu'il soit acheminé vers le nœud destination en traversant un certain nombre de routeurs (figure 4.35). Lorsque le datagramme émis arrive à la couche IP du destinataire, celle-ci lit dans l'en-tête du datagramme que c'est un segment TCP qui est contenu dans les données du datagramme, elle enlève donc l'en-tête IP et passe les données du segment (qui sont donc le segment TCP émis par la couche TCP de l'adresse source) à la couche TCP. La couche TCP constate

qu'elle a affaire à une demande d'ouverture de connexion sur un port particulier. Elle vérifie si un processus lui a signalé qu'il était à l'écoute de ce port. Si oui, elle prévient donc le processus qu'une connexion a été ouverte, cette connexion étant caractérisée par l'adresse IP et le port du client, et par l'adresse IP et le port du serveur. Le nœud contacté sait donc vers quelle adresse et sur quel port il doit répondre. La couche TCP renvoie donc un segment de demande d'ouverture de connexion, contenant en même temps un accusé de réception du segment reçu : la connexion est établie.

Par contre, si aucun processus ne s'était déclaré à la couche TCP comme étant à l'écoute du port demandé, la couche TCP aurait répondu avec un segment accusé de réception et RESET, signifiant qu'elle a bien reçu la demande d'ouverture de connexion, mais qu'elle la refuse.

Lorsqu'une connexion est ouverte, les deux processus distants, le client et le serveur, peuvent s'échanger des données de façon fiable. En effet, une fois la connexion établie, ils peuvent communiquer de la façon suivante : lorsque l'un des processus veut envoyer des données, il passe ces données sous forme d'octets à la couche TCP en lui demandant de transmettre ces données sur la connexion ouverte. Les données sont alors encapsulées dans un segment TCP (ajout d'un en-tête avec port source et destination), puis passées à la couche IP, etc. jusqu'à arriver au processus distant, qui peut alors les lire. Si nécessaire, la couche TCP peut fractionner les données (si elles sont trop grandes pour être mises dans un datagramme IP, caractérisé par une **MTU**, ***Maximal Transfert Unit***, donnant la taille maximale d'un datagramme sur le réseau : par exemple, 1 500 octets pour Ethernet, 4 470 octets pour FDDI), qui seront alors réassemblés par la couche TCP destination, avant d'être transmises au processus (n'oublions pas que les couches IP des routeurs peuvent elles aussi fractionner un datagramme transmis, qui dans ce cas sera réassemblé par la couche IP destination, avant d'être passé à la couche TCP).

Chaque segment transmis se voit répondre un accusé de réception. Si au bout d'un certain temps, la couche TCP n'a pas reçu d'accusé de réception pour un segment transmis, elle décide de le ré-émettre. Au bout d'un certain nombre de ré-émissions, la couche considère la connexion perdue.

La fermeture d'une connexion ne déroge pas à la règle de l'accusé de réception : envoi d'un segment FIN par l'une des couches TCP, réponse par un FIN, ACK de la couche distante. La connexion est alors fermée de façon bilatérale.

La décongestion du réseau, quant à elle, est assurée de la façon suivante : supposons que deux machines doivent échanger un gros volume de données. Un processus transmet donc à la couche TCP un nombre important d'octets, qui utiliseront plusieurs segments. Si tous les segments étaient envoyés dès la possibilité physique d'émettre sur le réseau local, les routeurs se retrouveraient saturés, leur mémoire tampon, contenant des messages à faire passer d'un réseau à un autre de débit plus faible, serait remplie, et ils ne pourraient plus prendre en compte les nouvelles données, qui seraient perdues. Les données perdues seraient alors ré-émises, ce qui saturerait encore et encore les routeurs, qui s'écrouleraient comme des châteaux de cartes. Afin d'éviter cela, TCP utilise une fenêtre glissante : le principe est qu'il existe une taille de fenêtre f, configurable, correspondant par exemple à 4 segments TCP de taille maximale. La règle est que la couche TCP ne doit pas émettre plus de f segments

© Dunod – La photocopie non autorisée est un délit.

sans avoir encore reçu d'accusés de réception. Par exemple, avec une fenêtre équivalente à 4 segments, la couche TCP transmet sur la connexion 4 segments, dont elle attend les accusés de réception. Dès qu'un accusé de réception arrive, elle n'attend plus que 3 accusés de réception, elle peut alors émettre un segment supplémentaire, ce qui la met en attente de 4 accusés de réception.

Pour une application, le protocole TCP/IP est donc extrêmement intéressant, car il repose sur une transmission fiable, bien que non déterministe en temps. Sa programmation est très simple, car tous les langages de programmation proposent des bibliothèques de fonctions pour programmer TCP/IP :

– l'application serveur prévient la couche TCP qu'elle désire écouter un certain port, puis se met en attente sur ce port, l'application se retrouve alors dans l'état *bloqué*, et repassera dans l'état *prêt* lorsqu'une connexion aura été établie à l'initiative d'un client ;

– l'application client doit demander à ouvrir une connexion TCP en fournissant l'adresse IP du serveur (abus de langage signifiant l'adresse IP du nœud sur lequel l'application serveur est susceptible de s'exécuter), ainsi que le port utilisé par celui-ci ;

– une connexion TCP est alors ouverte. On peut y écrire des données, comme on pourrait les écrire à l'écran ou dans un fichier : les données écrites sont transférées à la couche TCP qui se charge de les faire parvenir à destination. Notons que l'écriture est doublement suspensive, d'une part, localement, comme toute entrée/sortie, mais aussi à cause du fait que l'écriture de données est considérée comme effectuée lorsque les accusés de réception ont été reçus. On peut lire des données sur une connexion comme on pourrait lire des données en provenance d'un clavier : lire des données consiste soit à piocher des données dans une zone tampon des données reçues sur la connexion si des données sont présentes, soit à les attendre si elles ne sont pas arrivées, dans ce cas, la lecture est bloquante ;

– un protocole d'application doit bien sûr avoir été préalablement défini, permettant d'établir comment les applications pourront se comprendre (voir un exemple au chapitre 7).

On peut remarquer qu'un seul processus peut utiliser un port, ce qui fait que pour éviter d'empiéter sur les ports utilisés par certaines applications serveur standard (serveur HTTP web écoutant par défaut le port 80, serveur de fichiers FTP écoutant par défaut le port 21, serveur d'envoi de courrier électronique SMTP écoutant le port 25, etc.), les applications utilisateurs choisiront des numéros de ports élevés.

Le protocole UDP, quant à lui, est nettement plus simple, puisque c'est une version non fiable et non connectée d'un protocole de couche transport. Cependant, il est beaucoup moins utile aux applications de contrôle-commande, car il est gênant de ne pas savoir si les données transmises ont bien été reçues.

■ Réseaux de terrain

Les **réseaux de terrain** sont des réseaux de communication dédiés au contrôle de procédés. Les besoins sont différents par rapport aux réseaux généralistes :

– les volumes de données transférés sont moins importants (donnés sur l'état du procédé, informations provenant de capteurs ou commandes vers des actionneurs, commande de supervision…) ;
– les contraintes de temps dues aux aspects temps réel du contrôle nécessitent des modes d'accès déterministes au médium ;
– certains messages étant plus importants que d'autres, des mécanismes de priorité doivent être introduits ;
– face à un environnement de terrain, le matériel utilisé doit être robuste ;
– étant donnée la criticité des données transmises, le contrôle d'erreur doit être important ;
– les nœuds connectés par un réseau de terrain sont souvent des calculateurs ayant moins de puissance de calcul qu'un ordinateur, il faut donc des protocoles relativement simples ;
– généralement, un réseau de terrain est dédié à une et une seule application (une application de contrôle-commande et ou de supervision/suivi de production…). Il est alors inutile de mettre en œuvre les couches hautes du modèle OSI (établissement d'une connexion TCP entre deux processus par exemple), connexion logicielle TCP… Cependant, il est intéressant d'utiliser des connexions de bas niveau (par exemple LLC) avec un contrôle d'erreurs de bas niveau, et pourquoi pas un mécanisme d'accusés de réception de niveau LLC ;
– en général, les nœuds devant communiquer sont à vue directe sur le réseau, la plupart des protocoles de réseaux locaux s'arrêtent donc souvent à la couche 2. Cependant il existe pour la plupart d'entre eux des nœuds routeurs capables de faire le lien entre différents réseaux de terrain et des réseaux généralistes utilisant TCP/IP.

□ Le protocole CAN

Le protocole **CAN** (***Controller Area Network***), est un réseau de terrain très utilisé en milieu industriel notamment dans le monde automobile (figure 4.37). De nombreux microcontrôleurs du marché intègrent une interface de communication CAN. Ce protocole est à 2 couches (physique, MAC/LLC) et se base sur de la diffusion (qu'elle soit sur bus, ondes radio ou lumineuses) de bits en série (comme le protocole Ethernet), mais de façon synchrone. La seule contrainte sur le médium est qu'il permette un « et » logique matériel sur les bits transmis : si un nœud émet 0, et qu'un autre émet simultanément 1, alors le médium doit transporter 0. Si tous les nœuds émettent 1, le médium transporte 1.
L'émission synchrone est fondamentalement différente de l'émission asynchrone proposée par la plupart des réseaux généralistes : un signal de synchronisation passe périodiquement sur le réseau, un top qui indique aux nœuds qu'ils peuvent, s'ils le désirent, émettre des données. Le synchronisme est très pratique, puisqu'il existe alors un temps commun à tous les nœuds. Cependant, il a un coût en bande passante important : en effet, chaque bit a une durée égale à deux fois la durée de propagation d'un signal de bout en bout sur le médium : cela correspond à la tranche canal présentée pour CSMA/CD.

© Dunod – La photocopie non autorisée est un délit.

Il n'y a pas d'adresse de nœud pour CAN, et la trame CAN ne transporte pas d'informations sur la source ou la destination d'un message. Par contre, un message est caractérisé par un identifiant (ID) : un ID est un nombre codé sur 11 bits (CAN 2.0A) ou 29 bits (CAN 2.0B) permettant de transporter en même temps l'identifiant et la priorité du message. Plus le nombre est faible, plus la priorité est élevée. Ainsi, le concepteur d'une application utilisant CAN attribue une priorité aux messages, qui vont typiquement encapsuler des données capteur ou actionneur, et un nœud observant un message passer sur le réseau ne le lit que si l'ID l'intéresse. C'est une philosophie totalement différente de la philosophie client/serveur : en effet, étant donné qu'une application utilisant un réseau de terrain est déterminée à l'avance, le concepteur sait quels types de messages chaque nœud doit recevoir.

En théorie, le principe est le suivant : les nœuds synchronisent leur horloge sur les messages émis. Lorsque le médium est disponible, une station commence par émettre un bit *Start Of Frame* à 0 suivi de l'ID (priorité) du message en commençant par les bits de poids fort. À chaque bit émis, elle écoute le réseau (elle doit donc attendre un temps équivalent à la tranche canal), afin de vérifier que ce qu'elle lit est ce qu'elle a émis. À chaque bit de l'ID ainsi émis, la station écoute le médium afin de déterminer si elle peut continuer à émettre : si la station a émis un 0 (bit dominant), elle est sûre de pouvoir continuer à émettre, cependant, elle ne peut pas savoir s'il existe un autre nœud ayant commencé en même temps qu'elle à émettre (à une durée de propagation près sur le réseau) qui émet elle aussi un 0. Par contre, si le nœud émet un 1 (bit récessif), et lit un zéro, cela signifie qu'un autre nœud ayant commencé à transmettre au même moment a un numéro d'ID plus petit, donc plus prioritaire (le médium assure un « et » physique entre les bits transmis). Dans ce cas, le nœud stoppe sa transmission et tentera de ré-émettre son message à la fin de la transmission.

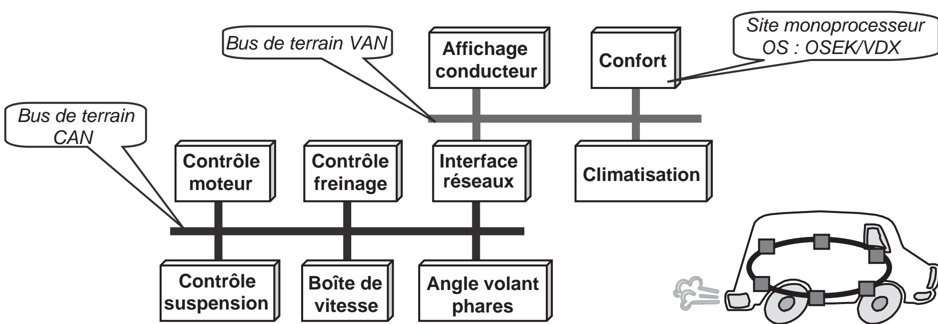


Figure 4.37 – Exemple de réseau de terrain CAN.

Dans le cas où un nœud a passé entièrement son ID, il peut alors continuer à émettre sa trame (figure 4.38). Pour augmenter la robustesse du protocole, celui-ci intègre au niveau 2 un mécanisme d'accusés de réception, et les nœuds qui prennent en compte un message envoient un accusé de réception : cela consiste à positionner le 1er bit du champ ACK à 0. Étant donné que la transmission est synchrone, le nœud

SOF	Identifier (ID)	RTR	IDE	r_0	DLC	Données	CRC	ACK	End of Frame
1 bit	11 bits	1 bit	1 bit	1 bit	4 bits	0 à 8 octets	15 bits	2 bits	7 bits

Trame CAN 2.0A

SOF	Identifier (ID)	SRR	IDE	Identifier long	RTR	r_1	r_0	DLC	Données	CRC	ACK	End of Frame
1 bit	11 bits	1 bit	1 bit	18 bits	1 bit	1 bit	1 bit	4 bits	0 à 8 octets	15 bits	2 bits	7 bits

Trame CAN 2.0B

SOF : *Start Of Frame*, bit dominant
SRR : *Substitute Remote Request* bit récessif tel que si deux messages de même ID 11 bits sont émis, le format 2.0A est plus prioritaire
RTR : *Remote Transmission Request*, détermine si c'est une trame donnée ou demande de donnée
IDE : *IDentifier Extension* détermine si c'est une trame 2.0A ou 2.0B
DLC : *Data Length Code*, taille du champs données
ACK : champs accusé de réception, l'émetteur envoie 2 bits recessifs, le premier est remplacé par un bit dominant par les récepteurs de la trame. L'émetteur voit donc au retour de sa propre trame l'A/R
CRC : *Cyclic Redundancy Check* permet de détecter les erreurs de transmission
r_0, r_1 : bits réservés pour le futur

Figure 4.38 – Format de trame CAN.

émettant la trame se rend compte si sa trame a été reçue par au moins un nœud, sans identifier lequel.

Ce protocole entre dans la catégorie des CSMA/CA, car il évite les collisions. Ce protocole est très intéressant pour les applications temps réel, puisque les messages les plus prioritaires (donc généralement les plus critiques) sont certains de pouvoir passer prioritairement par rapport aux autres messages sur le réseau : au pire lorsqu'un nœud veut émettre un message, il doit attendre la fin du message courant, et les éventuels messages plus prioritaires.

De plus, le protocole CAN est très résistant aux fautes : différents niveaux de contrôle d'erreur ont été introduits :

– contrôle **CRC** (***Cyclic Redundancy Check***) à la fin de chaque trame qui permet de vérifier les données ;
– contrôle de la forme de la trame au niveau physique ;
– accusé de réception de chaque trame reçue.

L'avantage principal de CAN dans les applications temps réel est qu'il est possible d'étudier *a priori* l'ordonnancement des messages sur le réseau : en effet, le réseau peut être vu comme un processeur géré de façon non préemptive (un message ne peut pas être interrompu) mais utilisant un ordonnancement basé sur la priorité des messages, puisqu'à la fin d'un traitement, c'est le message le plus prioritaire qui est émis.

La limitation principale de CAN est liée à son mode synchrone d'émission limitant fortement les débits ou longueurs de médiums de communication. En effet, sur un bus, un signal se propage à environ 200 000 km/s. Ce qui signifie que si le médium mesure n mètres, il faudra attendre $2n$ m/2×10^9 m/s $= n/10^9$ secondes, soit $n \times 10^{-9}$ secondes. Ce qui donne un débit théorique maximal de $10^9/n$ bits/s sur un bus de n mètres. Pour un bus de 100 m, le débit maximal pourrait être de 1 Mbits/s. Cependant, en pratique, ce débit est atteint pour des bus d'une longueur d'environ 40 m.

© Dunod – La photocopie non autorisée est un délit.

Le protocole FIP

WorldFIP est une autre illustration de réseau de terrain très utilisé dans l'industrie. Comme CAN, il permet de faire communiquer différents nœuds, certains nœuds étant spécialisés dans des acquisitions de capteurs ou dans des commandes d'actionneurs. Le protocole a une philosophie très particulière : son but, comme celui de CAN, est de faire communiquer des nœuds participant à la même application. Cependant il se base sur 3 types de messages : variables transmises périodiquement (rafraîchissement périodique), variables transmises de façon apériodique, et messages transmis de façon apériodique. Le protocole se base sur un arbitre gérant l'accès au médium (la redondance des arbitres permet d'augmenter la robustesse) en se basant sur un cycle (20 ms par exemple). L'arbitre sait à quel moment les variables périodiques doivent être émises sur le réseau, il envoie donc une demande pour chaque variable périodique à intervalle régulier. Un seul nœud étant producteur pour une variable, ce nœud doit alors envoyer la valeur de la variable, qui pourra être lue par les nœuds consommateurs de cette variable. Le reste du temps est accordé au trafic apériodique (valeurs de variables, messages, ou informations de configuration). Les débits atteints par les réseaux de terrain WorldFIP peuvent atteindre 25 Mbits/s sur de faibles distances (quelques dizaines de mètres), et de nombreux équipements existent. Il est ainsi possible de relier par un routeur World/FIP-Ethernet un réseau de terrain à un réseau généraliste (qui bien sûr n'est pas déterministe).
Le protocole FIP est composé des couches 1, 2 et 7. La couche application permet notamment aux applications de s'assurer de la cohérence temporelle et spatiale des variables reçues via le réseau. La cohérence temporelle consiste à s'assurer qu'une variable est fraîche, c'est-à-dire qu'elle a été lue dans le même cycle temporel. La cohérence spatiale permet à un nœud de savoir si les autres nœuds consommant les mêmes variables que lui ont des valeurs identiques pour ces variables.

5 • EXÉCUTIFS TEMPS RÉEL

Un **exécutif temps réel** peut être employé à la place d'un système d'exploitation généraliste par un système de contrôle-commande lorsqu'il est soumis à des contraintes de temps, ou bien lorsqu'il doit être embarqué sur un microcontrôleur.

5.1 Introduction

5.1.1 Limitations des systèmes d'exploitation généralistes

Lorsqu'une application de contrôle-commande est soumise à de fortes contraintes de temps, l'utilisation d'un système d'exploitation généraliste est inadaptée, car leurs objectifs sont les suivants :

- garantir l'**indépendance des processus** et les protéger les uns vis-à-vis des autres notamment grâce à la MMU (*Memory Management Unit*), ce qui implique une mise en œuvre lourde des communications entre les processus ;
- augmenter la vitesse moyenne de traitement à l'aide d'**optimisations locales** : utilisation d'un cache disque permettant un accès asynchrone, mais non prédictible, aux mémoires de masse ; utilisation des optimisations des processeurs tels les *pipelines* et la mémoire cache **nuisant à la prédictibilité** du temps d'exécution (les optimisations processeur peuvent être utilisées par les noyaux temps réel aussi) ;
- limiter le surcoût processeur dû au système d'exploitation, ce qui conduit souvent à une **gestion grossière du temps** : l'unité de temps typique est la milliseconde ou la dizaine de millisecondes ;
- favoriser l'**équité** dans l'ordonnancement au détriment du **temps de réponse**, utilisant des quanta de temps de l'ordre d'une dizaine de millisecondes ;
- faciliter la maintenance du système en offrant de **nombreux processus de maintenance** en concurrence avec les processus des utilisateurs, ce qui augmente la charge processeur et mémoire du système ;
- rendre transparente l'utilisation de la mémoire, en utilisant la **mémoire virtuelle**, ce qui **nuit à la prédictibilité**.

De plus, pour arriver à ces buts, les systèmes d'exploitation généralistes sont le plus souvent **monolithiques**. En d'autres termes, le système d'exploitation est muni d'un **noyau** de base permettant la gestion des processus, leur ordonnancement et la gestion de la mémoire et de la mémoire virtuelle. Les modules spécifiques liés à la gestion

© Dunod – La photocopie non autorisée est un délit.

du matériel (entrées/sorties, réseau, systèmes de gestion de fichiers, etc.) sont ajoutés au noyau pour offrir des services de plus haut niveau à l'utilisateur.

L'ensemble regroupant le noyau et les modules est souvent appelé un **exécutif** : toute ou partie de cet exécutif est non préemptible, c'est-à-dire que lorsque certains services s'exécutent, un processus utilisateur, ou même un autre service, ne peut pas être exécuté. Si l'exécution de ces services nécessite quelques dizaines de millisecondes, le contrôle très fin, à quelques microsecondes près d'un procédé relève de l'impossible. Le temps mis par le système d'exploitation à prendre en compte un événement (typiquement une interruption) en lançant la routine de traitement d'interruption liée s'appelle la **latence due au noyau**. Par exemple, lorsqu'un système d'exploitation généraliste est en surcharge (calcul intensif, combiné à de nombreuses entrées/sorties, notamment écriture sur les disques durs), on peut atteindre une latence de plusieurs centaines de millisecondes, ce qui est inacceptable pour un système de contrôle-commande fin.

Enfin, une structure monolithique est moins **robuste** qu'une structure en tâches de service, dans laquelle un noyau (figure 5.1) gère les processus ou les tâches, et où les services de l'exécutif sont fournis par des tâches ou processus situés au niveau utilisateur.

En cas d'erreur matérielle dans une structure en tâches de service, le processus ou la tâche de service liée au dispositif matériel s'arrête, mais le système peut continuer à s'exécuter, et les services qu'il fournissait sont alors indisponibles. Dans le cas d'un système monolithique, le système est arrêté. Notons cependant que généralement, une structure monolithique est plus intégrée, les services de l'exécutif sont alors plus performants en moyenne.

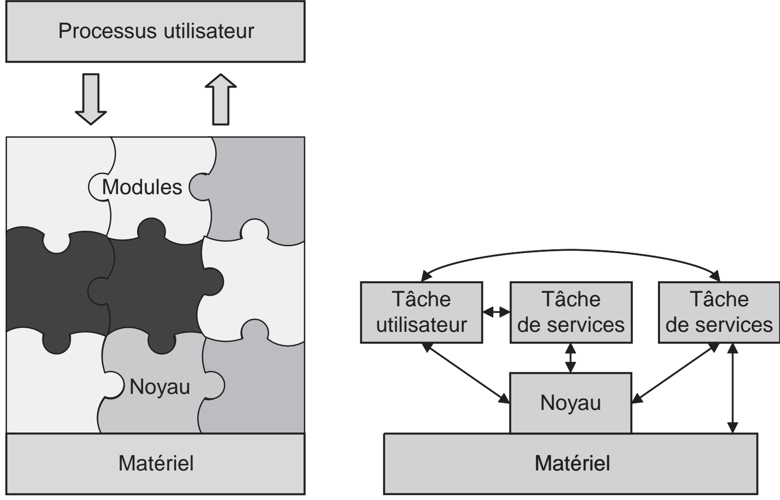


Figure 5.1 – Structure monolithique contre structure en tâches de service.

5.1.2 Noyau, exécutif et système d'exploitation temps réel

Le concept d'indépendance des processus assuré par un système d'exploitation généraliste augmente la complexité des primitives de communication entre processus. Or, nous avons vu avec la méthode DARTS qu'il était nécessaire que les éléments constituant un système de contrôle-commande communiquent. Par conséquent, la plupart des systèmes de contrôle-commande et des systèmes temps réel sont implémentés en utilisant des **tâches**. Les tâches, de même que les processus, concourent à l'obtention du ou des processeurs, et peuvent communiquer ou se synchroniser, mais *a contrario* des processus, c'est le même espace mémoire qu'ils partagent. Une tâche est donc caractérisée par une pile et un pointeur d'instruction, mais partage la mémoire avec les autres tâches du même processus.

Les **noyaux temps réel** assurent la gestion de tâches (ordonnancement, outils de synchronisation et de communication, gestion des interruptions) et de la mémoire. Ils sont appelés **micronoyaux** quand ils ont une faible **empreinte mémoire** (taille mémoire nécessaire à leur exécution). Un noyau temps réel est apte à être embarqué sur un microcontrôleur ou un microprocesseur disposant de peu de mémoire.

Un **exécutif** temps réel est une surcouche du noyau : il offre les services nécessaires à l'utilisation des entrées/sorties, du réseau, des *timers*, des fichiers, etc. Le plus souvent, ces services sont modulaires, et il est possible de configurer spécifiquement un exécutif en fonction de l'architecture utilisée. Ces modules peuvent être configurés de façon monolithique ou sous forme de tâches de service.

Un **système d'exploitation** temps réel est un système d'exploitation complet reposant sur un exécutif temps réel.

En fonction de l'**architecture cible** (architecture sur laquelle le système de contrôle-commande s'exécute), un concepteur doit choisir entre système d'exploitation et exécutif (il est rare que l'on puisse se contenter d'un noyau seul). Pour cela, le principal critère est l'empreinte mémoire. Un noyau utilise quelques kilo-octets, et chaque module de l'exécutif utilise quelques octets à quelques kilo-octets, alors qu'un système d'exploitation nécessite plusieurs méga-octets de mémoire.

Comme nous l'avons vu dans le chapitre 1, le principal inconvénient de l'utilisation d'un exécutif temps réel est la relative complexité de programmation. En effet, programmer sur un exécutif nécessite un **développement croisé** : un ordinateur, appelé **station de développement**, sert à développer le code. On utilise alors un compilateur croisé, afin de générer un code exécutable qui sera chargé sur la **cible** en vue de l'exécution. Généralement, les environnements de développement croisé fournissent des outils d'observation et de débogage de la cible. Malgré la présence de ces outils, un développement sur exécutif est généralement beaucoup plus long et laborieux qu'un développement sur système d'exploitation, ce qui va dans le sens de la justification de l'utilisation d'un cycle en W (le premier V, sur simulateur, est généralement exécuté sur un système d'exploitation, ce qui permet de réduire le temps de mise au point du second V qui a lieu sur la cible).

Les architectures matérielles étant très évolutives, la plupart des exécutifs temps réel disponibles supportent plusieurs processeurs/microcontrôleurs équipés de cartes de développement. Un ensemble d'outils de développement pour un exécutif temps réel est donc typiquement composé d'une base commune indépendante du support

© Dunod – La photocopie non autorisée est un délit.

matériel, et d'un ensemble d'outils et librairies spécifiques à une cible. Cet ensemble d'outils est regroupé sous la forme d'un ensemble de pilotes de périphériques, de fonctions spécifiques (gestion de la mémoire, accès aux matériels spécifiques, etc.) nommé **BSP** (***Board Support Package***).
En général, les exécutifs temps réel sont moins ouverts, moins flexibles, et optimisent moins la vitesse moyenne des traitements que les systèmes d'exploitation généralistes, mais ils favorisent la prédictibilité temporelle.

5.2 Concepts des exécutifs temps réel

Les langages de programmation utilisés pour développer une application s'appuient sur les services fournis par l'exécutif temps réel afin de gérer des tâches, de les faire communiquer, se synchroniser, de gérer le temps, de traiter les interruptions matérielles. Ces services influencent l'état des tâches, sur lequel s'appuie l'ordonnanceur afin de gérer l'exécution du système.

5.2.1 Gestion des tâches

Les noyaux temps réel gèrent les tâches (figure 5.2) suivant le même principe que les systèmes d'exploitation généralistes. Cependant, étant donné que les tâches sont gérées finement, certains noyaux distinguent la création et l'initialisation d'une tâche.
Une tâche est initialement ***créée***, ainsi, elle devient ***existante*** mais non initialisée. Elle doit être ***initialisée***, ce qui la met dans l'état ***prête***. C'est dans cet état qu'elle requiert un processeur. Lorsque l'ordonnanceur le décide, suivant la politique d'ordonnancement choisie, cette tâche se voit allouer un processeur afin d'être ***exécutée***.
De l'état *exécutée*, une tâche peut être ***préemptée***, ou bien se ***bloquer*** en attendant un message, un événement, ou bien l'accès à une ressource. Lorsqu'elle se met en attente pendant un certain temps ou bien jusqu'à une certaine date, on dit qu'elle est ***endormie*** (notons que les états *bloquée* et *endormie* sont sémantiquement très proches, et qu'ils peuvent être confondus).
Une tâche peut généralement être ***supprimée*** à partir de tout état. Afin de prévenir la perte d'une ressource suite à la suppression d'une tâche la détenant, certains exécutifs proposent des primitives de protection contre la suppression.
Il est à noter que de nombreux noyaux temps réel proposent l'état supplémentaire ***suspendue*** qui peut être atteint depuis n'importe quel autre état, et qui interdit à une tâche d'être exécutée jusqu'à sa reprise.

5.2.2 Outils de communication et de synchronisation

Ce paragraphe présente les outils de communication et de synchronisation permettant aux tâches d'interagir. Auparavant, nous revenons sur les outils de base que sont les sémaphores et les moniteurs, qui seront utilisés par la suite pour donner une implémentation d'outils de communication et de synchronisation, ainsi que sur les variables conditionnelles. En effet, tous les outils ne sont pas nécessairement proposés par les exécutifs, et le concepteur d'une application doit parfois lui-même les implémenter, en utilisant les outils de base.

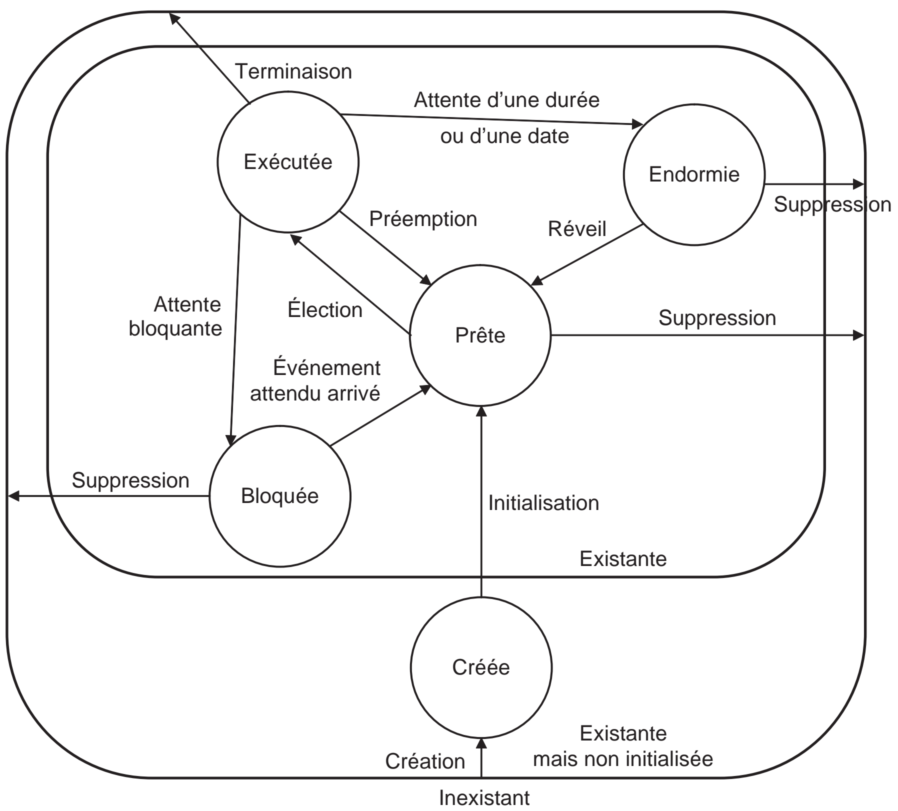


Figure 5.2 – Graphe simplifié des états possibles des tâches gérées par un noyau temps réel.

■ Éléments de base

□ Sémaphores

Nous avons présenté au chapitre 4 l'outil sémaphore : un sémaphore est une variable soit binaire (deux états sont possibles : libre et pris), soit *n*-aire. Dans le cas des sémaphores *n*-aires, appelés **sémaphores à compte**, la valeur 0 indique que le sémaphore est pris, et une valeur différente de zéro indique qu'il y a un certain nombre d'instances libres.

Un **sémaphore binaire** (figure 5.3a) est caractérisé par une valeur booléenne et une file d'attente de tâches en attente du sémaphore. La file d'attente peut être gérée, en fonction de l'exécutif sous-jacent, soit de façon FIFO, soit sous la forme de plusieurs files FIFO gérées par priorité des tâches désirant accéder au sémaphore. Souvent appelé ***mutex*** (*mutual exclusion*) lorsqu'il est utilisé pour assurer l'exclusion mutuelle, le sémaphore binaire peut être couplé, suivant les exécutifs, avec un protocole à priorité héritée ou bien un protocole à priorité plafond afin d'éviter le phénomène d'inversion de priorité (voir § 5.2.4). Dans le cas du protocole à priorité

© Dunod – La photocopie non autorisée est un délit.

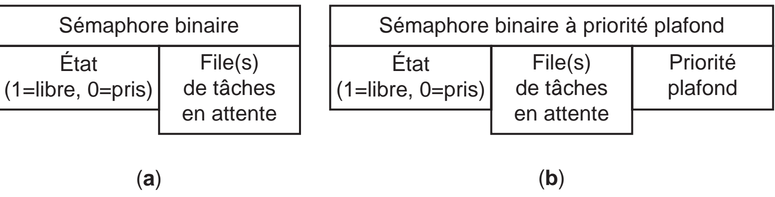


Figure 5.3 – Caractérisation d'un sémaphore binaire :
(a) sans protocole de gestion de ressource ou avec protocole à priorité héritée,
(b) avec protocole à priorité plafond.

plafond, il est nécessaire de caractériser le sémaphore par une priorité plafond (figure 5.3b), c'est-à-dire la plus grande priorité parmi les tâches susceptibles de l'utiliser.

Un **sémaphore à compte** ou sémaphore compteur (figure 5.4a) peut être utilisé pour garantir une exclusion mutuelle lors de l'accès à une ressource multi-instance, ou encore permettre un accès de type lecteur/écrivain à une ressource, ou bien encore pour effectuer une synchronisation à compte (voir § 5.2.2, p. 201). Dans les deux premiers cas, comme le *mutex*, il peut être couplé à un protocole de gestion de ressource (figure 5.4b).

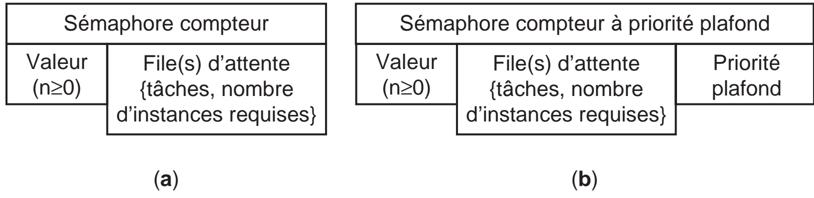


Figure 5.4 – Caractérisation d'un sémaphore compteur :
(a) sans protocole de gestion de ressource ou avec protocole à priorité héritée,
(b) avec protocole à priorité plafond.

Notons enfin qu'il existe une variante spécifique du sémaphore dans la norme POSIX (voir § 5.3.1, p. 219) : le *rwlock* (sémaphore d'accès de type lecteur/écrivain), qui sera abordé dans le cadre de la description de la norme.

☐ Moniteurs

Nous avons vu au chapitre 4 que le moniteur était un outil strictement plus puissant que le sémaphore. Il existe deux types de moniteur :

– Le moniteur classique, appelé **moniteur de Hoare**, dans lequel la non réentrance des primitives est couplée avec un mécanisme d'attente (*wait*) et de réveil (*signal*) permettant de mettre en attente ou réveiller une tâche accédant au moniteur sous certaines conditions.
– Le **moniteur à la Ada** (**objet protégé**) permet de mettre une garde (barrière logique) à l'entrée de chaque primitive du moniteur, ce qui permet une gestion

plus fine et plus transparente des conditions d'accès que dans le cas du moniteur de Hoare. De plus, cette gestion est plus efficace dans le cas où il y a plusieurs conditions de réveil : seule la tâche concernée est réveillée par le changement d'état du moniteur, contrairement au cas du moniteur de Hoare dans lequel toutes les tâches en attente sur *wait* sont réveillées sur un *signal* et doivent alors évaluer leur condition afin de se remettre en attente, ou de poursuivre leur exécution. De plus, le moniteur à la Ada possède une distinction native accès en lecture/accès en écriture, ce qui permet à plusieurs tâches d'accéder au moniteur en lecture en même temps.

Afin d'illustrer la différence existant entre les deux types de moniteur, considérons un problème simple : l'implémentation d'un double sémaphore (ensemble de deux sémaphores, pouvant être requis simultanément, par exemple « réseau » et « port série »). Comme cela a été dit au chapitre 4, ce type d'exemple ne doit être vu que comme un exemple purement académique.
L'implémentation illustrative proposée possède 6 primitives :

– *prendre_réseau* prend le sémaphore « réseau » ;
– *vendre_réseau* vend le sémaphore « réseau » ;
– *prendre_rs232* prend le sémaphore « série » ;
– *vendre_rs232* vend le sémaphore « série » ;
– *prendre_tout* prend les deux sémaphores ;
– *vendre_tout* vend les deux sémaphores.

Deux variables booléennes internes (*réseau_libre* et *rs232_libre*) donnent l'état des sémaphores.
L'implémentation d'un tel moniteur à la Hoare est donnée ci-dessous :

```
Moniteur Communication :
réseau_libre, rs232_libre : booléen := vrai
Procédure prendre_réseau
Début
      tant que réseau_libre = faux faire
      -- on attend que le réseau soit libre
            wait
      fait
      réseau_libre := faux
Fin
Procédure vendre_réseau
Début
      réseau_libre := vrai
      signal
Fin
Procédure prendre_rs232
Début
      tant que rs232_libre = faux faire
      -- on attend que le port série soit libre
            wait
      fait
      rs232_libre := faux
Fin
```

© Dunod – La photocopie non autorisée est un délit.

```
Procédure vendre_rs232
Début
      rs232_libre := vrai
      signal
Fin
Procédure prendre_tout
Début
      tant que réseau_libre = faux faire
      -- on attend que le réseau soit libre
            wait
      fait
      tant que rs232_libre = faux faire
      -- on attend que le port série soit libre
            wait
      fait
      réseau_libre := faux
      rs232_libre := faux
Fin
Procédure vendre_tout
Début
      réseau_libre := vrai
      rs232_libre := vrai
      signal
Fin
```

Nous pouvons remarquer que lorsque par exemple une tâche libère le réseau, les éventuelles tâches en attente sur le port série sont réveillées pour rien. Le moniteur à la Ada évite cet écueil en introduisant des **gardes** (conditions d'entrée) à l'entrée des primitives :

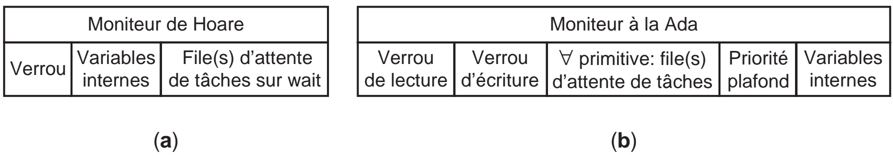
```
Moniteur Communication :
réseau_libre, rs232_libre : booléen := vrai
Procédure prendre_réseau quand réseau_libre
-- Lorsqu'une tâche entre ici, le réseau est forcément libre
Début
      réseau_libre := faux
Fin
Procédure vendre_réseau
Début
      réseau_libre := vrai
Fin
Procédure prendre_rs232 quand rs232_libre
-- Lorsqu'une tâche entre ici, le port série est forcément libre
Début
      rs232_libre := faux
Fin
Procédure vendre_rs232
Début
      rs232_libre := vrai
Fin
Procédure prendre_tout quand réseau_libre et rs232_libre
-- Lorsqu'une tâche entre ici, toutes les ressources sont libres
Début
      réseau_libre := faux
      rs232_libre := faux
Fin
```

```
Procédure vendre_tout
Début
      réseau_libre := vrai
      rs232_libre := vrai
Fin
```

On pourra apprécier la facilité de programmation du moniteur à la Ada par rapport à celle du moniteur de Hoare, présent notamment dans la norme POSIX et Java. Notons que le protocole à priorité plafond existe pour le moniteur à la Ada.

Le moniteur de Hoare, qui ne distingue pas les conditions d'attente, peut utiliser une seule file FIFO (ou un ensemble de files FIFO classées par priorité de tâches). Le moniteur à la Ada possède une file (ou un ensemble de files) par primitive d'accès, la discrimination de la condition d'accès étant effectuée au niveau de chaque primitive.

La différence de confort de programmation se ressent donc dans la caractérisation des deux types de moniteurs (figure 5.5).



Moniteur de Hoare		
Verrou	Variables internes	File(s) d'attente de tâches sur wait

Moniteur à la Ada				
Verrou de lecture	Verrou d'écriture	∀ primitive: file(s) d'attente de tâches	Priorité plafond	Variables internes

(**a**) (**b**)

Figure 5.5 – Caractérisation (a) d'un moniteur de Hoare, (b) d'un moniteur à la Ada.

□ Variables conditionnelles

Peu d'exécutifs et de langages de programmation proposent nativement le moniteur (excepté Ada à partir de la version 95). La primitive *wait* d'un moniteur de Hoare consiste, de façon atomique (non préemptible), à libérer le verrou du moniteur et à placer la tâche dans l'état *bloquée* en attente d'un *signal* dans le moniteur, puis au moment du *signal*, réveil de la tâche et prise du verrou. Ce fonctionnement n'est pas implémentable par sémaphore, et nécessiterait une portion de code non préemptible. De nombreux exécutifs proposent donc un outil appelé **variable conditionnelle** (figure 5.6) effectuant lors d'un *wait*, de façon atomique, : libération du verrou, passage de la tâche à l'état *bloquée* en attente de la variable conditionnelle, puis au moment du *signal*, réveil et prise du verrou.

Notons qu'en général, il est possible de prendre en compte les priorités des tâches en attente d'une même variable conditionnelle.

Variable conditionnelle	
Verrou du moniteur	File d'attente de tâches

Figure 5.6 – Caractérisation d'une variable conditionnelle.

© Dunod – La photocopie non autorisée est un délit.

Les variables conditionnelles sont des outils de base conjointement utilisés avec des sémaphores d'exclusion mutuelle pour la construction de moniteurs de Hoare, comme le montre ci-dessous l'implémentation de l'exemple présenté au paragraphe 5.2.2, p. 186.

```
verrou : sémaphore_mutex -- sémaphore binaire d'exclusion mutuelle
réseau_libre, rs232_libre : booléen := vrai
condition_réseau, condition_rs232 : variable_conditionnelle
Procédure prendre_réseau
Début
      prendre(verrou)
      tant que réseau_libre = faux faire
      -- on attend que le réseau soit libre
            wait(condition_réseau, verrou)
            -- de façon atomique, libération du verrou et attente
            -- de la condition_réseau
      fait
      réseau_libre := faux
      vendre(verrou)
Fin
Procédure vendre_réseau
Début
      prendre(verrou)
      réseau_libre := vrai
      signal(condition_réseau)
      -- signal de la condition_réseau
      vendre(verrou)
Fin
Procédure prendre_rs232
Début
      prendre(verrou)
      tant que rs232_libre = faux faire
      -- on attend que le port série soit libre
            wait(condition_rs232, verrou)
            -- de façon atomique, libération du verrou et attente
            -- de la condition_rs232
      fait
      rs232_libre := faux
      vendre(verrou)
Fin
Procédure vendre_rs232
Début
      prendre(verrou)
      rs232_libre := vrai
      signal(condition_rs232)
      vendre(verrou)
Fin
Procédure prendre_tout
Début
      prendre(verrou)
      tant que réseau_libre = faux faire
      -- on attend que le réseau soit libre
            wait(condition_réseau, verrou)
      fait
      tant que rs232_libre = faux faire
      -- on attend que le port série soit libre
            wait(condition_rs232, verrou)
```

```
      fait
      réseau_libre := faux
      rs232_libre := faux
      vendre(verrou)
Fin
Procédure vendre_tout
Début
      prendre(verrou)
      réseau_libre := vrai
      rs232_libre := vrai
      signal(condition_réseau)
      signal(condition_rs232)
      vendre(verrou)
Fin
```

L'utilisation de variables conditionnelles permet d'affiner la condition de réveil d'une tâche, ce qui rapproche le moniteur de Hoare implémenté par variable conditionnelle du moniteur à la Ada.

☐ Signaux

Les signaux sont des **événements** (sans données) pouvant être envoyés à une ou plusieurs tâches simultanément. Il y a deux types de signaux : les **signaux synchrones**, internes à une tâche ou un processus, et les **signaux asynchrones**, provenant d'autres tâches ou processus, ou bien de source matérielle externe.
Il est important de noter que la sémantique synchrone/asynchrone concernant les signaux est totalement différente de la sémantique utilisée dans le cas des communications. En effet, une communication synchrone caractérise le fait que l'émetteur et le récepteur d'un message effectuent ensemble une communication, alors que dans le cas asynchrone, l'action d'émettre est totalement déconnectée de l'action de recevoir. Dans le cas des signaux, le terme synchrone caractérise un signal interne à une tâche, alors que le terme asynchrone caractérise un signal externe.

Signaux synchrones

Un signal synchrone est le résultat d'un **événement interne** à une tâche. Cet événement est traité immédiatement (de façon synchrone à son **occurrence**) par la tâche. Lorsqu'un événement interne a lieu (par exemple erreur arithmétique comme une division par zéro par exemple, ou une violation de segmentation mémoire, ou bien le redimensionnement d'une fenêtre graphique gérée par une tâche, etc.), une occurrence du signal synchrone associé a lieu. À chaque signal est associée une action par défaut. Typiquement, l'action par défaut consiste à terminer la tâche, ou bien le processus père, ou encore à ne rien faire. Une tâche peut définir une action spécifique à effectuer lorsqu'elle reçoit un signal synchrone. De même, elle peut définir l'ensemble des signaux qu'elle veut bloquer (il y a certains signaux qui ne peuvent pas être bloqués).

Signaux asynchrones

Les signaux asynchrones sont des signaux provenant d'une **source externe** à l'adresse d'une tâche ou un processus (**signal privé**), ou bien à plusieurs tâches et/ou plusieurs processus (**signal public**).

© Dunod – La photocopie non autorisée est un délit.

Le principe fondamental repose sur la sensibilisation des tâches à différents signaux. Il y a deux façons pour une tâche de se sensibiliser à un signal : une tâche peut déclarer une **ASR** (*Asynchronous Service Routine*) et la lier à un signal ; la tâche devient alors réceptive au signal, et lorsque le signal a lieu, l'ASR est appelée **dans le contexte** (en utilisant la même pile d'exécution) de la tâche, interrompant le traitement en cours. Il est aussi possible pour une tâche de se mettre explicitement en attente d'un signal : la tâche devient donc *bloquée* et l'occurrence de l'événement la fait passer dans l'état *prête*.

L'émission d'un signal privé a pour destination une tâche ou un processus, alors qu'un message public peut être à destination de plusieurs tâches d'un processus.

Lorsqu'un signal privé est envoyé à un processus, la dernière tâche (chronologiquement) à s'être sensibilisée à l'événement peut percevoir l'événement. Lorsqu'un signal public est envoyé à un processus, toutes les tâches sensibilisées au signal peuvent le percevoir. L'utilisation de signaux publics suppose l'existence d'un répartiteur d'occurrences : se sensibiliser à un signal revient à se déclarer auprès du répartiteur, une occurrence concernant un processus est alors envoyée au répartiteur, qui s'occupera de prévenir chaque tâche sensibilisée. Le signal initial sera donc envoyé sous la forme de *n* signaux vers les tâches sensibilisées. Cette implémentation nécessite que le répartiteur ait une priorité plus importante que celle des tâches sensibilisées.

Si une tâche devant recevoir un événement n'est pas prête à le recevoir au moment où l'occurrence a lieu, il y a trois façons dont les occurrences non prises en compte peuvent être gérées :

– **signal fugace** : si l'occurrence ne peut être prise en compte, elle est perdue (figure 5.7) ;

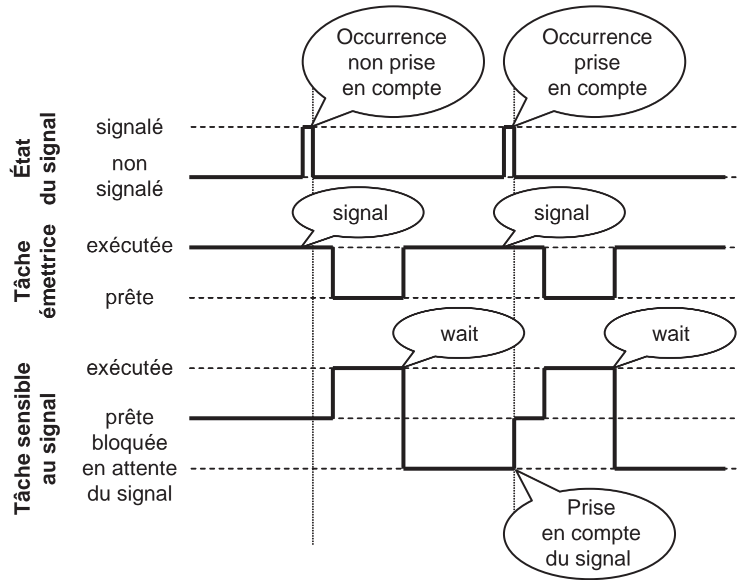


Figure 5.7 – Signal fugace, les occurrences non prises en compte immédiatement sont perdues.

– **signal mémorisé** : si elle ne peut être prise en compte, l'occurrence est mémorisée. Si une autre occurrence du même événement a lieu d'ici à sa prise en compte, elle est ignorée (figure 5.8). Ce type de mémorisation suppose qu'une tâche (typiquement tâche recevant le signal) réinitialise l'occurrence ;

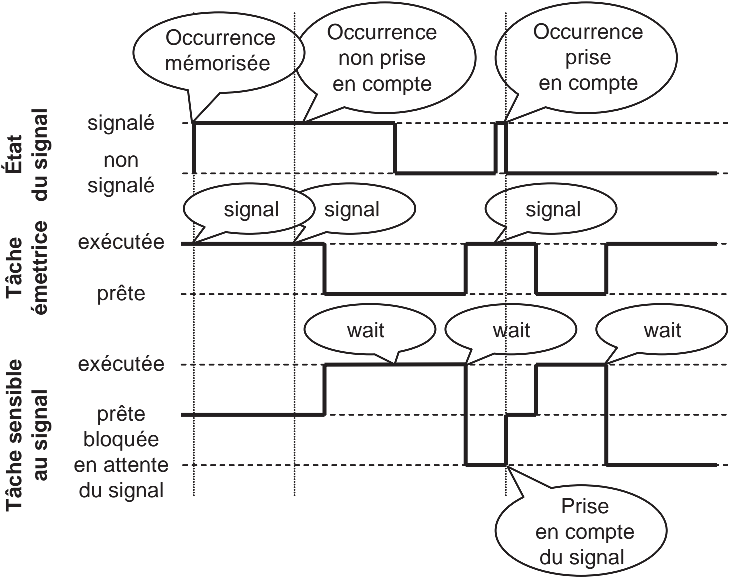


Figure 5.8 – Signal mémorisé, au plus une occurrence est mémorisée.

– **signal à compte :** les occurrences non prises en compte immédiatement sont comptées pour utilisation ultérieure (figure 5.9), dans ce cas, la prise en compte du signal suppose une décrémentation du compte.

■ Communication par message

Il existe différents outils permettant d'assurer des communications par messages :

– boîte aux lettres ;
– tube ;
– *socket* ;
– rendez-vous ;
– tableau noir ;
– …

Chacun de ces outils a ses spécificités propres qui sont présentées dans les paragraphes suivants.

© Dunod – La photocopie non autorisée est un délit.

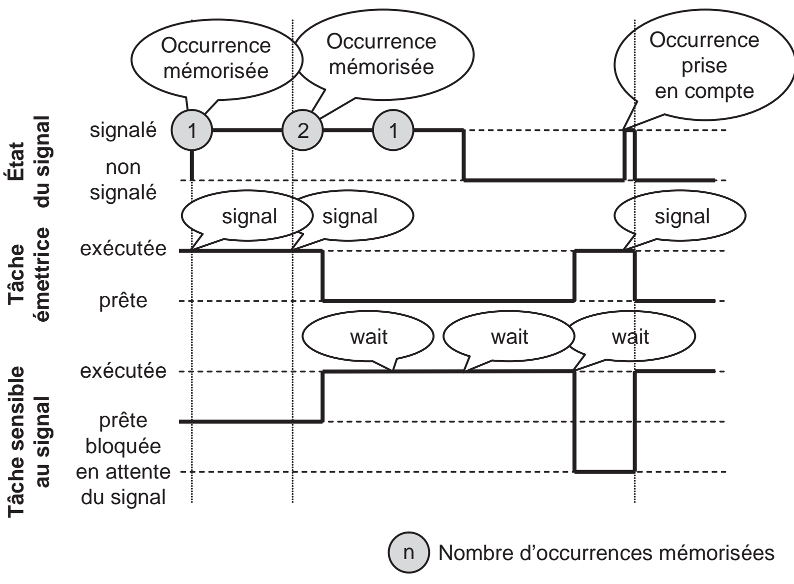


Figure 5.9 – Signal à compte,
le nombre d'occurrences non prises en compte est mémorisé.

☐ La boîte aux lettres

La **boîte aux lettres** permet une communication **asynchrone** puisque l'analogie existant entre communication asynchrone et communication par boîte postale permet d'appréhender très simplement le concept.

Une boîte aux lettres (figure 5.10) est constituée d'une zone d'échange tampon (*buffer*) dans laquelle une tâche dite **émettrice** peut déposer des données. La taille de la zone d'échange est donnée par le nombre de messages maximum multiplié par la taille de chaque message.

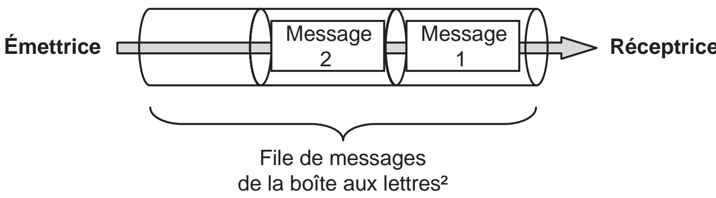


Figure 5.10 – Boîte aux lettres FIFO.

Les données de la boîte aux lettres sont gérées en *FIFO* (*i.e.* premier déposé/premier retiré) ou bien avec une file *FIFO* par niveau de priorité. Une tâche dite **réceptrice**, retire les données dans l'ordre d'arrivée ou de priorité. En fonction du langage support et de l'implémentation, la priorité peut être :

– orientée message (figure 5.11) : le message est muni d'une priorité influençant l'ordre dans lequel il va être lu ;

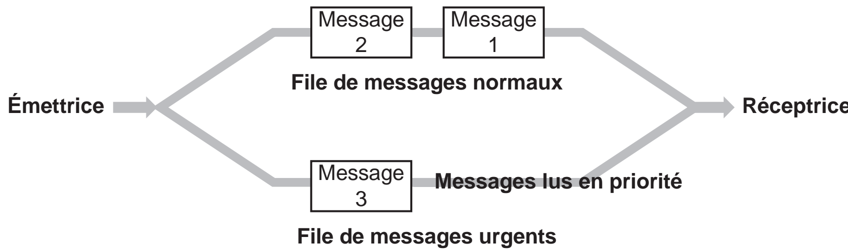


Figure 5.11 – Boîte aux lettres à priorités de messages.

– orientée tâche (figure 5.12) : la priorité du message est liée à la priorité de la tâche émettrice, ce qui n'a de sens que si plusieurs tâches sont à même d'émettre des messages dans la même boîte aux lettres.

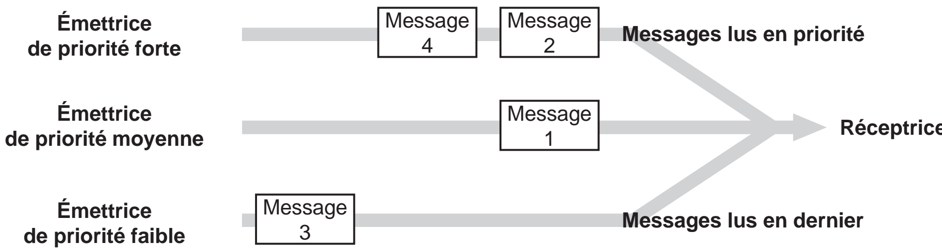


Figure 5.12 – Boîte aux lettres à priorités de tâches émettrices.

On parle de communication asynchrone car, en règle générale, une tâche émettrice n'a pas besoin d'attendre que la tâche réceptrice soit à l'écoute pour lui envoyer des données.

Cependant, les communications par boîtes aux lettres présentent quelques contraintes :

– si la boîte est vide, une tâche désirant recevoir des données est mise en attente dans l'état *bloquée* jusqu'à ce que des données aient été déposées dans la boîte ;

– une boîte aux lettres a une **taille bornée**, et lorsqu'une tâche doit déposer des données dans une boîte pleine, en fonction de l'implémentation choisie, soit le message le plus ancien est écrasé par le nouveau, on parle alors de boîte aux lettres **à écrasement** (ou bien de ***RT FIFO***), soit la tâche passe dans l'état *bloquée* jusqu'à ce que des données aient été retirées de la boîte, on parle alors de boîte aux lettres **sans écrasement**. Dans le cas général, le terme boîte aux lettres désignera une boîte sans écrasement, et on précisera « avec écrasement » dans le cas contraire. Remarquons que le concept de boîte aux lettres (sans écrasement) est identique au problème **producteur/consommateur** présenté au chapitre 4.

© Dunod – La photocopie non autorisée est un délit.

Une boîte aux lettres est généralement utilisée pour une communication entre une émettrice et une réceptrice ou bien *n* émettrices et une réceptrice. Bien que cela soit possible, il est assez rare que l'on utilise une boîte aux lettres pour faire communiquer *n* émettrices et *m* réceptrices.

Les boîtes aux lettres peuvent être à **attente bornée** aussi bien pour l'émission que pour la réception : en cas d'attente dépassant un délai spécifié, une émission ou réception de message est avortée, la primitive renvoyant alors une erreur.

Une boîte aux lettres (figure 5.13) peut donc être représentée par un tampon de messages qui peut être soit une file, soit un ensemble de files classées par priorités, une file d'attente de tâches désirant émettre (cas où la file est pleine et sans écrasement), et une file d'attente de tâches en attente de message (cas où la file est vide). Comme pour les messages, ces files d'attente peuvent être gérées en FIFO ou bien gérées sous forme de files de priorités.

Boîte aux lettres		
File(s) d'attente d'émettrices	Tampon = file(s) de messages	File(s) d'attente de réceptrices

Boîte aux lettres à écrasement	
Tampon = file(s) de messages	File(s) d'attente de réceptrices

Figure 5.13 – Composition d'une boîte aux lettres.

Lorsque le langage utilisé ne propose pas le type de boîte aux lettres désiré, il est possible d'en implémenter simplement à partir de sémaphores, en utilisant la technique du producteur/consommateur présentée au chapitre 4, ou bien de manière plus élégante en utilisant des moniteurs (voir § 5.2.2, p. 186). Nous présentons quelques implémentations de boîtes aux lettres à base de moniteurs dans le chapitre 6.

☐ Le tube

Un **tube** (ou ***pipe***) permet comme la boîte aux lettres une communication unidirectionnelle par passage de messages. Cependant, la philosophie du tube repose sur le concept de flots d'octets (comme tout périphérique Unix), exactement comme dans un fichier : une tâche transmettant des données à travers un tube les envoie comme un flot d'octets dans un fichier (figure 5.14). Les données insérées les unes à la suite des autres dans un tube, bien que pouvant tout à fait être stockées sur disque, sont destinées à être lues en FIFO par une autre tâche. Le tube utilise un concept assez proche du producteur/consommateur. Un producteur place *n* octets dans le tube à chaque fois qu'il veut transmettre un message vers le consommateur, qui prélèvera

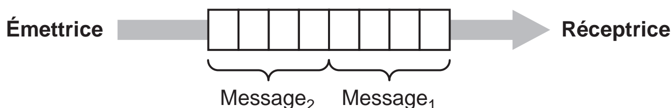


Figure 5.14 – Tube de communication.

les messages *n* octets par *n* octets. Le tube est bloquant en lecture, mais non bloquant en écriture : dans le cas où le tube est plein, l'écriture ne se bloque pas et renvoie une erreur au producteur. En effet, théoriquement, un tube peut être stocké sur disque et donc utiliser un tampon limité seulement par la taille du disque.
Un tube peut donc être vu comme une boîte aux lettres avec perte de messages dans le cas où il est plein. Sa structure (figure 5.15) est limitée à une file d'octets pouvant être stockée sur disque, et une file d'attente de réceptrice(s).

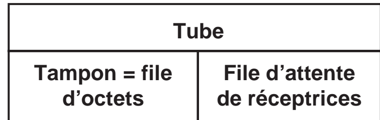


Figure 5.15 – Composition d'un tube.

Cet outil de communication étant à même d'utiliser des fichiers physiques, il n'offre **aucune garantie de délai de communication**.

☐ Le socket

Le *socket* est un outil de communication **bidirectionnelle** (figure 5.16) par message implémenté sur un protocole réseau (typiquement TCP ou UDP). Il permet à deux processus, ou tâches, de communiquer après établissement d'une connexion de type client/serveur (voir § 4.3.2, p. 173).

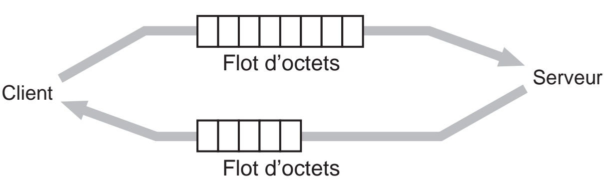


Figure 5.16 – Communication par *socket*.

Les messages sont des flots d'octets, comme pour le tube. Chaque élément communiquant est muni de deux tampons : l'un pour les messages à envoyer (par exemple, le protocole TCP, utilisant la technique du fenêtrage TCP, peut être amené à attendre un accusé de réception avant de continuer l'émission de segments TCP supplémentaires, la bande passante étant limitée, des données peuvent tout simplement être en attente d'accès au médium de communication), et l'autre pour les messages reçus mais non encore lus.
Le *socket* permet une communication entre 2 tâches ou processus (*socket* TCP ou UDP classique), ou bien entre *n* tâches ou processus (cas de la multidiffusion implémentée sur UDP).
Un *socket* se caractérise au niveau de chaque élément communiquant (figure 5.17) par deux tampons bornés : les données arrivées mais non lues, et les données en attente

© Dunod – La photocopie non autorisée est un délit.

d'envoi, et une file d'attente par tampon pouvant contenir la tâche ou processus communiquant sur le *socket*.

Socket			
File d'attente d'émission	Tampon de sortie = file d'octets	Tampon d'entrée = file d'octets	File d'attente de réception

Figure 5.17 – Caractérisation d'un *socket* au niveau d'un processus ou d'une tâche.

☐ Le rendez-vous

Le rendez-vous est un mode de **communication synchrone bidirectionnelle** : on peut le comparer à une communication téléphonique. Ce mode de communication est assez rarement mis en œuvre (il était présent dans Ada 83, mais a été rendu obsolète par l'arrivée des moniteurs dans Ada 95).

Une tâche, dite **acceptante**, attend un rendez-vous (figure 5.18) comme par analogie on pourrait être en attente devant un téléphone. Une tâche dite **appelante** peut demander à effectuer un rendez-vous avec une tâche acceptante, de la même façon qu'on pourrait téléphoner à quelqu'un.

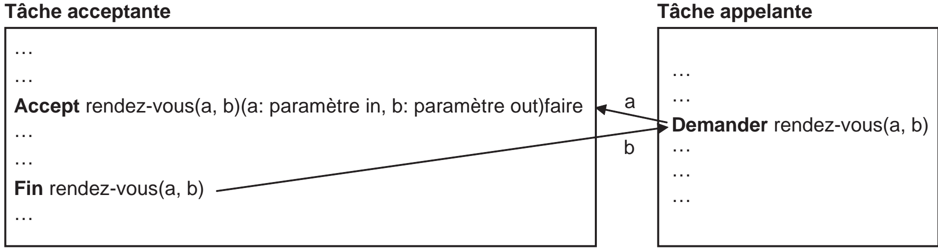


Figure 5.18 – Principe du rendez-vous.

Un rendez-vous n'a lieu que lorsque la tâche acceptante est sur une instruction *accept* et que la tâche appelante demande l'obtention du rendez-vous : la tâche appelante peut passer des données à la tâche acceptante, puis est bloquée jusqu'à la fin du rendez-vous exécuté par la tâche acceptante. Des données peuvent alors être passées de la tâche acceptante à la tâche appelante. Les chronogrammes d'exécution possibles d'un rendez-vous sont donnés sur la figure 5.19.

L'acceptation de rendez-vous, de même que la demande de rendez-vous, sont des instructions bloquantes, un rendez-vous peut donc être caractérisé (figure 5.20) par un pointeur vers la tâche acceptante, son état (en attente de rendez-vous ou non), et une file d'attente gérée en FIFO ou bien un ensemble de files d'attente FIFO gérées par niveau de priorité des tâches appelantes.

Le rendez-vous peut se ramener à l'utilisation de deux boîtes aux lettres (que nous nommerons dans l'exemple qui suit bal_demande_rdz et bal_fin_rdz).

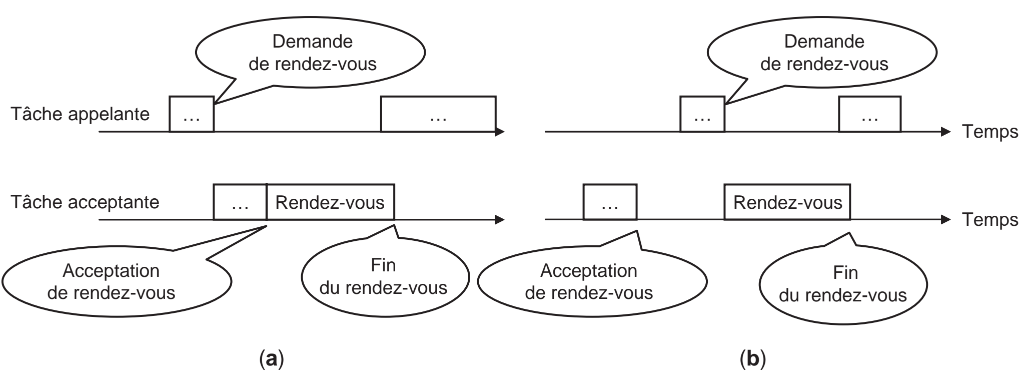


Figure 5.19 – Chronogrammes d'exécution possibles d'un rendez-vous :
(a) la demande a lieu avant l'acceptation, (b) l'acceptation a lieu avant la demande.

Rendez-vous		
Tâche acceptante	État (en attente, pas en attente)	File(s) d'attente d'appelants

Figure 5.20 – Caractérisation d'un rendez-vous.

```
bal_demande_rdz : boîte aux lettres
-- utilisée pour demander un rendez-vous à la tâche
   acceptante
Tâche demandant le rendez-vous :
      créer boîte aux lettres privée bal_fin_rdz
      -- boîtes aux lettres utilisée par la tâche
      -- acceptante pour signifier la fin du rendez-vous
      …
      envoyer un message contenant les paramètres
      d'entrée du rendez-vous et l'identifiant de
      bal_fin_rdz
      attendre un message dans bal_fin_rdz
      -- le message contenant les paramètres de sortie du
      -- rendez-vous sont contenus dans le message reçu
      …
Tâche acceptant le rendez-vous :
      …
      attendre un message dans bal_demande_rdz
      -- attente de rendez-vous, le message reçu contient
      -- les paramètres d'entrée et l'identifiant
      -- bal_fin_rdz de la boîte aux lettres dans
      -- on peut signifier la fin du rendez-vous
      …
      -- actions effectuées lors du rendez-vous
      envoyer dans bal_fin_rdz les paramètres de sortie
      du rendez-vous
```

© Dunod – La photocopie non autorisée est un délit.

On peut noter que dans cette implémentation, la boîte aux lettres de demande de rendez-vous est nécessairement publique, car toute tâche peut demander un rendez-vous, alors que la boîte utilisée pour signifier la fin du rendez-vous est spécifique à chaque tâche demandant le rendez-vous.
Remarque : il est possible, notamment en Ada, d'utiliser un mécanisme d'héritage de priorité lorsqu'une tâche prioritaire est en attente d'une autre tâche moins prioritaire.

☐ Le tableau noir

Le tableau noir (figure 5.21) est conceptuellement le plus simple des moyens de communication **asynchrone** : il utilise une zone de mémoire commune pouvant contenir un message. L'écriture d'un message écrase le message précédent, et la lecture est non bloquante et non destructive (une valeur déjà lue peut être relue tant qu'elle n'a pas été écrasée par une écriture).

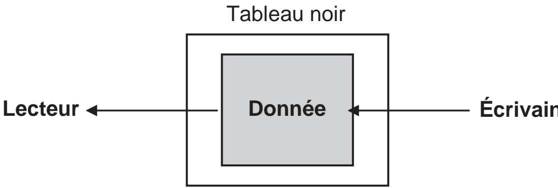


Figure 5.21 – Communication par tableau noir : la lecture est non destructive.

Entre processus, il existe des mécanismes particuliers de déclaration de zone de mémoire commune (le segment mémoire des processus est protégé des autres processus). Entre tâches, le mécanisme s'apparente à l'utilisation d'une variable classique.
Il nécessite cependant un mécanisme de protection garantissant l'**exclusion mutuelle** : en effet, il peut être dommageable qu'une tâche modifiant le message soit interrompue par une tâche lisant le message, car ce dernier pourrait alors être incohérent. Par contre, il n'est pas gênant que plusieurs lectures aient lieu en même temps. Le mécanisme de protection d'un tableau noir s'apparente au problème du **lecteur/écrivain** vu au chapitre 4.
L'un des problèmes qui se pose souvent est le choix d'une valeur initiale pour ce type d'outil de communication.

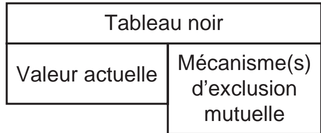


Figure 5.22 – Caractérisation d'un tableau noir.

Le mécanisme d'exclusion mutuelle utilisé est typiquement un sémaphore binaire, ou si le langage support le propose un sémaphore en lecture/écriture. Dans les deux

cas, si possible, il convient d'utiliser un protocole de gestion de ressources (protocole à priorité plafond ou à défaut à priorité héritée) afin d'éviter le phénomène d'inversion de priorités.

☐ Bilan sur la communication par message

La figure 5.23 (page suivante) présente un panorama des « concepts »-outils utilisés pour la communication par message.

■ Synchronisation

La synchronisation consiste à assurer des propriétés de fonctionnement mutuel des tâches. Ainsi, l'exclusion mutuelle, présentée au chapitre 4, nécessite une synchronisation. De la même façon, si une tâche doit absolument avoir eu lieu avant une ou plusieurs autres, sans pour autant avoir à transmettre de message, la contrainte de précédence est une synchronisation. Enfin, si plusieurs tâches doivent s'attendre en un point donné de leur code, une synchronisation de type rendez-vous a lieu (à ne pas confondre avec le rendez-vous présenté au paragraphe 5.2.2, p. 191).

☐ Exclusion mutuelle

Ce type de synchronisation a été vu au chapitre 4 : il s'agit d'empêcher les sections critiques utilisant la même ressource de se préempter mutuellement. Une variante de l'exclusion mutuelle correspond à distinguer l'accès en lecture de l'accès en écriture : dans ce cas, plusieurs lectures peuvent avoir lieu simultanément.
Une synchronisation d'exclusion mutuelle est donc définie (figure 5.24) par une ressource et une file d'attente pouvant être gérée en FIFO ou en files FIFO classées par priorité des tâches.

Exclusion mutuelle	
Ressource	Files(s) d'attente

Figure 5.24 – Caractérisation d'une exclusion mutuelle.

Comme dans le cas du tableau noir, pour lequel la valeur (stockée à un emplacement mémoire) est une ressource critique, le mécanisme d'exclusion mutuelle utilisé est typiquement un sémaphore binaire, ou si le langage support le propose un sémaphore en lecture/écriture. Dans les deux cas, si possible, il convient d'utiliser un protocole de gestion de ressources (protocole à priorité plafond ou à défaut à priorité héritée) afin d'éviter le phénomène d'inversion de priorités.

☐ Synchronisation *n*/1

La synchronisation *n*/1 (*n* producteurs, 1 tâche en attente de déclenchement) traduit une contrainte de précédence. Typiquement, une tâche n'étant activée que sur certaines conditions (déclenchement par une autre tâche, ou bien déclenchement suite à une interruption) est en attente de synchronisation et s'exécute sur déclenchement de celle-ci.

© Dunod – La photocopie non autorisée est un délit.

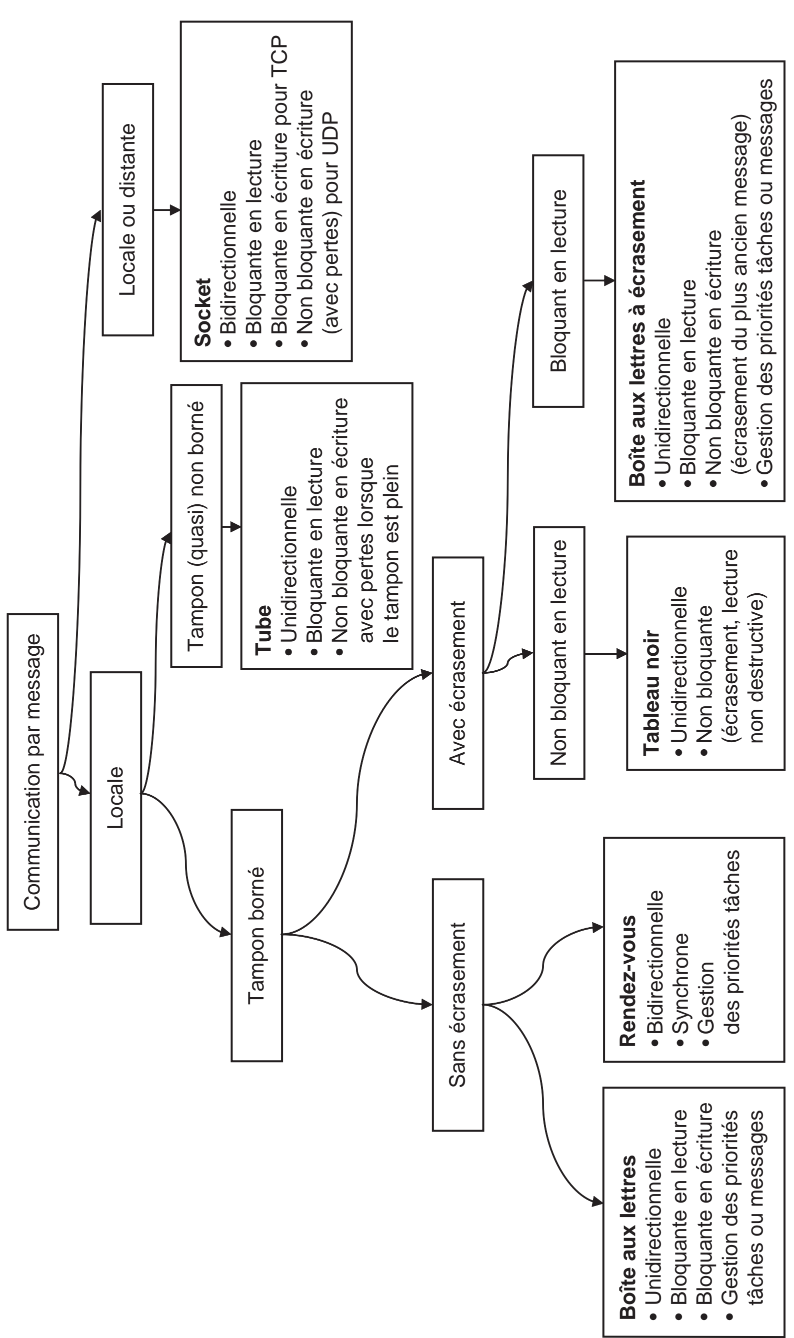


Figure 5.23 – Outils de communication par message.

Une synchronisation de tâche peut être binaire ou à compte : lorsqu'une tâche est en attente sur une synchronisation binaire, si un déclenchement n'a pas été pris en compte et qu'un second déclenchement a lieu, il écrase le dernier déclenchement et la tâche en attente ne sera déclenchée qu'une fois. Dans le cas d'une synchronisation à compte, le nombre de déclenchements non pris en compte est mémorisé, et la tâche en attente de synchronisation sera déclenchée autant de fois que la synchronisation est déclenchée. Ces deux sémantiques sont similaires aux sémantiques associées aux signaux mémorisés et signaux à compte (voir § 5.2.2, p. 191).
Une synchronisation à compte (figure 5.25a) peut être caractérisée par une file d'attente (qui contiendra au plus une tâche en attente) et à un entier contenant le nombre de déclenchement non encore pris en compte.
Une synchronisation binaire (figure 5.25b) peut être caractérisée par un booléen traduisant l'état de la synchronisation (déclenchée ou non), et une file d'attente contenant au plus une tâche.

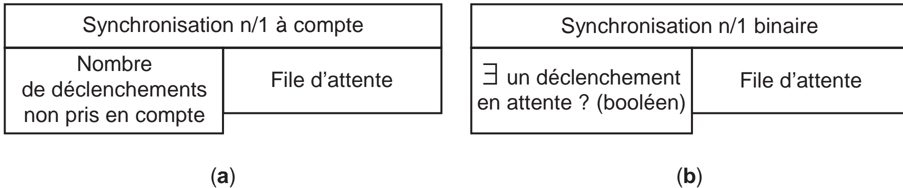


Figure 5.25 – Caractérisation d'une synchronisation *n*/1 (a) à compte, (b) binaire.

Deux implémentations de synchronisation à une tâche sont facilement réalisables : la première repose sur un sémaphore (binaire pour une synchronisation binaire, à compte pour une synchronisation à compte), la seconde repose sur les signaux privés (mémorisés ou à compte). On utilise généralement l'implémentation basée sur un sémaphore (voir ci-après), les signaux étant moins répandus sur les exécutifs que les sémaphores.

```
Implémentation d'une synchronisation à compteur
synchro : sémaphore à compte initialisé à 0
Tâche déclenchante
      …
      vendre(synchro)
      -- déclenchement de la tâche en attente
      …
Tâche en attente
      …
      prendre(synchro)
      -- attente de déclenchement, le sémaphore n'est
      -- disponible qu'à la suite d'un déclenchement
      …
```

© Dunod – La photocopie non autorisée est un délit.

☐ Synchronisation à diffusion

La synchronisation à diffusion est une extension à n producteurs, et m tâches en attente de déclenchement de la synchronisation $n/1$. Si le nombre de tâches en attente est connu, cela revient à utiliser m synchronisations $n/1$. Cependant, dans ce cas, un répartiteur de synchronisation doit être utilisé. L'ordre de prise en compte des déclenchements doit être cohérent avec les priorités des tâches déclenchées (déclenchement de la tâche la plus prioritaire d'abord), ce qui implique que soit le répartiteur connaît les priorités des tâches, soit il est plus prioritaire que toutes les tâches en attente de synchronisation.

Le principe est assez semblable à celui des signaux asynchrones publics vus au paragraphe 5.2.2, p. 191. Comme pour les synchronisations $n/1$, la synchronisation à diffusion peut être binaire (correspond au signal mémorisé, voir figure 5.8) ou à compte (correspond au signal à compte, voir figure 5.9).

On peut implémenter une synchronisation à diffusion à l'aide de signaux ou bien de sémaphores (sur le principe des synchronisations $n/1$) couplés à un répartiteur. Ce type de synchronisation est très rarement utilisé dans les systèmes temps réel et les systèmes de contrôle-commande. En cas de besoin, on préférera le déterminisme de plusieurs synchronisations $n/1$ avec répartiteur (voir exemple d'implémentation ci-après).

```
Synchronisation à compteur à diffusion n/m
synchro_répartiteur : sémaphore à compte initialisé à 0
synchros : tableau de m sémaphores à compte initialisés à 0
Tâche déclenchante
      …
      vendre(synchro)
      -- déclenchement du répartiteur
      …
Tâche répartiteur (priorité>=max(priorités) des tâches en attente
      Faire toujours
            prendre(synchro)
            -- attente de déclenchement
            Pour i allant de 1 à m faire
                  vendre(synchros[i])
                  -- déclenchement de chaque tâche en attente
            Fait
      Fait
Tâche en attente numéro i
      …
      prendre(synchros[i])
      -- attente de déclenchement (via le répartiteur)
      …
```

☐ Rendez-vous synchronisé

Dans les rares cas où l'on souhaite que n tâches soient en même temps à un emplacement spécifique de leur exécution, on y définit l'attente d'un rendez-vous synchronisé à n tâches. Un rendez-vous synchronisé est caractérisé par un nombre de tâches attendues au rendez-vous, et une file d'attente de tâches arrivées au rendez-vous (figure 5.26).

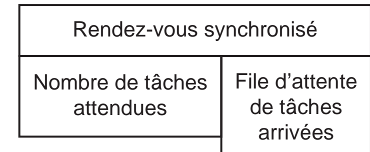


Figure 5.26 – Caractérisation d'un rendez-vous synchronisé.

Une tâche exécutant une attente de rendez-vous est insérée dans la file d'attente et passe dans l'état *bloquée* sauf si elle est la dernière tâche attendue au rendez-vous. Lorsque le nombre de tâches attendues est arrivé au rendez-vous, toutes les tâches de la file d'attente passent dans l'état *prête*.

Lorsque le rendez-vous synchronisé n'est pas disponible, il peut être implémenté à l'aide d'un signal public mémorisé (ou bien de sémaphores et d'un répartiteur) et d'un sémaphore protégeant l'entier comptant le nombre de tâches arrivées au rendez-vous.

```
Rendez-vous synchronisé à 3 tâches
nombre_tâches_attendues : entier := 3
nb_tâches_arrivées : entier := 0
-- nombre de tâches arrivées au rendez-vous
mutex : sémaphore binaire
-- garantit l'exclusion mutuelle des accès à
nombre_tâches_attendues
s : signal public fugace
-- signal déclenché lorsque le rendez-vous peut avoir lieu
Tâche i
      …
      -- attente du rendez-vous
      prendre(mutex)
      nb_tâches_arrivées := nb_tâches_arrivées + 1
      si nb_tâches_arrivées = nombre_tâches_attendues alors
            signaler(s)
            -- réveil de toutes les tâches arrivées
            vendre(mutex)
      sinon
            vendre(mutex)
            attendre(s)
            -- attente du rendez-vous
      fin si
      …
```

On pourrait penser qu'un signal fugace conviendrait, car on peut penser que lorsque les tâches du rendez-vous sont toutes arrivées, elles sont forcément toutes en attente du signal, qui serait donc pris en compte par toutes les tâches du rendez-vous. Le problème est qu'il est possible, la tâche devant vendre le sémaphore avant de se mettre en attente du signal, que l'avant-dernière tâche arrivant au rendez-vous soit préemptée par la dernière tâche arrivant au rendez-vous entre la libération du sémaphore et l'attente du signal. Dans ce cas, le signal ne réveillerait pas l'avant-dernière tâche.

© Dunod – La photocopie non autorisée est un délit.

5.2.3 Gestion des interruptions

Il y a deux types d'interruption : les **interruptions logicielles**, et les **interruptions matérielles**.

■ Interruptions logicielles

Les interruptions logicielles sont déclenchées par une tâche (ou l'exécutif lui-même) lors de son fonctionnement : par exemple débordement lors d'un calcul, erreur de segmentation, défaut de page sur pagination, division par zéro, etc. Dans ce cas, c'est à la tâche de traiter elle-même l'interruption. Enfin, certaines interruptions logicielles auront pour effet de réinitialiser le système (erreur de parité lors d'un transfert entre la mémoire centrale et le processeur, erreur du bus de données, etc.). Une interruption logicielle peut se traduire sous la forme de signal synchrone (voir § 5.2.2, p. 191).

■ Interruptions matérielles

Les interruptions matérielles sont générées par une source externe au processeur, comme par exemple un contrôleur de dispositif d'entrées/sorties (clavier, périphérique de stockage…) ou bien une horloge qui peut être programmée afin de déclencher une interruption au bout d'un certain temps ou à une certaine date. Dans ce cas, il faut qu'un traitement associé au noyau ait lieu, son rôle principal sera de relayer l'interruption vers la tâche de traitement adéquate.

Physiquement, une interruption matérielle est apportée par une ligne d'interruption. Le nombre d'interruptions possibles est limité par le matériel, ainsi, sur une architecture de type x86, il y a 16 interruptions matérielles. Les interruptions matérielles sont souvent appelées **IRQ** (*Interrupt ReQuest*).

Une table nommée **vecteur d'interruption**, contient l'adresse de début du traitement logiciel à exécuter pour chaque interruption. Ce traitement logiciel est exécuté dans le contexte du noyau (en utilisant sa pile, et en interrompant le programme en cours d'exécution) ou bien avec une pile spécifique aux traitements d'interruptions. La table contient aussi un bit par interruption signifiant si une interruption a eu lieu.

Le traitement s'appelle une **routine de traitement d'interruption** (**ISR**) : ce traitement commence éventuellement par prévenir l'élément matériel qui a signalé l'interruption que celle-ci a été prise en compte. Certaines architectures matérielles prévoient en effet qu'un élément matériel doive attendre d'obtenir un acquiescement de leur interruption avant d'en signaler d'autres : cela permet d'être certain qu'aucune interruption n'est perdue. Ce processus doit être très rapide, cela explique en partie qu'il n'est en aucun cas possible d'effectuer une opération bloquante dans une routine de traitement d'interruption.

Une routine de traitement d'interruption ne doit en aucun cas être réentrante (le traitement d'une interruption ne doit pas être interrompu par le traitement de cette même interruption). Lors du traitement d'une interruption, on **masque** donc généralement l'interruption traitée elle-même. Si elle survient à nouveau, le bit informant de l'occurrence de l'interruption est mis à 1 dans le vecteur d'interruptions, mais celle-ci devra attendre avant d'être traitée.

Si le traitement d'interruptions est hiérarchisé, et qu'une interruption plus prioritaire que celle qui est traitée survient, alors la routine de traitement d'interruption est elle-même interrompue par le traitement de l'interruption prioritaire. Dans le cas où le système n'est pas hiérarchisé, ou bien dans le cas où une interruption moins prioritaire survient, son traitement est mis en attente. Dans un système hiérarchisé, tout se passe comme si les interruptions moins prioritaires étaient masquées pendant le traitement d'une interruption.

La façon dont le processeur traite une interruption consiste, après le traitement de chaque instruction (ce qui peut nécessiter plusieurs cycles processeur), à vérifier si une interruption non masquée a eu lieu (bit à 1 dans le vecteur d'interruptions). Si c'est le cas, et que le traitement en cours n'a pas une priorité supérieure à l'interruption elle-même, il interrompt le programme en cours pour appeler la routine de traitement de l'interruption la plus prioritaire si il y a un mécanisme de priorités. L'interruption s'exécutera dans un contexte privilégié (soit dans le contexte noyau, soit dans un contexte spécifiquement défini pour le traitement des interruptions).

Rappelons que la routine de traitement d'une interruption se doit d'être rapide et ne peut pas effectuer d'instruction bloquante ou suspensive, ce qui comprend toutes les entrées/sorties, l'envoi de message dans une boîte aux lettres même à écrasement à cause de la possibilité de se bloquer sur le mécanisme d'exclusion mutuelle de manipulation de la file de messages, etc. Par conséquent, pour un système temps réel, le traitement d'une interruption consiste à déclencher une tâche de traitement de l'interruption (figure 5.27). Une tâche déclenchée par une routine de traitement d'interruption s'appelle une **DSR** (*Deffered Service Routine*) et n'a aucune limitation d'action puisque c'est une tâche ordinaire. Une DSR est déclenchée par synchronisation : le rôle d'une routine de traitement d'interruption se borne alors à déclencher la synchronisation (typiquement en vendant un sémaphore, voir § 5.2.2, p. 201). Généralement, on utilise une synchronisation à compte afin que la tâche puisse prendre en compte plusieurs interruptions survenues lors de son traitement d'une interruption.

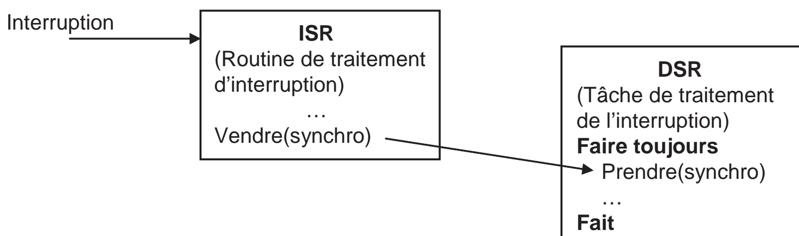


Figure 5.27 – Principe de fonctionnement du traitement d'une interruption matérielle.

5.2.4 Gestion du temps

La gestion du temps sur les systèmes d'exploitation généralistes se base sur une horloge de faible résolution (de l'ordre de la milliseconde) ou bien utilise de façon grossière une horloge de haute résolution.

© Dunod – La photocopie non autorisée est un délit.

En effet, typiquement, l'horloge est programmée de sorte à déclencher une interruption périodiquement. Cette interruption, traitée comme toutes les autres, voit son traitement aboutir à l'exécution de l'ordonnanceur, qui évalue les priorités des tâches et appelle le *dispatcher*. Celui-ci rétablit alors le contexte de la tâche la plus prioritaire. Un exécutif temps réel gère finement le temps, et sa résolution peut approcher la micro seconde. Il n'est pas concevable de traiter une interruption à chaque micro seconde. La stratégie généralement utilisée consiste à utiliser un ou plusieurs circuits programmables spécialisés dans la gestion du temps. Ces circuits peuvent être programmés de sorte à :

– déclencher périodiquement une interruption, typiquement, ce type de programmation est utilisé par l'exécutif pour déclencher périodiquement l'ordonnanceur. Plusieurs exécutifs qualifient la période ainsi utilisée de ***tick*** : le noyau prend la main à chaque *tick* ;
– déclencher une interruption à une certaine date, ce qui permet typiquement à une tâche de se réveiller périodiquement, avec une résolution temporelle importante ;
– déclencher une interruption au bout d'un certain temps, ce qui permet typiquement à une tâche de s'endormir pendant un certain temps, ou bien de surveiller la durée d'un traitement par **chien de garde** : le traitement est lancé parallèlement au décompte d'une horloge, ce qui permettra le cas échéant d'effectuer une action spécifique au cas où le traitement prendrait un temps plus élevé que prévu.

Ainsi, seules les interruptions correspondant à des instants programmés ont lieu, permettant ainsi une extrême finesse temporelle, sans nuire aux performances du système.
Les exécutifs et systèmes d'exploitation temps réel se basent donc sur les horloges pour proposer l'une des deux méthodes suivantes de gestion du temps :

– **Noyau dirigé par le temps** : seule la notion de *tick* permet de gérer le temps, les tâches ou processus ne peuvent être réveillés que sur des *ticks*, c'est le cas par exemple des exécutifs VxWorks® (voir § 5.4.1) et RTEMS (voir § 5.4.3).
– **Noyau dirigé par les événements** : les horloges sont programmables à leur granularité la plus fine, et il est possible de réveiller une tâche ou un processus de façon très fine, c'est le cas par exemple des exécutifs et systèmes d'exploitation de type POSIX et OSEK/VDX.

Les noyaux dirigés par les événements sont strictement plus puissants que les exécutifs dirigés par le temps : il est possible sur les premiers d'obtenir un fonctionnement similaire aux derniers, mais l'inverse n'est pas vrai. Les noyaux dirigés par les événements peuvent atteindre une résolution d'horloge de l'ordre de la microseconde, voire moins, alors que sur un noyau dirigé par le temps il est quasi impossible sans dispositif matériel additionnel d'atteindre une résolution inférieure à quelques centaines de microsecondes (de plus, à ce grain temporel, le surcoût processeur dû au noyau est prohibitif).
Enfin, notons que beaucoup de tâches sont périodiques, et il faut avoir conscience des problèmes inhérents à un code comme ce qui suit :

```
Tâche une_tâche_périodique
Début
      Faire toujours
            -- des actions
      Attendre une durée égale à une période
      Fait
Fin
```

Soit t l'instant où la tâche est activée, il lui faut un temps non nul pour effectuer les actions, d'autant que d'autres tâches peuvent utiliser le processeur et la retarder. L'instruction d'attente d'un délai a donc lieu à une date $t + \varepsilon_1$, la tâche n'est donc réveillée qu'au plus tôt à la date $t + période + \varepsilon_1$. Il lui faut ensuite un temps ε_2 avant d'arriver à l'instruction d'attente, ce qui décale son prochain réveil au plus tôt à la date $t + 2période + \varepsilon_1 + \varepsilon_2$. Et ainsi de suite à chaque période. Ce phénomène s'appelle la **dérive des horloges**. Par conséquent, une tâche périodique se doit de reposer sur un réveil par date (on peut calculer les dates de réveil désirées en fonction d'un instant initial) et non par délai.

5.3 Principales normes temps réel

Ce paragraphe présente deux normes temps réel, POSIX et OSEK. Les outils et concepts définis dans ces normes définissent les services rendus par un exécutif ou un système d'exploitation compatible avec ces normes, appliquées au langage C. Cependant, le langage utilisé par un programmeur tirant parti de ces services peut être un autre langage que le C (typiquement Ada). Il existe d'autres normes temps réel, comme ITRON/BTRON, norme principalement développée au Japon.

5.3.1 La norme POSIX

■ Introduction

POSIX (*Portable Operating System Interface*) est un standard initialement normalisé en 1988 par l'IEEE sous le nom de P1003 et par l'ISO/IEC sous le nom ISO/IEC-9945. Le but de cette norme est de normaliser l'accès aux services offerts à différents niveaux par les systèmes d'exploitation Unix, de sorte à assurer le plus de portabilité possible aux programmes.

La norme a évolué depuis 1988 et est encore en évolution. En réalité, POSIX est un ensemble de normes, regroupées sous le nom 1003.*na* (où *n* est la partie, et *a* l'amendement, comme 1003.1b qui est l'amendement b de la partie 1) sous le standard IEEE et sous forme de parties de la norme ISO/IEC-9945.

Différents groupes de travail participent à l'élaboration de cette norme, et le standard évolue perpétuellement afin de rendre compte des besoins modernes. La plupart des aspects du système d'exploitation Unix sont normalisés, du réseau au système de fichiers en passant par le temps réel, l'interfaçage avec différents langages de programmation, etc. Les standards proposés concernent des interfaces et non pas des

© Dunod – La photocopie non autorisée est un délit.

implémentations, qui sont laissées libres aux développeurs de systèmes d'exploitation, exécutifs ou noyaux conformes à tel ou tel ensemble de standards POSIX.

Afin de donner un aperçu des travaux relatifs à POSIX, le tableau 5.1 présente un aperçu de l'état des standards POSIX en décembre 2003 (la table complète des standards POSIX est donnée en annexe B).

Tableau 5.1 – Standards POSIX.

Nom IEEE	Nom	Notes
1003.1	Interface système (*System Interface*)	Définit l'interface de programmation système d'un système d'exploitation POSIX – Dernière version 2004. Depuis 2001, intègre différents amendements donnés ci-après.
1003.1b	Extensions temps réel (*Realtime Extensions*)	Dernière version 1993, intégrée dans 1003.1 depuis 2001.
1003.1c	Tâches (*Threads*)	Dernière version 1995, intégrée dans 1003.1 depuis 2001.
1003.1d	Extensions temps réel additionnelles (*Additional Realtime Extensions*)	Dernière version 1999.
1003.1h	Tolérance aux fautes (*Fault Tolerance*)	Dernière version 2000 – Devenu 1003.25.
1003.1i	Corrections aux extensions temps réel (Fixes to 1003.1b)	Dernière version 1995, intégrée dans 1003.1 depuis 2001.
1003.1j	Extensions temps réel avancées (*Advanced Realtime extensions*)	Dernière version 2000.
1003.5	Interface Ada avec 1003.1 (*Ada Binding to 1003.1*)	Dernière version 1997.
1003.5a	Mise à jour Ada (*Ada Update*)	Abandonné en 1996.
1003.5b	Ada temps réel (*Ada Realtime*)	Dernière version 1996.
1003.13	Profils temps réel (*Realtime Application Environment Profile*)	Dernière version 1998.
1003.25	Tolérance aux fautes (*Fault Tolerance*)	Nouveau nom de 1003.1h. En développement depuis 1999.
2003.1	Méthodes de test pour mesurer la conformité à 1003.1 (*Test Methods for 1003.1*)	Dernière version 2000.
2003.1b	Méthodes de test pour mesurer la conformité à 1003.1b (*Test Methods for 1003.1b*)	Dernière version 2001.

Beaucoup de systèmes d'exploitation, d'exécutifs temps réel et de noyaux temps réel se disent conformes à POSIX. Il est donc important de savoir à quel ensemble de parties et d'amendements le distributeur se réfère, ainsi que l'année prise en compte. L'année est très importante puisque les amendements 1003.1b et 1003.1c sont directement inclus dans la norme 1003.1 depuis 2001, alors qu'ils ne l'étaient pas avant. Typiquement, un système d'exploitation de type Unix/Linux est compatible à toute ou partie de la norme POSIX.1 (*i.e.* 1003.1). La société/organisme distribuant ce système informe donc généralement les utilisateurs potentiels des tests de conformité POSIX effectués (par exemple un test de type 2003.1, pratiqué par un organisme comme l'*Open Group*).

■ POSIX et le temps réel

Ce qui intéresse généralement le concepteur d'une application de contrôle-commande, éventuellement temps réel, est la conformité aux normes POSIX dites temps réel, c'est-à-dire la conformité aux normes 1003.1b, 1c, 1d (avant 1994 numérotées respectivement 1003.4, 4a et 4b), 1i, 1j (avant 1994 1003.4d) et 1003.13. Depuis 1994, la renumérotation de la norme POSIX a vu la suppression des parties 1003.4, dites temps réel, intégrées pour la plupart comme amendements de 1003.1, les fonctionnalités multitâches, et la gestion du temps devant alors passer fonctionnalités quasi standard des systèmes de type Unix.
Il convient d'être particulièrement attentif aux tests de conformité appliqués aux systèmes dits conformes à POSIX 1003.*xy*, car il arrive parfois que seules les interfaces compatibles à POSIX soient implémentées, mais que les fonctions associées soient vides (non implémentées).
Malgré le but de POSIX, qui est d'uniformiser les interfaces de programmation sur système de type Unix, le concepteur d'une application temps réel peut donc se heurter aux problèmes de fonctions POSIX non implémentées, notamment spécifiques au temps réel. Ainsi, il n'est pas certain que tel ou tel programme implémenté en pur POSIX soit portable tel quel sur tout système « conforme à POSIX ».
Ces problèmes sont heureusement cantonnés à des portions de plus en plus congrues, et on peut espérer qu'au fil des mois ou des années, les systèmes Unix et Linux seront de plus en plus conformes à POSIX, notamment à tous les amendements de la partie 1003.1.
Les paragraphes suivants donnent quelques éléments sur les normes POSIX orientées temps réel. Le rôle de ces paragraphes est de fournir au lecteur les points d'entrée nécessaire à une utilisation des fonctionnalités POSIX.

□ POSIX 1003.1b et 1003.1i : programmation « temps réel »

L'amendement POSIX 1003.1b (1993) corrigée par le 1003.1i en 1995, propose notamment les outils suivants :

- IPC (*Interprocess Communication*) regroupant communication par boîte aux lettres, sémaphores, et tableau noir permettant à des processus de communiquer et se synchroniser ;
- signaux temps réel ;
- fichiers temps réel ;

© Dunod – La photocopie non autorisée est un délit.

– ordonnancement ;
– horloges haute précision ;
– mémoire non virtuelle.

Communication et synchronisation de processus

Trois outils sont définis pour permettre aux processus de communiquer et se synchroniser : les boîtes aux lettres nommées *message queues*, les sémaphores, et les tableaux noirs (nommés *shared memory*) : ce sont les IPC.
Un IPC pouvant être utilisé par plusieurs processus (rappelons que la mémoire des processus est privée), il est créé par le système d'exploitation et caractérisé par un nom sous forme de chaîne de caractères. Lorsqu'un processus demande à créer un IPC partagé, il lui donne un nom : si le nom n'est pas déjà utilisé par un IPC du même type, le système d'exploitation crée alors l'objet, et le processus récupère un identifiant d'IPC lui servant à utiliser cet objet. Si le nom correspond à un IPC du même type (boîte aux lettres, tableau noir, sémaphore) déjà existant, alors le système lui renvoie l'identifiant de l'IPC déjà existant.

Boîtes aux lettres nommées

La manipulation d'une boîte aux lettres nommée se fait à l'aide de fonctions dont le nom est préfixé par « mq_ » pour ***message queue***. Ainsi, la fonction *mq_open* crée une nouvelle boîte ou bien ouvre une boîte aux lettres existantes.
La différence entre créer et ouvrir réside dans l'existence ou non du nom de la boîte aux lettres. En effet, lorsque deux processus doivent communiquer à travers une boîte aux lettres nommée par exemple « ma_bal », on ne sait généralement pas lequel des deux processus exécutera le premier l'instruction *mq_open*, c'est donc le premier *mq_open* qui créera la boîte, celle-ci étant inexistante, le second ouvrira la boîte « ma_bal » sans la créer, celle-ci existant déjà.
Une *message queue* est une boîte aux lettres bornée sans écrasement, avec possibilité d'affecter une priorité aux messages.
Lors de la création, on définit les attributs de la boîte aux lettres comme le nombre maximal de messages et la taille de chaque message.
Les envois et attentes de messages sont effectués avec les fonctions *mq_send* et *mq_receive*. Notons que l'amendement 1003.1j ajoute la possibilité de borner l'attente lors des envois ou réceptions de messages (dans ce cas, si le délai ou bien la date maximale est dépassé, la fonction retourne une erreur).
Notons qu'il existe une interface alternative, nommée XSI (*X/Open System Interface*), manipulant les mêmes boîtes aux lettres, dans laquelle les fonctions sont nommées *msgrcv*, *msgsnd*…

Sémaphores

Les **sémaphores** sont basés sur le même principe que les *message queues* : ouverture ou création d'un sémaphore avec *sem_open*. Un **sémaphore nommé** (nom généralement donné aux sémaphores POSIX.1b) est un sémaphore à compte. On peut prendre un sémaphore avec la fonction *sem_wait* et le libérer avec la fonction *sem_post*. Notons que comme dans le cas des boîtes aux lettres, l'amendement 1003.1j ajoute la possibilité de faire des attentes bornées à une durée ou une date.

De même, une interface XSI alternative existe. L'interface XSI augmente les possibilités des sémaphores POSIX en permettant de manipuler des tableaux de sémaphores. Cet outil est intéressant puisqu'il évite les risques d'interblocage : lorsqu'un processus doit prendre plusieurs sémaphores, s'il utilise les primitives de prise de tableau de sémaphores, le cycle menant à l'interblocage est impossible, car les sémaphores sont pris dans un ordre déterminé (voir chapitre 4).

Notons qu'aucun mécanisme de gestion de ressource (protocole à priorité héritée ou plafond) n'est disponible pour les sémaphores nommés.

Tableaux noirs

La mémoire des processus étant protégée, le mécanisme de **tableau noir** n'est possible qu'en utilisant des zones mémoire spécifiquement partagées. Celles-ci portent le nom de ***shared memory***, et sont créées ou ouvertes par la fonction *shm_open*. Après avoir ouvert/créé une zone de mémoire partagée, on peut la calquer sur une variable du processus avec la fonction *mmap* : tout accès à la variable est alors un accès à la zone de mémoire partagée, et non pas l'accès à une variable interne au processus.

Ce mécanisme, de même que le mécanisme de tableau noir se couple généralement à un sémaphore afin de garantir l'exclusion mutuelle des accès.

Comme pour les autres IPC, les tableaux noirs peuvent être accédés à partir d'une interface XSI.

Signaux temps réel

Les **signaux temps réel** sont des signaux asynchrones avec données caractérisés par un numéro, situé entre deux constantes dépendantes de l'architecture : SIGRTMIN et SIGRTMAX. Un tel signal peut arriver à un processus suite à l'appel de la fonction *kill* (envoi de signal, comme son nom ne l'indique pas), ou *raise* (envoi de signal à soi-même) ou bien la fin d'une horloge (délai programmé, ou date programmée), ou bien encore événement envoyé à la suite d'une interruption matérielle…

La spécificité des signaux temps réel par rapport aux signaux non temps réel est qu'un signal émis est forcément reçu (grâce à un mécanisme de files de signaux), qu'ils supportent la notion de priorité, et qu'ils fonctionnent sur un mode hiérarchisé : le traitement d'un signal asynchrone peut être interrompu par le traitement d'un signal plus prioritaire (sauf si le signal est explicitement masqué). De plus, il est possible d'associer une valeur à un signal temps réel.

Un processus peut masquer ou démasquer chaque signal temps réel. Typiquement, un processus utilisant les signaux temps réel commence par masquer l'ensemble des signaux (fonction *sigemptyset*), puis à définir une action pour chacun des signaux qu'il souhaite traiter (fonction *sigaction*). Le traitement a lieu soit par l'appel d'une fonction exécutée dans le contexte initial du processus (préemption du traitement en cours), soit par la création d'une tâche qui exécutera une fonction dans son contexte propre.

Il est possible d'attendre explicitement un ou plusieurs signaux (fonction *sigwait*).

Notons que certaines fonctions bas niveau masquent les signaux pendant leur exécution (comme la plupart des fonctions manipulant les IPC, les fonctions manipulant les signaux elles-mêmes, etc.). Si plusieurs signaux arrivent et ne peuvent être traités,

© Dunod – La photocopie non autorisée est un délit.

ils sont, si possible, placés dans des files d'attente. Lorsque le processus est prêt à traiter un signal, il commence par traiter les signaux de plus petit numéro.

Fichiers temps réel

Les fichiers temps réel asynchrones utilisent des tampons de données à écrire (ou lire) en mémoire vive, et se chargent de piloter la mémoire de masse de sorte à ne pas ralentir les processus souhaitant utiliser des fichiers. Une opération de lecture ou d'écriture est donc initiée par un processus, mais celui-ci n'a pas à attendre que l'opération soit effectuée pour continuer son exécution. Le processus peut être prévenu par signal à la fin de l'opération. Les opérations en attente sont traitées en respectant la priorité attribuée à l'opération.

Ordonnancement

POSIX offre deux niveaux de programmation concurrente : processus et tâches (*threads*). L'ordonnancement peut donc se faire à deux niveaux : local (figure 5.28) et global (figure 5.29), avec une possibilité de mixer les deux niveaux (figure 5.30).

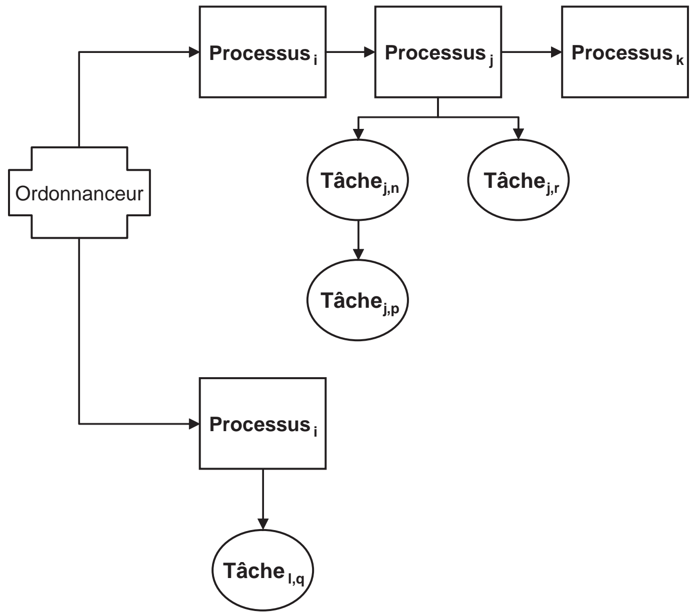


Figure 5.28 – Ordonnancement POSIX local.

Dans le cas d'un **ordonnancement local**, le modèle d'ordonnancement utilisé est hiérarchique : les processus sont en concurrence pour le processeur (tout se passe comme s'il y avait une file d'attente par niveau de priorité). À l'intérieur d'un processus multitâche, les tâches sont aussi gérées par priorité, et le temps alloué au processus est réparti aux tâches le composant. Ainsi, sur la figure 5.28, toutes les tâches sont ordonnancées à l'intérieur de leur processus.

Le problème du modèle local est que lorsqu'une tâche fait une action bloquante, par exemple une entrée/sortie, le processus entier est bloqué. De plus, les différentes tâches d'un processus ne peuvent être ordonnancées simultanément si plusieurs processeurs sont à disposition.

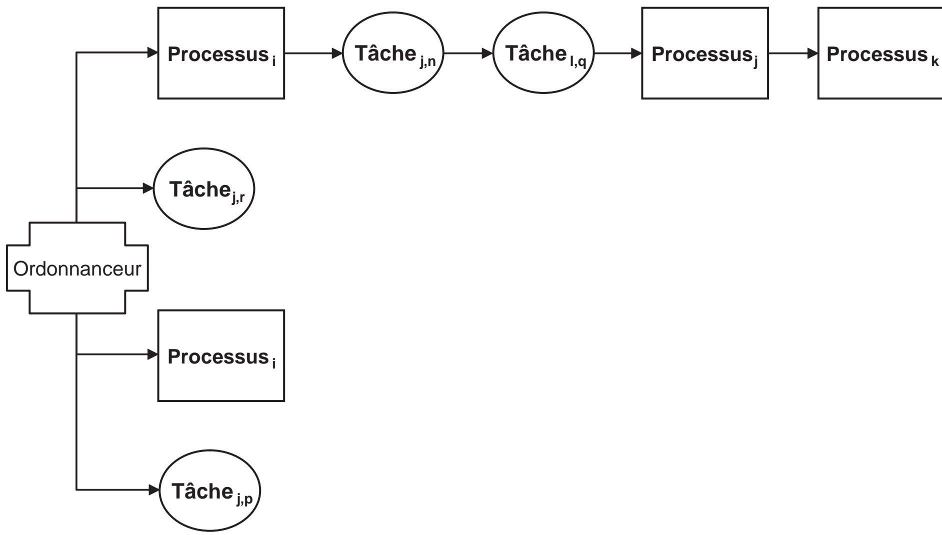


Figure 5.29 – Ordonnancement POSIX global.

Dans un **ordonnancement global** (figure 5.29), chaque élément, qu'il soit tâche ou processus, est ordonnancé au même niveau. L'inconvénient est qu'il est nécessaire d'effectuer un appel système, plus lourd qu'un appel de fonction interne au processus, afin de modifier un paramètre influençant l'ordonnancement. Cependant, les tâches d'un même processus ne se bloquent pas lorsque l'une des tâches du processus fait une entrée/sortie bloquante, et elles peuvent être exécutées simultanément sur plusieurs processeurs.

Dans un **ordonnancement mixte**, on peut choisir pour chaque tâche si elle doit être ordonnancée localement (à l'intérieur de son processus père) ou globalement. Ainsi, sur la figure 5.30, la tâche $Tâche_{l,q}$ est ordonnancée au niveau local, alors que les autres tâches sont ordonnancées au niveau global. Avec un choix judicieux des tâches locales et globales, les avantages des deux approches peuvent être cumulés.

POSIX impose au minimum 32 niveaux de priorité, et propose quatre politiques d'ordonnancement :

– SCHED_FIFO : la tâche prête ou le processus prêt ayant la plus forte priorité au niveau global se voit attribuer le processeur. En cas d'égalité, SCHED_FIFO exécute les tâches ou processus dans leur ordre d'arrivée dans la file des tâches prêtes.

© Dunod – La photocopie non autorisée est un délit.

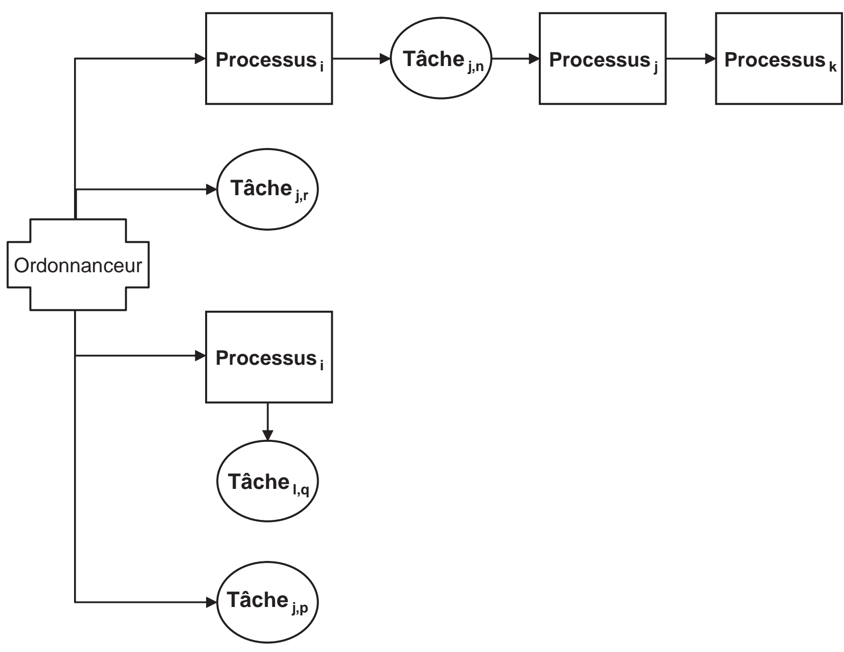


Figure 5.30 – Ordonnancement POSIX mixte.

– SCHED_RR : fonctionne comme SCHED_FIFO en se basant sur les priorités. En cas d'égalité, SCHED_RR (RR signifie *Round Robin* ou tourniquet) se base sur un quantum de temps, et applique l'algorithme du tourniquet (voir chapitre 4).
– SCHED_OTHER : défini par l'implémentation.
– SCHED_SPORADIC : définie en 1998 dans POSIX.13, cette politique d'ordonnancement se base sur la technique du serveur sporadique (voir chapitre 8). Le principe consiste à allouer périodiquement une certaine quantité de temps processeur à des tâches. La quantité de temps s'appelle la capacité du serveur. Au fur et à mesure qu'une tâche utilisant un serveur sporadique s'exécute, la capacité du serveur diminue. Si la capacité tombe à 0, la priorité de la tâche tombe à une priorité faible. La politique de base est la politique SCHED_FIFO.

Notons qu'en cas d'ordonnancement local ou hiérarchique, la politique d'ordonnancement choisie au niveau global, peut être différente de la politique d'ordonnancement choisie au niveau local.

Horloges

Deux améliorations sont apportées aux horloges temps réel par rapport aux horloges classiques : la résolution d'horloge et la gestion du temps absolu (afin d'éviter la dérive des horloges, notamment pour les tâches périodiques).
Toutes les opérations proposées peuvent utiliser une horloge, appelée CLOCK_REALTIME. Normalement, cette horloge représente le temps écoulé depuis le 1er janvier 1970 à 00 h 00. Cette horloge a une résolution au pire de 20 ms, mais,

en fonction de l'architecture matérielle sous-jacente, on peut attendre une résolution de l'ordre de la microseconde. Afin de prévoir une granularité temporelle très fine au niveau des horloges, la structure représentant une date ou une durée a une résolution d'une nanoseconde. Cependant, lorsque l'on utilise une durée ou bien une date, les fonctions de gestion de temps arrondissent à la granularité de l'horloge.

Comme il est possible de modifier l'horloge CLOCK_REALTIME, une seconde horloge a été introduite dans la norme (attention cette horloge est optionnelle dans les implémentations). Cette horloge, nommée CLOCK_MONOTONIC, a comme son nom l'indique, une propriété de monotonie : elle est modifiée par l'horloge physique, mais ne peut pas, normalement, être modifiée par programme.

Deux autres horloges peuvent optionnellement exister : CLOCK_CPU_TIME, donnant le temps processeur consacré à un processus, et CLOCK_THREAD_CPU-TIME_ID donnant le temps processeur consacré à une tâche. Ces deux horloges, mises à 0 au moment du lancement d'un processus ou d'une tâche, sont généralement utilisées pour calculer la durée d'exécution des tâches ou processus.

Les horloges CLOCK_REALTIME et CLOCK_MONOTONIC fonctionnent sur deux modes :

– le mode ***one shot*** consiste à associer à une horloge soit une durée, soit une date absolue (la durée ou date est arrondie à la résolution de l'horloge) afin de créer un ***timer***. Lorsque le *timer* expire, c'est-à-dire lorsque l'horloge atteint la durée désirée ou bien atteint la date absolue désirée, une action, typiquement l'envoi d'un signal, a lieu. Le mode *one shot* est utilisé pour attendre exceptionnellement un certain temps ou une certaine durée, ou programmer un chien de garde (*watchdog timer*) associé à une opération potentiellement longue ou bloquante, etc. ;
– le mode périodique : le mode périodique consiste à créer un timer *périodique* : on associe à une horloge une date de première expiration (ou bien une durée jusqu'à la première expiration) du *timer*, et une période d'expiration. Tout se passe comme si on avait un *timer one shot* qui expire de manière strictement périodique à partir de sa première expiration. Ce mode est extrêmement intéressant dans le cas de tâches périodiques.

Lorsque les deux horloges (temps réel et monotone) sont disponibles, il est conseillé d'utiliser l'horloge temps réel pour donner une date absolue à un *timer*, et l'horloge monotone pour donnée un délai.

Les principales fonctions associées aux horloges sont données dans le tableau 5.2.

POSIX est un système dirigé par les événements (voir § 5.2.4).

Gestion de la mémoire

Afin d'éviter les indéterminismes liés à l'utilisation de la mémoire virtuelle, la norme POSIX.1b définit le concept de mémoire bloquée. Il est possible, à l'aide des fonctions *mlock* et *mlockall* de forcer toute ou partie de la mémoire d'un processus à résider en mémoire centrale. La mémoire bloquée ne peut pas faire l'objet de *swap* (voir § 4.2.3, p. 157).

De même, un fichier peut être placé en mémoire centrale avec la fonction *mmap*, ce qui permet des temps d'accès presque déterministes.

© Dunod – La photocopie non autorisée est un délit.

Tableau 5.2 – Principales fonctions POSIX de gestion d'horloges.

clock_getres	Renvoie la résolution d'une horloge, soit l'écart séparant deux *ticks*. La résolution est définie en fonction de l'architecture sous-jacente et ne peut normalement pas être modifiée par programme. Elle donne la précision maximale des dates ou durées utilisées par les *timers*.
timer_create	Crée un *timer* dans un processus, tel qu'il envoie un signal asynchrone à expiration(s).
timer_settime	Déclenche un *timer* soit en mode *one shot*, soit en mode périodique, avec des délais ou des dates absolues. Le signal choisi lors de la création est envoyé au processus à expiration(s) du *timer*.
nanosleep	Endort le processus (ou la tâche) appelant pendant une certaine durée ou jusqu'à une date absolue (utilise l'horloge CLOCK_REALTIME).
clock_nanosleep	Identique à *nanosleep* sauf que l'horloge utilisée peut être différente de CLOCK_REALTIME.

□ **POSIX 1003.1c et 1003.1j : tâches POSIX**

L'amendement POSIX.1c définit le concept de tâches ou ***pthread***. Plus légères à manipuler que les processus eux-mêmes, des tâches POSIX appartenant à un processus partagent le même espace mémoire global (elles ont le même tas), et se distinguent notamment par une pile différente, ainsi que des attributs particuliers (priorité, type d'ordonnancement, signaux…).

L'ordonnancement des tâches peut s'effectuer au niveau interne au processus (local), au niveau noyau (global), ou bien de façon mixte (voir § 5.3.1, p. 214).

Tâches POSIX

Une tâche est créée par la fonction *pthread_create*. Ses arguments principaux sont des attributs de tâches, un pointeur de fonction (cette fonction est le code de la tâche) et un pointeur pouvant contenir le ou les arguments à passer à la fonction.

Les attributs d'une tâche ne sont pas portables et ne sont pas définis par la norme POSIX, ils peuvent définir notamment :

– le type d'ordonnancement à appliquer à la tâche (local ou global) ;
– la taille de la pile sur les architectures matérielles qui le nécessitent ;
– une priorité ou bien la définition d'un serveur sporadique associé à la tâche (voir § 5.3.1, p. 214) ;
– l'état détaché ou attaché de la tâche. En effet, on peut exécuter une tâche de façon synchrone (*join*), dans ce cas, le créateur de la tâche attend la terminaison de celle-ci avant de poursuivre sont exécution. Ou bien les tâches peuvent être détachées (*detach state*), ce qui les rend indépendantes de leur créateur. Typiquement, si les tâches sont créées par la procédure principale d'un processus, et que l'on veut éviter

la terminaison prématurée du processus avant la terminaison des tâches, on pourra attacher les tâches créées à la procédure principale du processus.

Après sa création, il est possible de modifier les attributs d'une tâche, et par exemple de l'attacher à son créateur. Pour cela, il est nécessaire de connaître l'identificateur d'une tâche, renvoyé lors de la création. Une tâche peut connaître son propre identifiant grâce à la fonction *pthread_self*.

Synchronisation et communication de tâches

POSIX définit pour les tâches des outils de communication et de synchronisation plus légers, plus rapides, et plus puissants que les IPC. Les plus intéressants sont les *mutex*, les sémaphores en lecture/écriture, et les variables conditionnelles.

Tous les outils de synchronisation sont définis à l'aide d'attributs, comme le sont les tâches. L'utilisation d'attributs, modifiables par clés, rend POSIX utilisable sur des architectures variées, au détriment de la portabilité du code : certains attributs peuvent exister sur une plateforme, mais pas sur une autre.

Un attribut spécifique, nommé *pshared*, permet de rendre un outil de synchronisation visible à l'intérieur d'un processus uniquement, ou bien à tous les processus.

Enfin, notons la possibilité d'effectuer des attentes bornées sur les instructions bloquantes des objets de synchronisation et de communication (soit en durée, soit jusqu'à une certaine date).

Sémaphores binaires

Les sémaphores binaires (ou *mutex*) sont dédiés à l'exclusion mutuelle. Les attributs d'un *mutex* permettent de choisir le protocole de gestion de ressource associé au *mutex* (priorité plafond immédiat ou bien priorité héritée).

On peut noter l'existence de fonctions permettant d'effectuer une attente bornée (soit en durée, soit jusqu'à une date déterminée) sur un sémaphore.

Les tâches en attente d'un sémaphore sont classées en files FIFO à priorités.

Sémaphores en lecture/écriture

POSIX.1c propose les sémaphores en lecture/écriture (*rwlocks*) : ce sont des sémaphores binaires que l'on peut utiliser en lecture ou en écriture. La fonction *pthread_rwlock_rdlock* permet de requérir le sémaphore en lecture, alors que *pthread_rwlock_wrlock* requiert le sémaphore en écriture.

Aucun protocole de gestion de ressources n'est défini sur les sémaphores en lecture/écriture.

Variables conditionnelles

Comme nous l'avons vu au paragraphe 5.2.2, p. 189, les variables conditionnelles représentent un outil puissant, permettant de créer des moniteurs. POSIX.1c propose cet outil, nommé *pthread_cond*. Après sa création (*pthread_cond_init*), une variable conditionnelle peut servir, conjointement avec un *mutex*, à synchroniser des tâches.

Il faut cependant être attentif au fait que si plusieurs tâches sont susceptibles d'être bloquées simultanément sur la même variable conditionnelle, un appel à l'équivalent de *signal* (*pthread_cond_signal*) ne réveille qu'une seule tâche bloquée sur la variable

© Dunod – La photocopie non autorisée est un délit.

conditionnelle. Dans le cas où plusieurs sont susceptibles d'être bloquées sur une variable conditionnelle, on peut être amené à utiliser la fonction (*pthread_cond_broadcast*) qui réveille toutes les tâches bloquées sur une variable conditionnelle.
Il est à noter qu'il est possible d'effectuer une attente bornée sur une variable conditionnelle. L'attente bornée sur une variable conditionnelle n'étant jamais signalée est une technique utilisée pour rendre une tâche périodique, comme nous le verrons au chapitre 6.

Rendez-vous

L'amendement 1003.1j introduit les rendez-vous synchronisés (voir § 5.2.2, p. 204) sous le nom de *pthread_barrier*. Après création d'un rendez-vous synchronisé (*pthread_barrier_init*) pour un certain nombre de tâches, des tâches peuvent s'attendre mutuellement sur une instruction *pthread_barrier_wait*. Lorsque le nombre de tâches requises est arrivé au rendez-vous, celles-ci sont réveillées et peuvent poursuivre leur exécution.
Notons qu'une attente de rendez-vous ne peut pas être bornée dans le temps.

SpinLocks

L'amendement 1003.1j introduit les ***spinlocks***, aussi appelés *Test and Set Lock*. Cet outil de synchronisation, antérieur au sémaphore, fonctionne de la façon suivante : un verrou est défini par un entier valant 0 lorsqu'il est ouvert, et 1 lorsqu'il est fermé. Verrouiller un *spinlock* (instruction *pthread_spin_init*) consiste, de façon atomique, à lire la valeur actuelle du verrou et à mettre celui-ci à 1. Après avoir verrouillé un *spinlock*, on peut donc être sûr que le verrou est mis. On peut donc alors observer la valeur lue avant la fermeture du verrou : si elle valait 0, le verrou était ouvert, et la tâche est responsable de sa fermeture. Dans le cas contraire, si le verrou était déjà fermé (valeur 1), alors une autre tâche avait déjà mis le verrou.
L'utilisation de *spinlocks* est très risquée, car l'utilisation de la fonction *pthread_spin_lock* effectue une attente active sur le verrou (boucle testant le verrou jusqu'à ce qu'il soit disponible). Ce mécanisme est cependant extrêmement rapide et peut être utilisé notamment dans le cas multiprocesseur.

□ POSIX 1003.13 : profils temps réel

Nous avons vu lors de la description sommaire des extensions temps réel POSIX, ainsi que dans le tableau de l'annexe B, que la norme POSIX était très vaste, et englobait une partie non négligeable de ce qui définit un système d'exploitation complet.
Un exécutif temps réel se doit d'être compact et d'avoir une petite empreinte mémoire (occupation mémoire) afin d'être embarqué sur une architecture de type petit microprocesseur ou microcontrôleur. De plus, ce type d'architecture matérielle ne fournit que rarement tous les services fournis par un système d'exploitation (gestion matérielle du changement de contexte, mémoire virtuelle, systèmes de fichiers…).
POSIX.13 définit donc des profils (voir tableau 5.3), restreignant les éléments utilisés. Ces profils sont préfixés par PSE (profil d'environnement générique), puis 5 (numéro des profils temps réel), et sont au nombre de quatre : PSE 51, PSE 52, PSE 53 et PSE 54.

Tableau 5.3 – Profils temps réel définis dans POSIX 1003.13.

PSE 51	Profil de système temps réel minimaliste (*Minimal Realtime System Profile*)	Définit un système multitâche à un seul processus, sans système de fichiers (les seuls fichiers utilisés sont les fichiers spéciaux, moyens d'accéder aux interfaces d'entrées/sorties). Nécessite un processeur (ou microcontrôleur) et de la mémoire. Ne nécessite pas de support matériel pour la pagination (MMU), ni de périphériques standard (console, clavier, périphériques de stockage…).
PSE 52	Profil de système de contrôle temps réel (*Realtime Controller System Profile*)	Extension du profil PSE 51 avec prise en compte d'un système de fichiers supportant un accès asynchrone. Ne nécessite pas de MMU.
PSE 53	Profil de système temps réel dédié (*Dedicated Realtime System Profile*)	Extension multiprocessus de PSE 51, avec un système de fichier simpliste (pas de hiérarchie comme les répertoires).
PSE 54	Profil de système temps réel généraliste (*Multi-Purpose Realtime System Profile*)	Multiprocessus, avec système de fichiers. Autorise les interactions avec un utilisateur. Implémente POSIX.1, 1b, 1c et ou 5b, 2 et 2a. PSE 54 définit un système d'exploitation minimal, avec périphériques d'entrées/sorties, périphériques de stockage, pagination de la mémoire, support réseau… Supporte la présence de tâches temps réel et non temps réel.

Le profil PSE 51 est utilisé pour un système embarqué minimaliste, typiquement un système embarqué de contrôle-commande. Un tel profil a une empreinte mémoire de quelques kilo-octets.

Le profil PSE 52, proposant un système de fichiers en plus du profil PSE 51, n'est pas (ou très peu) implémenté. Il serait utile dans le cas d'un système de contrôle-commande enregistrant des données sur un disque embarqué.

Le profil PSE 53 est typiquement utilisé à la place du profil PSE 51 sur un système embarqué multiprocesseur sur lequel on ne souhaite pas répliquer de noyau sur chaque processeur.

Le profil PSE 54 englobe les autres profils, mais a une empreinte mémoire beaucoup plus importante. Il est typiquement utilisé sur une architecture de type PC ou autre micro-ordinateur.

5.3.2 La norme OSEK/VDX

■ Introduction

La norme OSEK/VDX est née en 1995 de la fusion d'un consortium de constructeurs d'automobiles allemands (OSEK est l'acronyme de *Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug*) et d'un consortium de constructeurs d'automobiles français (VDX est l'acronyme de *Vehicle Distributed eXecutive*).

© Dunod – La photocopie non autorisée est un délit.

Le but de la norme OSEK/VDX est de définir un exécutif adapté au contrôle embarqué dans les systèmes automobiles, composés de plusieurs unités de contrôle distribuées sur un ou plusieurs réseaux de terrain. La normalisation a pour effet de diminuer les coûts d'intégration de composants entre les constructeurs automobiles et les instrumentiers.

Bien qu'initialement conçue pour le domaine automobile, cette norme est très bien adaptée à d'autres domaines, comme les applications de contrôle-commande. Moins utilisée que POSIX dans le cas général, elle a l'avantage d'être plus facile d'accès, de par sa spécialisation.

Ce paragraphe présente sommairement l'état de cette norme dans sa seconde version, élaborée en 1997, ainsi que la norme portant sur les communications qui en est à sa troisième version.

La norme OSEK/VDX définit un ensemble d'objets pour lesquels un exécutif OSEK/VDX doit ou peut définir un ensemble de services. Les objets sont :

– les tâches ;

– les interruptions ;

– les synchronisations ;

– les *timers* ;

– les messages ;

– les erreurs.

La plupart des objets sont définis de façon statique à l'élaboration de l'exécutif et de l'application. En effet, le type de cible typique est un microcontrôleur, dont les performances en vitesse de traitement et mémoire sont nettement inférieures à celles d'un microprocesseur. De plus, la ligne directrice de la norme est de favoriser le déterminisme et la stabilité. Il ne faut donc pas s'attendre à trouver les mêmes fonctionnalités que dans POSIX : bien que la plupart des objets OSEK/VDX puissent être configurés en ligne, ceux-ci ne peuvent pas être détruits ou créés dynamiquement.

La mise en place d'un programme tournant sur OSEK/VDX utilise donc non seulement l'exécutif et le code source utilisateur (en C), mais aussi une description statique du système, donnée sous la forme de sources **OIL** (*OSEK Implementation Language*). Les sources OIL associées à un système fournissent à l'élaboration du programme les renseignements sur les objets susceptibles d'exister sur le système.

La figure 5.31 représente le mode de développement typique d'un système s'exécutant sur OSEK/VDX.

Ce choix, bien que moins flexible que POSIX, présente des avantages indéniables, notamment en ce qui concerne les techniques de validation, notamment temporelles, pouvant être employées.

■ Les objets OSEK/VDX

□ Les tâches

Afin de permettre un traitement optimisé des tâches simples par le noyau, OSEK/VDX distingue deux types de tâches : les **tâches basiques**, et les **tâches étendues**.

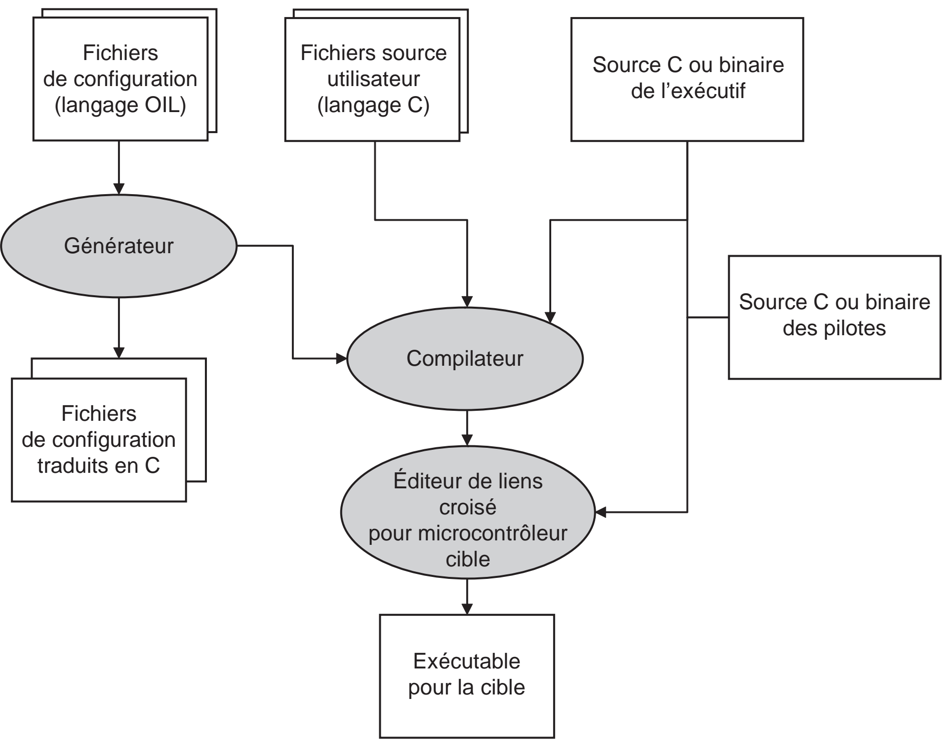


Figure 5.31 – Mode de développement typique d'un système s'exécutant sur OSEK/VDX.

Les tâches basiques ne peuvent pas se bloquer (attente d'événement, message, ressource…) entre deux activations successives. Elles ne possèdent donc pas d'état *bloquée* (figure 5.32). De plus, une tâche est créée de façon statique, il n'y a donc pas d'état *inexistante* contrairement à la figure 5.2. L'initialisation, quant à elle, est effectuée automatiquement au démarrage de l'exécutif. Notons qu'une tâche ne se termine jamais (*i.e.* son contexte, la mémoire utilisée, etc. ne sont jamais libérés) et que le mot *terminer* signifie « se mettre en attente d'activation » dans la terminologie OSEK/VDX. Ici, le terme *endormie* correspond à *suspendue*.

Les tâches étendues peuvent se bloquer durant leur exécution (figure 5.33). Typiquement, une tâche se bloque en attendant une synchronisation ou une ressource.

Comme dans la norme POSIX, une tâche est définie par une fonction. Cependant, contrairement à celle-ci, une tâche n'est pas créée dynamiquement, mais est définie statiquement dans le fichier OIL, ainsi que tous les objets de synchronisation et de communication.

Une tâche basique ne peut donc pas se bloquer pendant son exécution, ce modèle peut donc typiquement implémenter une tâche périodique (endormie ente chaque activation) ou encore une tâche en attente de message, de synchronisation, ou d'interruption à chacune de ses occurrences. Cependant, il faut noter que dans ce cas, une telle tâche ne peut pas accéder à une ressource critique, puisque le mécanisme d'exclusion est potentiellement bloquant.

© Dunod – La photocopie non autorisée est un délit.

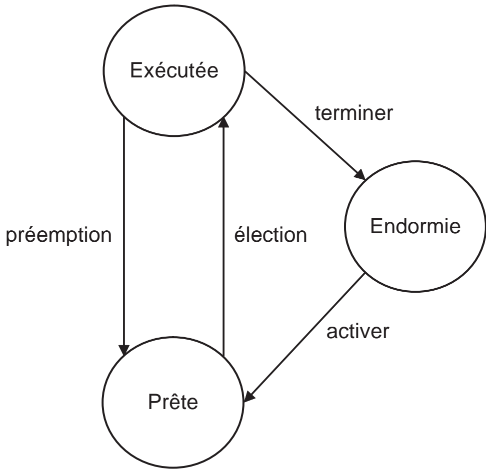


Figure 5.32 – États possibles d'une tâche basique.

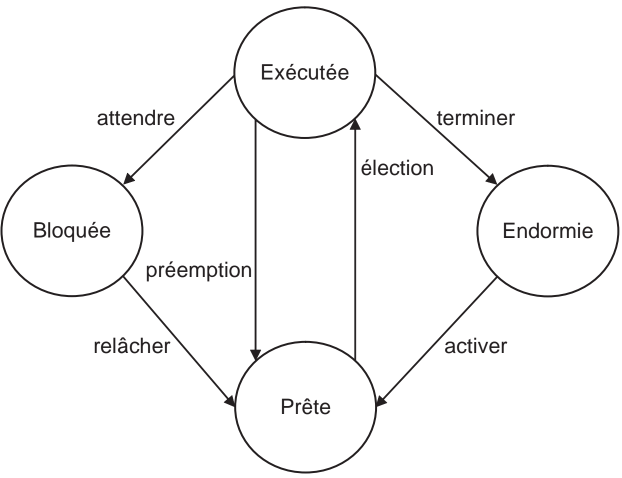


Figure 5.33 – États possibles d'une tâche étendue.

Dans le cas où une tâche peut attendre un message, une interruption, synchronisation ou encore accéder à une ressource pendant son exécution, il conviendra d'utiliser une tâche étendue.

De façon générale, l'accent est mis sur la prédictibilité du système, par conséquent, plusieurs restrictions sont appliquées. La terminaison d'une tâche ne peut provenir que de son propre appel à *TerminateTask*.

L'exécutif propose un certain nombre de services pour gérer les activations de tâches :

– *ActivateTask* permet d'activer une tâche endormie ;

– *ChainTask* permet à une tâche de se terminer en activant une autre tâche.

Synchronisation et communication de tâches

OSEK/VDX met l'accent sur les concepts du multitâche (exclusion mutuelle, communication par message, attente d'interruption) plus que sur les outils contrairement à POSIX. Il en résulte que les outils fournis correspondent à des concepts, et il s'ensuit une relative simplicité de compréhension.

Exclusion mutuelle

L'exclusion mutuelle est assurée par la définition de ressources (de façon statique dans le langage OIL), qui correspondent à des sémaphores binaires d'exclusion mutuelle, gérés avec le protocole à priorité plafond immédiat, afin d'éviter l'inversion de priorité et de borner la durée de blocage d'une tâche en attente d'une ressource.
OSEK/VDX fournit deux primitives, *GetResource* et *ReleaseResource* pour prendre et vendre une ressource. Tenter de prendre une ressource non disponible a pour effet de placer une tâche dans l'état *bloquée*.
Notons ici une spécificité d'OSEK/VDX : il existe des ressources internes qui sont prises implicitement par les tâches liées à cette ressource dès qu'elles s'exécutent. Si plusieurs tâches sont liées à la même ressource interne, on parle alors d'un **groupe de tâches**. Lorsqu'une tâche du groupe s'exécute, elle hérite automatiquement de la priorité plafond de la ressource interne (d'après le protocole à priorité plafond immédiat), c'est-à-dire de la plus haute priorité des tâches du groupe. Il en résulte qu'une tâche du groupe ne peut être préemptée que par une tâche de priorité supérieure à la priorité de la tâche la plus prioritaire du groupe. Une extension naturelle de la notion de ressource interne est le mode non préemptible des tâches : si toutes les tâches sont liées à une même ressource interne, alors une tâche prête ne peut pas être préemptée par une autre tâche, étant donné qu'elle hérite pendant son exécution de la priorité plafond de la ressource, c'est-à-dire de la priorité la plus forte des tâches.
Le langage OIL propose cependant un moyen plus simple de rendre des tâches non préemptibles, en proposant tout simplement la préemptibilité comme attribut de tâche au moment de sa définition.

Synchronisation par événements

Les événements, définis de façon statique dans le langage OIL, sont des synchronisations binaires de type $n/1$: chaque tâche étendue est liée à un certain nombre d'événements, qu'elle seule peut attendre. Un événement est d'ailleurs défini par le nom de son propriétaire et son nom propre, étant donné qu'une tâche peut être liée à plusieurs événements.
OSEK/VDX propose une primitive de déclenchement (*SetEvent*), une primitive d'attente d'événement (*WaitEvent*) qui est distinguée de l'effacement de l'événement (*ClearEvent*). Seule la tâche propriétaire d'un événement a le droit de l'attendre et de l'effacer.

Communication par message

OSEK/VDX a été créé initialement pour le monde de l'automobile, dans lequel les architectures de contrôle-commande sont de type calculateurs reliés par un réseau de communication de type réseau de terrain (comme CAN ou VAN). Les commu-

© Dunod – La photocopie non autorisée est un délit.

nications entre les tâches se calquent naturellement sur ce type d'architecture. Pour le programmeur, le fait qu'une communication ait lieu entre deux tâches du même calculateur ou bien entre deux tâches situées sur des calculateurs distincts se doit d'être le plus transparent possible, tout en utilisant au mieux les mediums de communication.

Les communications par message sont de type communication n/m (n émettrices, et m réceptrices possibles pour un message). Dans le cas d'une communication de type boîte aux lettres, si un message a m réceptrices, chacune d'entre elles est munie d'une file d'attente de messages gérée en FIFO, et consomme ses messages indépendamment des autres réceptrices (*i.e.* chaque message envoyé est déposé dans chacune des files d'attente de réceptrices de ce message).

Le type de communication peut aussi être de type tableau noir (*i.e.* la lecture est non bloquante et non destructive, et le message lu est le message le plus récemment écrit).

Généralement, la taille des messages est statique, mais il est possible, modulo quelques restrictions, de définir des messages de taille dynamique. Des messages vides peuvent aussi être transmis, dans ce cas, ils servent de synchronisation entre tâches pouvant se trouver sur des calculateurs distincts. Par conséquent, l'exécutif fournit trois primitives d'envoi de message : *SendMessage* (envoi d'un message de taille fixe), *SendDynamicMessage* (envoi d'un message de taille dynamique), et *SendZeroMessage* (envoi d'un message vide).

Un même message peut être simultanément transmis de façon interne et sur un medium de communication, si par exemple une réceptrice se trouve sur le même calculateur que l'émettrice, et qu'une autre réceptrice se trouve sur un autre calculateur. Afin d'optimiser l'utilisation du medium, plusieurs messages envoyés sur le medium de communication peuvent être regroupés dans le même paquet (typiquement, sa taille est fonction du protocole de communication sous-jacent). Ce paquet, nommé I-PDU (*Interaction layer Processor Data Unit*), constitué d'un ou plusieurs messages, pourra alors être émis physiquement à destination d'un ou plusieurs calculateurs distants. Notons que le fait que la communication soit orientée messages facilite l'utilisation d'un protocole de type CAN (voir § 4.3.2, p. 177).

La figure 5.34, extraite de la norme OSEK/VDX, montre le schéma simplifié de communication par message sur un calculateur muni de l'exécutif OSEK/VDX.

Notons la présence de filtrage possible à l'émission vers un calculateur distant (typiquement, un filtre pourra éviter les envois de valeurs identiques d'un même message sur le réseau).

De même, il est possible d'effectuer un filtrage à la réception de message, de sorte à ne prendre en compte que les messages jugés intéressants (par exemple, message différent des précédents).

Enfin, il existe différentes notifications permettant aux tâches d'être averties, soit de l'arrivée d'un message (ce qui est logique), soit du départ d'un I-PDU, ou encore d'une erreur, ou d'une violation d'échéance.

OSEK/VDX permet de définir des messages périodiques, qui seront automatiquement envoyés à chaque période. Chaque message peut être muni d'une échéance : si le message n'est pas reçu dans le délai imparti, une notification de violation d'échéance est envoyée.

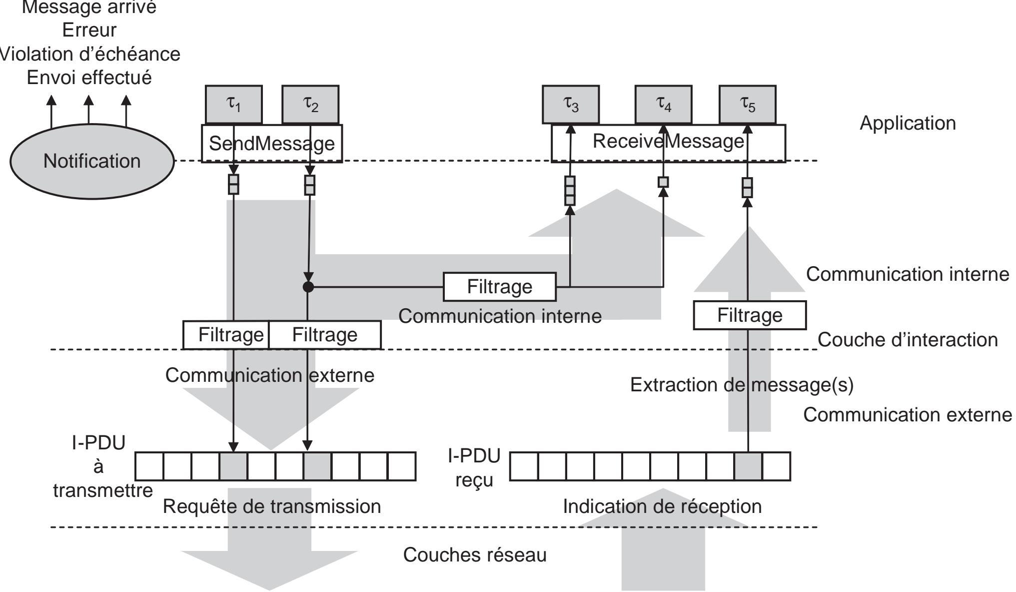


Figure 5.34 – Paradigme de communication par message OSEK/VDX.

☐ Gestion des interruptions

Deux types de traitement d'interruptions sont proposés : les ISR (*Interrupt Service Routine*) de niveau 1, n'utilisant pas de primitives de l'exécutif (sauf le masquage/démasquage d'interruptions), censées être très rapides, et les ISR de niveau 2 n'ayant pas de limitations.

Typiquement, il est impossible pour une ISR de niveau 1 de déclencher une tâche sur interruption.

Une hiérarchie de traitement est implicite : les ISR de niveau 1 sont plus prioritaires (voir § 5.3.2, p. 228) que les tâches et les ISR de niveau 2.

Comme tous les objets gérés par OSEK/VDX, la définition des ISR, et leur niveau, est faite de façon statique dans le fichier OIL.

Pendant le fonctionnement d'une application, il est aisé de désarmer/armer toutes les interruptions pouvant être désarmées (primitive *DisableAllInterrupts* et *EnableAllInterrupts*) : dans ce cas, ces interruptions sont tout simplement ignorées. De même, on peut masquer/démasquer toutes les interruptions (rappelons que le traitement d'une interruption masquée est retardé au démasquage de l'interruption) avec les primitives *SuspendAllInterrupts et ResumeAllInterrupts*. On peut aussi choisir de masquer/démasquer les interruptions liées à des ISR de niveau 2. Étant donnée la hiérarchie inhérente aux deux niveaux d'interruptions, il est impossible de masquer seulement les ISR de niveau 1.

© Dunod – La photocopie non autorisée est un délit.

☐ Horloges

À l'instar de POSIX, OSEK/VDX impose la présence d'au moins une horloge, mais en fonction de l'architecture, plusieurs horloges peuvent être disponibles.

Comme dans POSIX (voir § 5.3.1, p. 216), il est possible de définir des *timers* périodiques, ou bien *one-shot*, et cela de façon relative ou absolue.

On peut lier une tâche, un événement, ou bien une fonction (considérée presque comme une ISR de niveau 1) à un *timer* : à chaque expiration d'un *timer*, la tâche liée à celui-ci est activée, ou l'événement est déclenché ou bien la fonction est exécutée.

Les *timers* sont définis de façon statique dans le fichier OIL.

Enfin, notons qu'OSEK/VDX est un exécutif dirigé par les événements (voir § 5.2.4).

☐ Ordonnancement

Chaque tâche est munie d'une priorité, ne pouvant être modifiée que temporairement lors d'un héritage de priorité dû au protocole à priorité plafond immédiat. De même, chaque interruption se voit munie d'une priorité (normalement, les interruptions sont plus prioritaires que les tâches, et que l'ordonnanceur lui-même).

L'ordonnancement de base est de type SCHED_FIFO (voir § 5.3.1, p. 214), c'est-à-dire que les tâches prêtes sont placées dans des files d'attente FIFO gérées par priorité. L'une des spécificités OSEK/VDX est que l'ordonnancement peut être préemptif, non préemptif, ou mixte.

Si elle est **préemptible**, la tâche en exécution peut être préemptée en vue de l'élection d'une autre tâche lorsque :

- une tâche plus prioritaire est activée (*ActivateTask*) ;
- une tâche plus prioritaire devient prête suite à l'occurrence d'un événement, la libération d'une ressource, l'arrivée d'un message, etc. ;
- une tâche devient plus prioritaire (suite à la diminution de la priorité de la tâche exécutée due à la fin d'un héritage de priorité) ;
- sur une instruction exécutée par la tâche en exécution : libération explicite du processeur grâce à la primitive *Schedule* (dans ce cas, la tâche est mise à la fin de la file FIFO de son niveau de priorité), libération d'une ressource, appel d'une primitive bloquante, comme accès à une ressource non disponible, attente d'un événement non déclenché, etc. ;
- sur la fin d'une routine de traitement d'interruption.

> **Remarque**
>
> Lorsqu'une tâche est préemptée, elle reste en tête de la file d'attente de son niveau de priorité.

Une tâche **non préemptible** ne peut être préemptée en vue de l'élection d'une autre tâche qu'à sa propre initiative, c'est-à-dire à l'appel d'une primitive bloquante, ou bien à un appel explicite de *Schedule*.

Enfin, nous avons abordé le concept de groupes de tâches au paragraphe 5.3.2, p. 225, pour lesquels les tâches d'un groupe sont rendues non préemptibles les unes par rapport aux autres à l'aide de l'utilisation d'une ressource interne.

La norme recommande explicitement de rendre les tâches courtes (dont la durée est de l'ordre de quelques durées de changement de contexte) non préemptibles, et les tâches plus longues préemptibles.

□ Particularités d'OSEK/VDX

Crochets

Afin de pouvoir modifier le comportement standard du système lors d'événement particulier (erreurs, activation et terminaison de tâche, lancement et terminaison du système…), il est possible de définir des crochets (*hook*). Un crochet est une fonction exécutée avec une priorité plus haute que celle des tâches, non interruptible par les ISR de niveau 2.

Gestion des erreurs

Chaque primitive de l'exécutif retourne un code d'erreur : un code différent de E_OK correspond à une erreur d'application (tâche non définie, tâche terminée alors qu'elle possède encore des ressources…) ou bien une erreur critique (erreur matérielle, de mémoire…).
Contrairement à un noyau C standard, il n'est pas nécessaire de tester indépendamment la valeur de chaque retour de fonction de l'exécutif, puisqu'en cas d'erreur, un crochet nommé *ErrorHook* est exécuté. Il est alors possible dans cette fonction d'obtenir toutes les informations concernant l'erreur ayant eu lieu, et de prendre les dispositions nécessaires.

Modes de fonctionnement

Beaucoup d'architectures, notamment les microcontrôleurs, supportent différents types d'exécution : programme chargé en RAM, en ROM, ou dans une mémoire FLASH, etc. De plus, pendant la phase de développement, le débogage doit absolument être actif, alors que lors de la mise en fonctionnement, il est rare de consacrer de la mémoire afin d'autoriser le débogage.
Par conséquent, OSEK/VDX prévoit la définition de différents modes de fonctionnement dans les fichiers OIL.

Profils

Comme POSIX qui définit différents profils afin de permettre la normalisation d'exécutifs implémentant un sous-ensemble des fonctionnalités prévues, OSEK/VDX définit 4 profils définissant des sous-ensembles de la norme. Le nom donné aux profils est *Basic* ou *Extended Conformance Class.*
Les deux classes de conformité basiques (BCC1 et BCC2) ne fournissent que des tâches basiques. Les exécutifs BCC1 ne mémorisent pas les activations multiples d'une tâche (si une tâche est en cours de fonctionnement et qu'elle est activée, cette activation n'est pas mémorisée). Ils n'autorisent qu'une seule tâche active par niveau de priorité, et ne proposent qu'une seule ressource interne : le processeur. Il n'est donc pas possible de mixer tâches préemptibles et non préemptibles : les tâches sont soit toutes préemptibles, soit toutes non préemptibles.

© Dunod – La photocopie non autorisée est un délit.

Les classes de conformité étendue permettent l'utilisation des tâches étendues. Il est à noter que les exécutifs ECC1 ne mémorisent pas, à l'instar de BCC1, les activations multiples de tâches (alors que pour ECC2, les activations multiples de tâches basiques sont possibles). De plus, les exécutifs ECC1 n'autorisent qu'une seule tâche active par niveau de priorité.
Notons enfin que la norme impose seulement 8 niveaux de priorités de tâches distincts pour les exécutifs BCC, et 16 pour les ECC.
De même, 4 profils de communication sont définis (*Com Conformance Class*). Les profils CCCA et CCCB ne proposent que les communications internes. CCCA est limité aux messages sans file d'attente. Les profils CCC0 et CCC1 correspondent à CCCA et CCCB avec un support des communications externes.

5.3.3 La norme Ada

Le langage Ada lui-même est une norme, supportant différentes implémentations, le langage Ada est présenté au chapitre 6.
Les implémentations de la norme Ada reposent sur un exécutif ou noyau, pouvant être lui-même conforme à POSIX ou encore OSEK.

5.4 Exemples d'exécutifs temps réel

Il y a un nombre considérable d'exécutifs temps réels sur le marché, gratuits ou commerciaux. Il est très difficile d'obtenir une évaluation précise du poids de chaque exécutif ou système d'exploitation temps réel sur le marché. La figure 5.35 présente les résultats d'un sondage de 10 000 utilisateurs du site internet de l'*Open Group*.

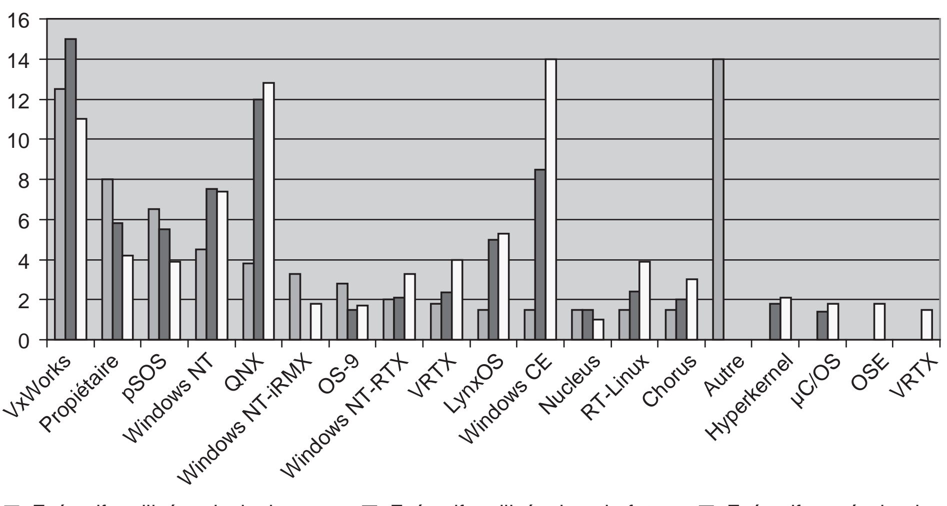


Figure 5.35 – Utilisation des exécutifs temps réel par les internautes inscrits sur le site *Open Group*, sondage effectué en 2001.

La plupart des exécutifs utilisent un développement croisé, et fournissent un certain nombre d'outils de traçage d'application, d'évaluation de performances et de débogage. La figure 5.36 présente l'architecture logicielle typique utilisée lors d'un développement croisé.

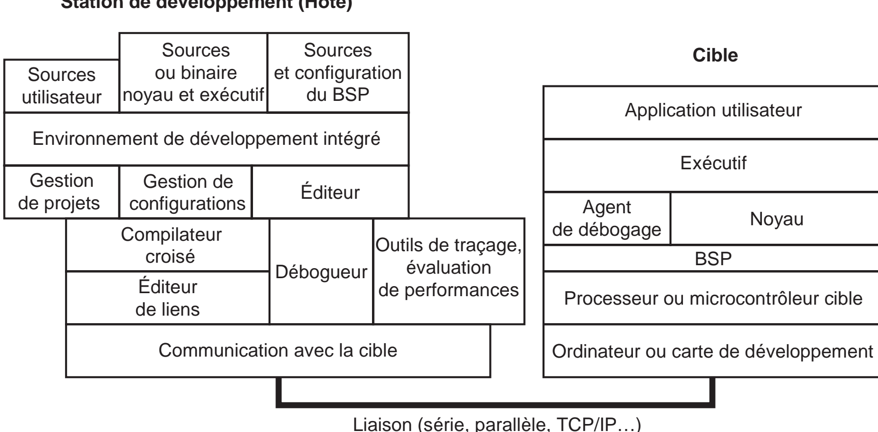


Figure 5.36 – Architecture logicielle typique lors d'un développement croisé.

Il est assez difficile d'évaluer un exécutif temps réel, car les besoins des concepteurs d'applications de contrôle-commande peuvent être radicalement différents. Il en résulte que certains noyaux ou exécutifs (offrant des possibilités similaires à celle d'OSEK/VDX, ou au profil 51 POSIX) ciblent principalement des microcontrôleurs, alors que les systèmes d'exploitation offrant une interface POSIX complète ou quasi complète ciblent principalement les micro-ordinateurs.
Le tableau 5.4 présente quelques critères à prendre en compte.

5.4.1 VxWorks 5.x

VxWorks est l'exécutif temps réel le plus répandu sur le marché. Certains éléments descriptifs ci-après peuvent être absents sur certaines cibles.

– **Nom** : *VxWorks.*
– **Type** : micro noyau et tâches de service, permettant un développement incrémental (possibilité de modifier le code applicatif sans redémarrer la cible).
– **Société** : *WindRiver.*
– **Noyau** : *Wind.*
– **Environnement de développement** : *Tornado II.*
– **Systèmes hôtes** : *Win32* (*Windows95/98/NT/2000/XP…*), *Solaris*, *HP-UX*, *Linux.*
– **Cibles** : Familles x86, PowerPC, Coldfire, m68k, Arm, StrongArm, MIPS, SuperH… Chaque cible dispose de plusieurs dizaines de BSP spécifiques.
– **Support multiprocesseur** : oui avec l'extension VxMP.

© Dunod – La photocopie non autorisée est un délit.

Tableau 5.4 – Critères d'évaluation d'un exécutif temps réel.

Interfaces de programmation normalisées	La plupart des exécutifs proposent plusieurs interfaces de programmation, compatibles avec telle ou telle norme temps réel. Ces interfaces ont-elles été certifiées ? Peut-on programmer en POSIX, si oui, sur quel ensemble de normes, sur quel profil ? Peut-on programmer en OSEK/VDX, si oui, sur quel profil ? Peut-on programmer en ITRON ? Si oui, sur quel profil ?
Richesse des bibliothèques de programmation	Y a-t-il un support des calculs à virgules flottantes ? Est-il logiciel ou matériel ? Y a-t-il une communauté de développeurs mettant en commun leur savoir-faire en partageant du code ?
Langages de programmation supportés	Au minimum, l'assembleur est supporté, la plupart des exécutifs sont accompagnés de compilateurs permettant de programmer en C, voire en C++ ou encore en Ada.
Cibles supportées	Microprocesseurs et microcontrôleurs supportés.
BSP (*Board Support Package*)	Librairies spécifiques aux cartes de développement sur microcontrôleur, aux différentes spécificités et optimisations des microprocesseurs, etc.
Certification	Un crédit supplémentaire est accordé aux exécutifs certifiés suivant des normes comme la DO-178B.
Noyau, exécutif ou système d'exploitation ?	Les noyaux et exécutifs, plus légers, imposent un développement croisé (station de développement et cible sont distincts), alors qu'un système d'exploitation permet un développement sur la cible elle-même.
Dirigé par le temps ou les événements	Le noyau se base-t-il sur un *tick* (dirigé par le temps) ou sur la programmation d'horloges (dirigé par les événements) pour gérer le réveil des tâches ?
Architecture du noyau	On préférera souvent un noyau de type noyau et modules ou tâches de service à une structure monolithique.
Tolérance aux fautes	Possibilité d'être informé des fautes et de les traiter.
Support pour architecture mono ou multiprocesseur	
Type d'objets actifs	Support des architectures logicielles multitâches, multiprocessus, ou les deux.
Algorithme d'ordonnancement	Déterministe, à priorités fixes ou variables, durée d'un changement de contexte (relativement à l'architecture matérielle sous-jacente).
Protocole de gestion de ressources	Support du protocole à priorités plafonds (de préférence) ou héritées.

Nombre de niveaux de priorités	Typiquement, cela va de 8 au minimum pour certains noyaux à 256 ou plus pour d'autres. Comment l'ordonnanceur gère-t-il les tâches de même niveau de priorité (FIFO, tourniquet) ?
Nombre maximal de tâches	Typiquement, cela va de 8 au minimum pour certains noyaux à 256 ou plus pour d'autres.
Support mémoire	Empreinte mémoire (taille du noyau lui-même, et des différents modules à intégrer). Taille maximale de la mémoire (pile) allouée à une tâche ou à un processus (code, pile et tas). Protection de la mémoire si multiprocessus. Allocation dynamique de mémoire. Mémoire virtuelle.
Gestion des interruptions	Interruptions hiérarchisées ou non, possibilité ou non pour une ISR d'être interrompue, flexibilité de programmation.
Gestion des horloges	Nombre et précision des horloges, granularité temporelle, possibilité de travailler en durées et en dates.
Support réseau	Support pour TCP/IP, CAN…
Entrées/Sorties	Facilité d'utilisation des interfaces d'entrées sorties. Existence des pilotes de périphériques utilisés.
Systèmes de fichiers	L'exécutif gère-t-il les systèmes de fichiers, si oui sur quel modèle (hiérarchie de répertoires ou à plat, système compatible Unix, Windows…)
Gestion d'une console alphanumérique ou graphique	La plupart des exécutifs permettent la mise en œuvre d'une console alphanumérique (soit directement sur un écran, soit via une communication série RS-232). Certains permettent un affichage graphique (soit directement sur un écran dans le cas des systèmes d'exploitation, soit sur un écran séparé).
Facilité de développement	Quels sont les systèmes d'exploitation à partir desquels on peut développer sur l'exécutif ? Y a-t-il un environnement de développement intégré (éditeur permettant la compilation, le transfert du code et le débogage distant dans la même application) ? Peut-on modifier le code source sans réinitialiser la cible (développement incrémental) ?
Extensibilité du noyau	Le système est disponible sous forme de sources pouvant être modifiés, ou bien sous forme d'un fichier binaire immuable ? Peut-on associer (*hook*) des traitements spécifiques aux différents événements (changement de contexte, début et fin de tâche, etc.) ?
Coût, *royalties*, et support technique et/ou communautaire	Coût de l'acquisition du système et de l'environnement de développement, coût éventuel du déploiement d'applications utilisant le système. Le support technique est un facteur déterminant lors de la mise en œuvre d'un exécutif, de même que la largeur de la communauté des utilisateurs. En effet, plus il y a d'utilisateurs, plus on trouvera de forums d'échanges d'expériences, de code source, etc.

- **Interface native de programmation** : fonctions spécifiques.
- **Interface POSIX** : limitée au profil 52, POSIX 1003.1b (limitée au fait qu'il n'y a pas de support multiprocessus), 1003.1c (*pthread*) et un sous-ensemble basique de 1003.1.
- **Interface OSEK/VDX** : non disponible pour *VxWorks*, mais disponible pour le système « frère » *OSEKWorks*.
- **Interface ITRON** : *a priori* non, mais quelques sites internet japonais semblent parler d'une interface ITRON compatible avec *VxWorks*.
- **Langages de programmation supportés** : C/C++ de base, mais plusieurs autres langages sont disponibles via des produits de sociétés tiers (par exemple, la société *Aonix* propose un compilateur Ada pour VxWorks).
- **Support multiprocessus** : non.
- **Gestion du temps** : dirigé par le temps.
- **Gestion mémoire** : non protégée par défaut, protection possible avec l'extension VxVMI, allocation dynamique possible.
- **Support réseau** : la plupart des protocoles réseaux sont supportés.
- **Systèmes de fichiers** : FAT, NFS, raw, TrueFFS.
- **Niveaux de priorité** : 256 (plus le numéro est petit, plus la priorité est élevée).
- **Nombre de tâches** : limité uniquement par la taille mémoire.
- **Type** : commercial.
- **Résolution d'horloge** : 60 Hz par défaut, mais peut descendre jusqu'à 1 000 Hz (ou plus avec un surcoût processeur important).
- **Ordonnancement** : à priorités, avec un tourniquet par niveau de priorité.
- **Priorités variables** : oui.
- **Protocole de gestion de ressources** : protocole à priorités héritées.
- **Remarques** : De nombreux outils de suivi permettent d'analyser de façon détaillée l'exécution sur la cible, représentant sur un graphe aussi bien l'ordonnancement que les appels systèmes.

VxWorks utilise donc nativement une interface de programmation multitâche propriétaire. L'interface POSIX est implémentée par les primitives natives de *VxWorks*. Il est à noter qu'il est très difficile d'utiliser un temps absolu en utilisant les primitives natives de *VxWorks*, et qu'il faut pour cela utiliser les primitives POSIX (comme *nanosleep*).

VxWorks se caractérise aussi par la possibilité très simple, à l'aide de l'environnement de développement *Tornado II*, de configurer à loisir le *BSP* utilisé. Ainsi, du profil PE 52 disponible, il est relativement simple d'alléger la fabrication du noyau pour se rapprocher d'un profil plus léger lorsque la cible possède peu de ressources.

■ Gestion du temps

La résolution d'horloge est limitée par la façon dont VxWorks® gère le temps. L'unité de base est le *tick* : la fréquence d'occurrence des *ticks* peut être programmée, ce qui a pour effet de programmer le matériel de sorte à générer une interruption

d'horloge périodique. À chaque interruption d'horloge correspond un *tick* (figure 5.37), VxWorks® prend alors la main afin d'effectuer l'ordonnancement du système, de gérer les réveils éventuels des tâches (une tâche peut s'endormir afin d'attendre un certain nombre de *ticks*), etc.

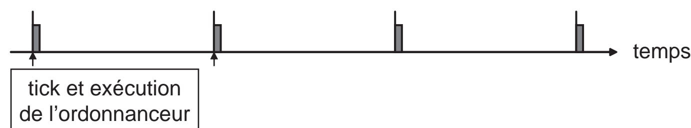


Figure 5.37 – Gestion du temps par *VxWorks®*.

Bien que cela soit en général possible, il est déconseillé d'augmenter la fréquence des *ticks* au-delà d'1 kHz (ce qui offre une résolution d'1 milliseconde) car cela impliquerait un surcoût processeur important. Le principal problème dû à cette implémentation est que la granularité du système est relativement grossière par rapport aux noyaux dirigés par les événements.

L'interface de programmation VxWorks® est relativement simple à comprendre et à manipuler. Lors de la phase de développement, on utilise généralement une tâche *tShell* fournissant un ***shell*** (invite de commande permettant à l'utilisateur de dialoguer avec l'exécutif) qui permet, via une console contrôlée par la cible ou bien par l'hôte, de lancer directement des fonctions C chargées sur la cible, et de gérer les tâches (obtenir des informations, tuer les tâches…). Il est aussi très facile, via le support réseau très étendu, d'accéder au système de fichiers de l'hôte ou d'une autre machine du réseau.

Différents choix très pratiques ont été effectués : ainsi, la tâche *tLogTask*, permettant d'enregistrer une trace d'exécution (*log*) peut être utilisée à l'aide de la fonction *logMsg* afin d'afficher ou enregistrer du texte dans un fichier, ce qui facilite le suivi du fonctionnement de l'application sans appeler directement des fonctions d'affichage ou d'écriture dans un fichier (*fprintf*), c'est-à-dire sans ajouter d'appel suspensif perturbant l'ordonnancement de l'application.

■ Gestion des tâches

Les états possibles d'une tâche VxWorks®, ainsi que les noms de primitives natives impliquant les transitions entre ces états, sont donnés sur la figure 5.38.

La mémoire est partagée par l'exécutif et toutes les tâches, comme cela est présenté sur la figure 5.39. Chaque tâche VxWorks®, comme chaque tâche utilisateur, est caractérisée uniquement pas sa pile : elle contient donc les variables locales à la tâche ainsi que son contexte d'exécution (voir § 4.2.3, p. 154). Notons que les interruptions sont traitées en utilisant une pile prédéfinie.

Les tâches sont créées (*taskInit* et *taskActivate* ou *taskSpawn* qui est la combinaison des deux actions) et détruites (*taskDelete*) dynamiquement. Lors de la création, étant donnée la gestion mémoire employée, il est nécessaire de spécifier explicitement la taille de la pile à allouer à une tâche, ainsi que des paramètres (10 en tout) à passer à la tâche. Il est aussi nécessaire de spécifier explicitement si la tâche utilise les registres

© Dunod – La photocopie non autorisée est un délit.

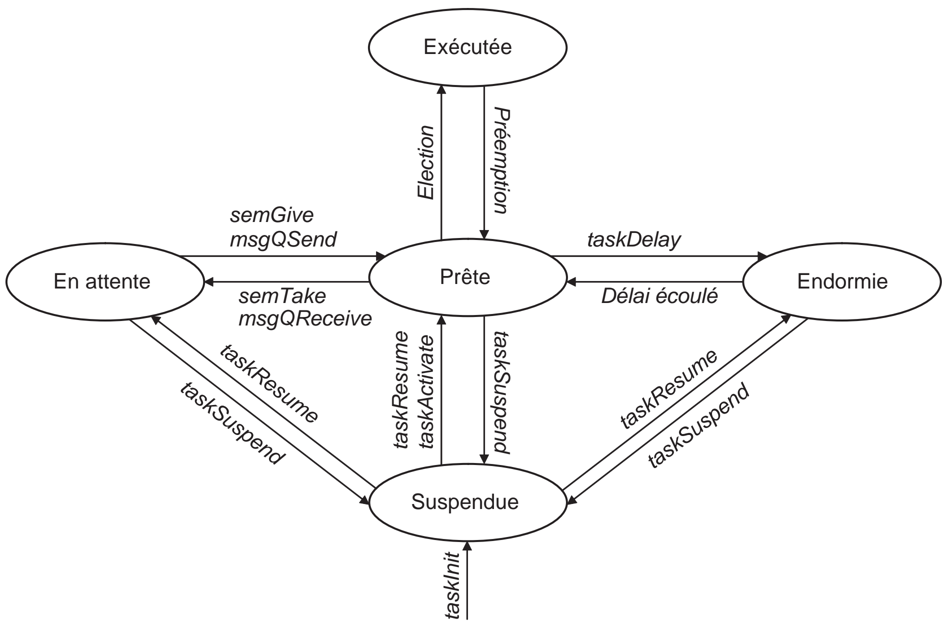


Figure 5.38 – États d'une tâche VxWorks.

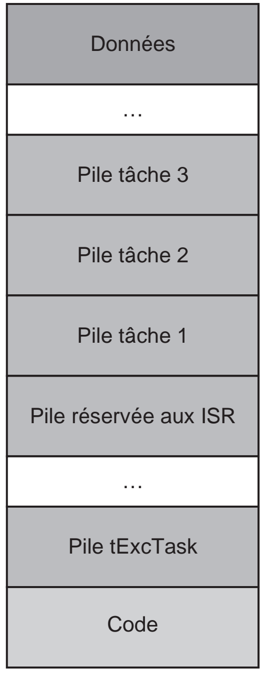


Figure 5.39 – Modèle de gestion mémoire
(les positions relatives sont données à titre purement indicatif).

de calculs à virgule flottante afin de déterminer si le changement de contexte d'une tâche nécessite leur sauvegarde. Afin d'éviter le risque qu'une ressource ne soit pas libérée avant la terminaison d'une tâche, il existe un mécanisme de protection de tâche : entre les instructions *taskSafe* et *taskUnsafe*, une tâche ne peut pas être tuée.

Une tâche peut modifier sa priorité ou la priorité d'une autre tâche à l'aide de l'instruction *taskPrioritySet*. Elle peut aussi se rendre non préemptible (mais interruptible par le traitement des interruptions) entre les instructions *taskLock* et *taskUnlock*.

Les tâches s'exécutent en mode détaché, ce qui explique que le code suivant soit erroné :

```
void UneTache(int *tableauEntiers, int tailleTableau) {
/* Fonction implémentant une tâche affichant un tableau
   Entrée : un tableau d'entier indexé de 0 à
   tailleTableau-1 */
      int i ;
      for (i=0 ; i<tailleTableau ; i++) {
            printf("%d",tableauEntiers[i]);
      }
}
void LancerTache() {
      int tableau[]={0,3,5} ; /* Tableau de 3 entiers */
      taskSpawn("affiche tableau",150,0,2048,(FUNCPTR)
      UneTache, (int)tableau,3,0,0,0,0,0,0,0,0) ;
      /* Création et activation de la tâche nommée "affiche
      tableau", implémentée par la fonction UneTache, de priorité
      150, n'utilisant pas les virgules flottantes, munie d'une
      pile de 2 Ko, prenant en paramètres un tableau d'entiers et
      sa taille. Noter que VxWorks fournit 10 paramètres entiers,
      et que la taille d'un pointeur est la taille d'un entier,
      ce qui permet d'effectuer sans problèmeune coercion de type.
      Noter aussi la nécessité de donner 10 paramètres, même si
      seulement 2 sont utiles à la fonction */
}
```

Ce petit programme, lorsqu'il est chargé sur la cible, permet de lancer, par exemple à l'aide du *shell*, la fonction *LancerTache*. Là il y a de très fortes chances que les entiers affichés ne correspondent pas du tout aux entiers attendus. En effet, si *LancerTache* est exécutée dans le contexte de la tâche *tShell*, la variable *tableau* est allouée sur la pile de cette tâche. Lorsque la fonction se termine, cette variable se voit écrasée par la suite de l'exécution de *tShell*. Lorsque la tâche créée s'exécute, l'adresse du tableau correspond à un tableau qui a été écrasé par d'autres valeurs dans la pile de *tShell*.

Il est donc nécessaire de déclarer le tableau à passer en paramètres dans le tas, c'est-à-dire en variable globale. La version correcte de ce petit programme est donc :

```
int tableau[]={0,3,5} ; /* Tableau de 3 entiers alloué
dans le tas */ void LancerTache() {
      taskSpawn("affiche tableau",150,0,2048,
      (FUNCPTR)UneTache, (int)tableau,3,0,0,0,0,0,0,0,0) ;
}
```

© Dunod – La photocopie non autorisée est un délit.

■ Outils de synchronisation et de communication

En plus de l'interface POSIX 1003.1b et 1003.1c, sur laquelle nous ne revenons pas ici, VxWorks® propose ses propres outils de communication et de synchronisation, qui sont plus rapides d'après la documentation.
Trois types de sémaphores sont proposés : le sémaphore binaire (créé par *semBCreate*), le sémaphore compteur (créé par *semCCreate*) et le sémaphore d'exclusion mutuelle (créé par *semMCreate*) sont pris ou vendus par les mêmes primitives *semTake* et *semGive*. La spécificité du sémaphore d'exclusion mutuelle est qu'il est possible de lui adjoindre le protocole de gestion de ressources à priorité héritée.
Notons qu'il est possible lors de la prise de sémaphore de borner l'attente maximale de celui-ci.
La communication par message est assurée à l'aide de boîtes aux lettres FIFO sans écrasement, gérant deux niveaux de priorité de messages (normal et urgent). Une boîte aux lettres est créée avec *msgQCreate* qui se voit donner la taille de la file, la taille maximale d'un message, et un paramètre disant si les tâches en attente doivent être traitées en FIFO ou suivant leur niveau de priorité. L'envoi de message (*msgQSend*) et l'attente de message (*msgQReceive*) peuvent être bornés dans leur temps d'attente maximal.

■ Horloge et chiens de garde

Comme nous l'avons dit en introduction du paragraphe 4.1, la granularité de *VxWorks* est le *tick*. Cela rend la manipulation du temps assez grossière, et l'instruction d'attente fournie par *VxWorks* est une instruction d'attente relative *taskDelay* qui prend en paramètre un nombre (entier) de *ticks*. Le seul moyen d'accéder à un temps absolu (modulo la granularité des *ticks*) est d'utiliser l'interface POSIX.
Les chiens de garde (*watchdogs*) permettent de programmer l'exécution d'une fonction qui se déclenchera au bout d'un certain nombre de *ticks*. Il faut noter que la fonction exécutée par un chien de garde n'a pas le droit d'utiliser d'instruction bloquante ou suspensive (une entrée/sortie par exemple), puisqu'elle est exécutée dans le contexte de la tâche *tExcTask*, de priorité maximale.

■ Gestion des interruptions

La gestion des interruptions est extrêmement simple puisque *VxWorks* permet d'associer des fonctions (ISR) aux interruptions à l'aide de la fonction *intConnect*. Comme les chiens de garde, les ISR n'ont pas le droit aux instructions bloquantes ou suspensives car elles sont exécutées dans le contexte de *tExcTask*. Elles sont donc généralement implémentées de sorte à déclencher une synchronisation permettant d'activer une tâche de traitement de l'interruption (DSR), comme cela est décrit au paragraphe 5.2.3, p. 206.

■ Avantages et inconvénients

L'avantage indéniable de *VxWorks* sur la plupart de ses concurrents est sa simplicité de mise en œuvre grâce aux très nombreux *BSP* existants, aux fonctionnalités de développement incrémental, à la possibilité d'intégrer des tâches de service comme une tâche de *shell*, une tâche de trace, une tâche de gestion réseau, etc. De plus,

pour un novice, l'interface de programmation est très simple à prendre en main. Les outils de suivi (notamment *WindView*) permettent d'obtenir des informations très utiles, et très pédagogiques, sur l'application en fonctionnement. Notons aussi qu'il est relativement aisé de configurer le *BSP* choisi à son bon vouloir. Enfin, notons la présence d'un simulateur permettant de tester une application sur l'hôte, dont l'inconvénient, dans sa version de base, est qu'il n'implémente pas les protocoles réseau.

L'inconvénient majeur de *VxWorks* est sa granularité temporelle grossière, et la difficulté de gérer un temps absolu. Sa granularité temporelle est due au fait que le noyau est géré par le temps (voir § 5.2.4), ce qui implique un surcoût important dès lors que l'on souhaite travailler à granularité fine. Il est bien sûr possible d'installer soi-même une gestion plus fine du temps, en programmant une horloge physique, et en travaillant sur les interruptions générées par celle-ci. Enfin, l'interface de programmation spécifique est moins intéressante que ne le serait une interface normalisée.

5.4.2 OSEKturbo

OSEKturbo est un exécutif temps réel conforme à OSEK/VDX, profils ECC1 et communication CCCB (communication interne uniquement) dans la version de base. Comme la norme, celui-ci, extrêmement léger et rapide, est principalement développé pour des microcontrôleurs (8, 16 ou 32 bits).

– **Nom** : *OSEKturbo.*
– **Type** : micro noyau.
– **Société** : *Metrowerks.*
– **Noyau** : *OSEKturbo.*
– **Environnement de développement** : *CodeWarrior.*
– **Systèmes hôtes** : *Win32* (*Windows95/98/NT/2000/XP…*).
– **Cibles** : plusieurs microcontrôleurs PowerPC, ARM, NEC V850, C16x…
– **Support multiprocesseur** : non.
– **Interface native de programmation** : OSEK/VDX.
– **Interface POSIX** : non.
– **Interface OSEK/VDX** : BCC1, ECC1, communication CCCA, CCCB.
– **Interface ITRON** : non.
– **Langages de programmation supportés** : C.
– **Support multiprocessus** : non.
– **Gestion du temps** : dirigé par les événements.
– **Gestion mémoire** : non protégée.
– **Support réseau** : de base, non.
– **Systèmes de fichiers** : aucun.
– **Niveaux de priorité** : 62.
– **Nombre de tâches** : 64 tâches actives au plus.
– **Type** : commercial.

© Dunod – La photocopie non autorisée est un délit.

- **Résolution d'horloge** : définie par l'horloge interne du microcontrôleur, ou bien par des horloges matérielles spécifiques. Possibilité d'obtenir une résolution en dessous de la microseconde.
- **Ordonnancement** : conforme à OSEK/VDX.
- **Priorités variables** : non (sauf lors d'un héritage de priorité dû au protocole à priorité plafond).
- **Protocole de gestion de ressources** : protocole à priorité plafond immédiat.
- **Remarques** : De nombreux outils de suivi permettent d'analyser de façon détaillée l'exécution sur la cible, l'ordonnancement, les temps de réponse des tâches.

OSEKturbo est un micro noyau extrêmement léger permettant de générer des programmes de quelques centaines d'octets ou quelques Ko. Il est complètement compatible OSEK/VDX, ce qui facilite sa prise en main. Nous pouvons aussi citer, fait rarissime, la présence d'un outil d'analyse d'ordonnançabilité.
Les inconvénients sont les inconvénients de la norme OSEK/VDX (absence de système de fichiers par exemple, définition statique de l'application) et ses avantages sont les avantages de la norme OSEK/VDX, soit principalement le déterminisme, et la possibilité d'atteindre une très grande résolution d'horloge. En effet, OSEK/VDX est dirigé par les événements (voir § 5.2.4) : les interruptions horloges impliquent uniquement la décrémentation des *timers*, et seule l'expiration d'un *timer* implique un travail supplémentaire.

5.4.3 RTEMS

RTEMS (*Real-Time Executive for Multiprocessor Systems*) version 4.6 est l'un des exécutifs temps réel gratuits et *open source* les plus riches que l'on puisse trouver. Issu initialement d'un projet militaire (il a notamment été développé pour le contrôle de missiles), cet exécutif s'est enrichi au fil des années et des versions.

- **Nom** : *RTEMS*.
- **Type** : noyau et tâches de service.
- **Société** : *OAR Corporation*.
- **Noyau** : *RTEID*/*ORKID*.
- **Environnement de développement** : divers outils en ligne de commande.
- **Systèmes hôtes** : *Unix*, *Linux*, *Win32* (sous l'environnement *Cygwin*, portage des outils *Unix* sur système *Win32*).
- **Cibles** : Chaque utilisateur étant à même de développer un BSP, la plupart des processeurs et microcontrôleurs sont supportés (de façon officielle ou non). Il faut cependant s'attendre, lorsque l'on utilise un BSP non officiel, à être confronté à quelques bogues.
- **Support multiprocesseur** : oui.
- **Interface native de programmation** : interface spécifique.
- **Interface POSIX** : limitée au profil 52, implémentation presque complète de POSIX 1003.1b (limitée au fait qu'il n'y a pas de support multiprocessus), 1003.1c et 1003.1h.

- **Interface OSEK/VDX** : *a priori* non.
- **Interface ITRON** : une partie de l'interface uITRON 3.0 (l'interface est en cours de développement).
- **Langages de programmation supportés** : C/C++, Ada.
- **Support multiprocessus** : non.
- **Gestion du temps** : dirigé par le temps.
- **Gestion mémoire** : non protégée (il est possible de protéger certains segments contre l'écriture).
- **Support réseau** : la plupart des protocoles réseaux classiques sont implémentés.
- **Systèmes de fichiers** : *In Memory File System* (*IMFS*), FAT12, FAT16, FAT32, NFS et TFTP pour accès distant à un système de fichiers.
- **Niveaux de priorité** : 255 (1 étant la priorité la plus forte).
- **Nombre de tâches** : limité uniquement par la taille mémoire.
- **Type** : *GPL* (*General Public License*) et *Open Source*.
- **Résolution d'horloge** : définie par l'horloge interne du processeur ou microcontrôleur, ou bien des horloges matérielles spécifiques. Possibilité d'obtenir une résolution en dessous de la microseconde.
- **Ordonnancement** : à priorités, avec un tourniquet par niveau de priorité, *Rate Monotonic*.
- **Priorités variables** : oui.
- **Protocole de gestion de ressources** : priorité héritée et priorité plafond immédiat.
- **Remarques** : le système étant *Open Source*, la communauté de développeurs met largement son expérience en commun, ce qui compense le manque d'environnement de développement intégré.

Les concepts de base de RTEMS sont assez proches des concepts de base de VxWorks®, notamment dans la gestion du temps basée sur des *ticks*.
Il souffre donc des mêmes avantages et inconvénients (notamment au niveau de la gestion du temps par *ticks* impliquant une granularité grossière, et obligeant à recourir à des horloges physiques et aux interruptions) que *VxWorks*. Il a l'avantage d'être gratuit et *Open Source* mais l'inconvénient de ne pas fournir d'environnement de développement intégré simple à mettre en œuvre. Enfin, citons la possibilité d'utiliser un compilateur Ada sans avoir à acquérir une extension.

© Dunod – La photocopie non autorisée est un délit.

5.4.4 RTLinux

RTLinux est l'un des systèmes d'exploitation temps réel basé sur Linux. Ses principales caractéristiques sont :

- **Nom** : *RTLinuxPro*/*RTLinuxFree*.
- **Type** : micro noyau accompagné d'un système d'exploitation *Linux*.
- **Société** : *FSMLabs*/communauté de développeurs.
- **Noyau** : *RTCore*.

– **Environnement de développement** : outils de développement du monde *Unix*.
– **Systèmes hôtes** : le système lui-même, avec *Linux* ou *Unix*. Ce système peut tourner sur différents processeurs (famille x86 à partir du 386, *PowerPC*, *ARM*, *StrongARM*, *XScale*, *MIPS*, *Alpha*…).
– **Cibles** : le système hôte.
– **Support multiprocesseur** : non.
– **Interface native de programmation** : POSIX 1003.1c et 1j (*pthread*).
– **Interface POSIX** : profil 51.
– **Interface OSEK/VDX** : non.
– **Interface ITRON** : non.
– **Langages de programmation supportés** : C/C++, Ada en partie.
– **Support multiprocessus** : non (supportés bien entendu dans la partie *Linux* non temps réel).
– **Gestion du temps** : dirigé par les événements.
– **Gestion mémoire** : la mémoire des tâches temps réel peut être protégée dans la version pro avec l'extension *PSDD* (certaines tâches temps réel peuvent se voir allouer un espace mémoire protégés de la zone non temps réel).
– **Support réseau** : la pile UDP/IP sur Ethernet, et CAN.
– **Systèmes de fichiers** : non (profil 51) mais l'accès aux systèmes de fichiers est effectué par le noyau *Linux* non temps réel chargé au-dessus du noyau *RTCore*.
– **Niveaux de priorité** : 1 000 000.
– **Nombre de tâches** : limité uniquement par la taille mémoire.
– **Type** : commercial pour *RTLinuxPro*, *GPL* (*General Public License*) pour *RTLinux-Free*.
– **Résolution d'horloge** : définie par l'horloge interne du processeur. Possibilité d'obtenir une résolution de l'ordre de la microseconde.
– **Ordonnancement** : conforme à POSIX.
– **Priorités variables** : oui.
– **Protocole de gestion de ressources** : priorité plafond immédiat.

Initialement, RTLinux a été développé sur le mode de l'*Open Source*. Les deux personnes à l'origine du projet ont ensuite fondé la société *FSMLabs* qui développe la version « pro » de *RTLinux*, tout en conservant la version libre. Suite à cela, la version libre évolue peu. Ceci explique qu'il y ait deux versions de *RTLinux*. En tant que système d'exploitation, *RTLinux* se base sur un micro noyau temps réel *RTCore* (conforme au profil 51 de la norme POSIX) situé entre le matériel et un système d'exploitation *Linux* très légèrement modifié. Le noyau émule alors le matériel et les horloges (figure 5.40) pour le noyau *Linux*, qui obtient le processeur quand il n'est pas utilisé par une tâche temps réel. Ce concept permet d'allier la flexibilité d'un système d'exploitation (*Linux*) au déterminisme d'un système d'exploitation temps réel.

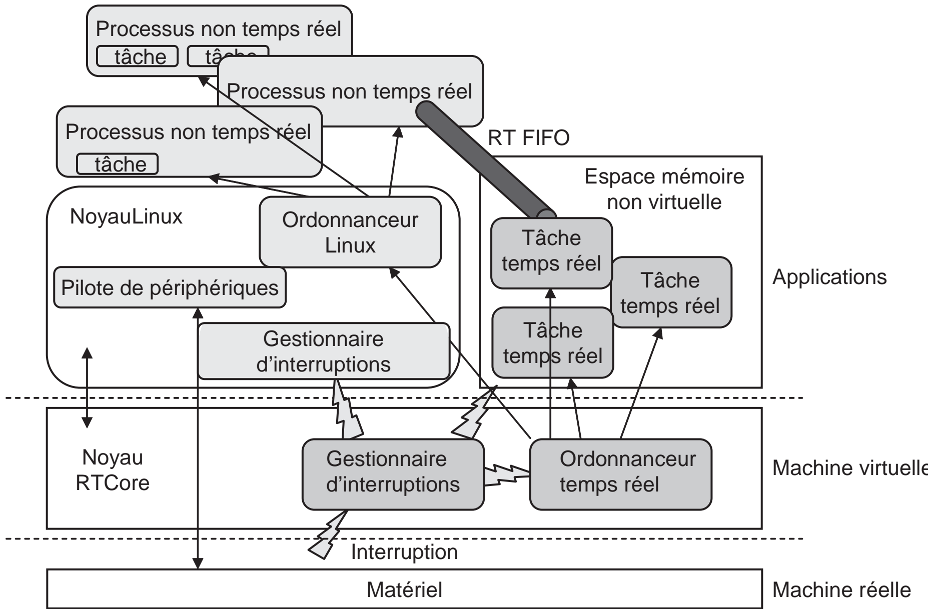


Figure 5.40 – Architecture *RTLinux*.

RTLinux est l'une des adaptations de *Linux* au temps réel. Différentes techniques sont utilisées par les systèmes d'exploitation temps réel basés sur Linux pour réduire la latence (voir § 5.1.1) du noyau.

La première consiste à modifier les primitives du noyau afin d'insérer des points de préemption ou de diminuer au maximum les parties non préemptibles de celui-ci (*Patch Low Latency*, *Preempt Kernel*).

La seconde, utilisée par *RTLinux* et le système d'exploitation *RTAI*, resté quant à lui logiciel libre, consiste à utiliser un micro noyau temps réel fournissant une machine virtuelle à un noyau *Linux*. Celui-ci est légèrement modifié de sorte à l'empêcher de masquer les interruptions ou de les détourner directement (rôle du gestionnaire d'interruptions du noyau *RTCore*), et les horloges qu'il utilise sont en fait simulées par le noyau *RTCore*. Ainsi, la granularité temporelle de *RTLinux* peut descendre jusqu'à la microseconde sans surcoût processeur, puisque seules les interruptions correspondant à des expirations de *timers* impliquent un traitement particulier de l'exécutif (qui peut d'ailleurs consister à générer une interruption horloge simulée pour *Linux*).

Le noyau *Linux* est ordonnancé dans les temps creux laissés par les tâches temps réel *RTLinux*, ce qui permet d'exécuter des processus non temps réel sur la partie *Linux*. Les processus non temps réel sont des processus *Linux* classiques, par conséquent, ils ont accès à toute la richesse de ce système d'exploitation généraliste (accès au système de fichiers, aux périphériques non temps réel, au réseau, à une interface graphique, etc.).

Afin de permettre une communication entre les tâches temps réel et des processus non temps réel sans perdre le déterminisme lié au noyau temps réel, *RTLinux* fournit

© Dunod – La photocopie non autorisée est un délit.

un outil de communication par messages : les **files temps réel** (*RT FIFO*) et la mémoire partagée (*mbuff*) entre processus non temps réel et tâches temps réel.
Une file temps réel est vue comme un fichier de caractères du côté du processus non temps réel l'utilisant (ouverture, lecture et écriture bloquantes ou non, fermeture), et comme un tube (voir § 2.2.2, p. 196) non bloquant en lecture du côté de la tâche temps réel l'utilisant.
Afin de palier le manque de boîte aux lettres dans la norme POSIX 1003.1c, *RTLinux* propose aussi un mécanisme de boîte aux lettres spécifique (*rt_mq*).
Les avantages de *RTLinux* sont liés au fait qu'il est conforme au profil 51 de POSIX, à sa gestion du temps, et surtout à sa cohabitation avec *Linux*. Son inconvénient majeur est lié à l'empreinte mémoire énorme prise par *Linux*, d'où une limitation des cibles potentielles.

6 • PROGRAMMATION DES SYSTÈMES MULTITÂCHES

Après une brève présentation des langages C, Ada, et LabVIEW, ce chapitre présente des règles de passage d'une conception DARTS à une application de contrôle-commande.

Nous supposons que le lecteur est familier avec au moins un langage de programmation impératif ou bien flots de données. Par conséquent, la présentation des trois langages de programmation supports est faite en commun, de sorte qu'avec la connaissance d'un des trois langages (ou d'un langage proche), le lecteur puisse rapidement se faire une idée des deux autres.

6.1 Programmation C, Ada et LabVIEW

6.1.1 Présentation générale des trois langages

Nous avons choisi dans cet ouvrage de présenter l'implémentation de systèmes de contrôle-commande à travers trois langages de programmation :

– le langage C, car parmi les représentants des langages impératifs, c'est le langage le plus répandu pour l'implémentation de systèmes de contrôle-commande. De plus, il est le langage de référence pour les systèmes d'exploitation, et est à la base de différentes normes comme POSIX et OSEK/VDX ;

– le langage Ada, recommandé dans les systèmes à haut niveau de sûreté ;

– le langage LabVIEW, langage graphique flots de données, est quant à lui très utilisé dans le contrôle de procédés industriels.

Ce chapitre n'a pas pour objectif de présenter exhaustivement les normes et langages sus-cités, mais de les présenter dans le but d'implémenter une conception DARTS.

■ Le langage C

Le langage C a vu le jour à la fin des années 60. Son rôle initial était de permettre la portabilité de la majeure partie du code d'un système d'exploitation d'une architecture matérielle à une autre. En 1973, il est utilisé avec succès pour écrire un système d'exploitation. Depuis, son utilisation n'a cessé de croître et il est le langage utilisé dans l'implémentation de la majeure partie des systèmes d'exploitation. C'est aujourd'hui encore l'un des langages de programmation les plus utilisés.

© Dunod – La photocopie non autorisée est un délit.

Le langage C est un langage de type **impératif** (en opposition aux langages déclaratifs, fonctionnels, flots de données ou orientés objets) compilé. Il est faiblement typé, à compilation séparée. Le langage C est sensible à la casse (majuscules/minuscules).
Le langage C dispose d'une bibliothèque exceptionnelle de composants logiciels, et la quasi-totalité des langages de programmation peuvent s'interfacer avec le langage C.
Le langage C est fait pour le multiprocessus, mais pas pour le multitâche, ce qui explique l'émergence de différentes normes temps réel dont POSIX (chapitre 5) est le représentant le plus répandu.
Le langage C est assez proche de la machine, comparé aux langages modernes. Cet inconvénient est cependant un atout indéniable lorsque l'on doit faire de la programmation bas niveau, comme dans la programmation des entrées/sorties spécifiques aux applications de contrôle-commande.
Le langage C est un langage de programmation normalisé. Aujourd'hui, des centaines de compilateurs C et environnements de développement associés existent, aussi bien en logiciel libre (le plus connu étant *GNU C Compiler*, *GCC*) qu'en logiciel commercial.

■ Le langage Ada

L'idée du langage Ada naît au milieu des années 70 sous l'impulsion du département de la défense américain du constat suivant : il n'y avait pas de langage « universel » pour la programmation des systèmes embarqués. La diversité des langages utilisés sur les différents projets coûtait cher en validation, formation, maintenance, etc. Un concours a alors été lancé et parmi les quatre propositions de langages finalistes (représentés anonymement par 4 couleurs), le langage *green* a été choisi. Il porte le nom d'Ada en mémoire du premier programmeur au monde : Augusta Ada Byron.
Ada est un langage impératif à typage fort, compilé, modulaire, à compilation séparée. Il permet le traitement des exceptions, et la généricité. Tout a été mis en œuvre pour qu'il soit le plus sûr possible. De plus, il est nativement multitâche : en langage Ada, une tâche est un type que l'on peut instancier. Ada est insensible à la casse (majuscules/minuscules).
Sa première version a été normalisée en 1983 : elle présentait des lacunes notamment au niveau des facilités de communication entre les tâches (seul le rendez-vous avec données existait). La seconde version, Ada 95, permet la programmation objet et surtout toute forme de communication asynchrone et synchronisation grâce à l'introduction de moniteurs (objets protégés).
Ada est un langage normalisé. Plusieurs compilateurs Ada existent sur la plupart des plateformes. Le plus connu des compilateurs libres est *GNAT*.

■ Le langage LabVIEW

Le langage LabVIEW est un des rares représentants des langages flots de données. C'est un langage graphique, typé, modulaire, et en tant que langage flots de données, il est naturellement parallèle.
Contrairement à C et Ada, LabVIEW est un environnement de développement propriétaire, développé par la société *National Instruments*. LabVIEW se base sur le langage G (il semble que cela soit la seule implémentation de ce langage).

LabVIEW est un langage relativement récent, puisqu'il a fait son apparition sur *Macintosh* au milieu des années 80. Initialement, il était dédié à la programmation d'instruments virtuels utilisant des cartes d'acquisition de la société *National Instruments*. L'exemple typique d'un instrument virtuel, souvent repris pour présenter l'intérêt initial de LabVIEW, est un oscilloscope logiciel se basant sur une carte d'acquisition multifonctions : l'oscilloscope est composé d'un certain nombre de fonctionnalités internes, et d'une interface utilisateur. De même, tout programme ou sous-programme LabVIEW est représenté par une interface graphique, nommée **face avant** et une description du fonctionnement interne sous forme de flots de données, nommée **diagramme**.
Depuis, le langage LabVIEW a évolué et s'est élargi au fil des versions successives, jusqu'à devenir un langage complet de programmation, plaçant LabVIEW parmi les langages les plus agréables à utiliser pour les applications de contrôle-commande.

6.1.2 Éléments de programmation comparés

■ Types et littéraux

□ Langage C

Le tableau 6.1 présente les types de base du langage C.
Notons en particulier le fait que le type *int*, très employé, est souvent représenté par la taille d'un registre de calcul. De nombreuses personnes assument de plus qu'une variable de type *int* a la taille suffisante pour contenir une adresse (un pointeur).
Le fait que différents types aient une longueur dépendante de l'architecture sous-jacente proscrit l'utilisation de valeurs constantes lorsque l'on manipule des tailles de données. Un opérateur, nommé *sizeof*, est toujours utilisé lorsque l'on doit connaître la taille en octets d'une variable, d'une constante ou bien d'un type. Ainsi, sur une architecture 32 bits classique, *sizeof(int)* vaut 4.
Notons l'absence du type *booléen* : en langage C, les tests peuvent s'effectuer sur n'importe quel type de valeur : si la valeur est nulle, le test est faux, si la valeur est non nulle, le test est vrai.
Toute variable doit être définie avant son utilisation et avant toute instruction du même bloc (un bloc est défini entre accolades et correspond à un groupe de définitions et instructions).

```
int i; /* Déclaration d'un entier i non initialisé */
float d=1.51E+2f; /* Déclaration d'un flottant double précision d
initialisé à 151 */
unsigned a,b; /* Déclaration de deux entiers non signés */
char c='a' + 1 ; /* Déclaration d'un caractère valant 'b' (code ASCII
suivant celui de 'a') */
```

Un **type enregistrement** est construit à l'aide du constructeur *struct* :

```
struct Complexe { /* Un complexe est composé de deux flottants */
      float reel;
      float imaginaire;
};
struct Complexe C; /* Déclaration d'une variable de type struct
Complexe */
C.reel = 3.5; /* Accès au champ "reel" de C */
```

© Dunod – La photocopie non autorisée est un délit.

Tableau 6.1 – Types de base en langage C.

Type	Description	Taille (en bits)	Domaine	Exemples de littéraux
int	entier	généralement 16 ou 32	$-2^{n-1}..2^{n-1}-1$ avec n taille des registres de calcul	décimal : 152, -15 octal : 0230, -017 hexadécimal : 0x98, -0xF
unsigned	entier non signé	généralement 16 ou 32	$0..2^{n}-1$	décimal : 152U octal : 0230U hexadécimal : 0x98U
char	caractère	8	-128..127	caractère : ’a’ ou code octal \141, ou caractère non imprimable ’\n’,’\t’
unsigned char	caractère non signé	8	0..255	caractère : ’a’ ou code octal \141, ou caractère non imprimable ’\n’,’\t’
short	entier court	16	$-2^{16}..2^{16}-1$	décimal : 152, -15 octal : 0230, -017 hexadécimal : 0x98, -0xF
unsigned short	entier court non signé	16	$0..2^{16}-1$	décimal : 152UL octal : 0230ul hexadécimal : 0x98uL
long	entier long	32 (parfois 64)	$-2^{31}..2^{31}-1$	décimal : 152L, -15L octal : 0230L, -017L hexadécimal : 0x98L, -0xFL
unsigned long	entier long non signé	32 (parfois 64)	$0..2^{32}-1$	décimal : 152uL octal : 0230ul hexadécimal : 0x98UL
long long	entier très long	64	$-2^{63}..2^{63}-1$	décimal:152LL,-15LL octal :0230LL,-017LL hexadécimal :0x98LL, -0xFLL
unsigned long long	entier très long non signé	64	$0..2^{64}-1$	décimal : 152ULL octal : 0230ull hexadécimal : 0x98ULL
float	flottant simple précision	généralement 32	voir norme IEEE 754	152F, -15f 1.52E+2f, -1.5E+1F
double	flottant double précision	32 ou 64	voir norme IEEE 754	152.0, -15.0 1.52E+2, -1.5E+1
long double	long flottant double précision	64, 80 ou 128	voir norme IEEE 754	152.0L, -15.0L 1.52E+2L, -1.5E+1L

Noter l'emploi obligatoire du mot clé *struct* lors de l'utilisation du type. Pour éviter cette lourdeur syntaxique, on peut utiliser le mot clé *typedef* qui permet de nommer un type. Dans l'exemple ci-après, le même type enregistrement est déclaré de façon anonyme, mais défini par le nom *Complexe* :

```
typedef struct {
      float reel;
      float imaginaire;
} Complexe;
Complexe C; /* Déclaration d'une variable de type Complexe */
Complexe C2={1,3.5}; /* C2 vaut 1+3.5i */
```

Un **type énuméré**, rarement utilisé en langage C, se définit avec le mot clé *enum* :

```
enum couleurs {rouge, vert, bleu};
/* En fait, rouge vaut 0, vert vaut 1 et bleu vaut 2 */
enum couleurs couleur=vert; /* définition de la variable couleur,
valant vert */
```

Afin d'éviter la lourdeur de répétition du mot clé *enum*, on utilise souvent le nommage de type, comme pour les types enregistrement :

```
typedef enum {rouge, vert, bleu} couleurs;
/* Ici, enum {rouge, vert, bleu} est nommé couleurs */
couleurs couleur=vert; /* définition de la variable couleur, valant
vert */
```

Le constructeur de type `union` permet de définir une superposition (union) de plusieurs types.

```
typedef union {
      char c;
      short s;
} mon_union; /* les champs c et s sont superposés */
/* la taille de l'union est max(sizeof(char),sizeof(short))=2 octets
*/
mon_union m;
m.s = 0x8054; /* m.s vaut 0x8054, du même coup, m.c vaut 0x54 */
m.c = 0; /* m.c vaut 0, du même coup m.s vaut 0x8000 */
```

Rappelons que C a un typage faible. Ainsi, l'exemple suivant passe parfaitement sur la plupart des compilateurs.

```
m.c = 'a'+3+vert; /* en introduction sur C, nous parlons de typage
faible : on affecte à un caractère la somme d'un caractère, d'un
entier et d'une couleur, soit le code ASCII de 'e' */
```

☐ Langage Ada

Le tableau 6.2 présente les types de base du langage Ada.
À l'instar du langage C, la norme Ada n'impose pas de taille pour les types numériques. Généralement, c'est la taille d'un registre de calcul.
La norme prévoit, mais n'impose pas, la présence de types numériques courts, longs, ou très longs : ainsi, *Long_Integer* correspond à un entier long, *Short_Natural* correspond à un entier non signé court, *Long_Long_Float* correspond à un flottant très long, etc.

© Dunod – La photocopie non autorisée est un délit.

Tableau 6.2 – Types de base en langage Ada.

Type	Description	Taille (en bits)	Domaine	Exemples de littéraux
integer	entier	généralement 16 ou 32	$-2^{n-1}..2^{n-1}-1$ avec n taille des registres de calcul	décimal : 152, -15 octal : 8#230#, -8#17# hexadécimal : 16#98#, -16#F#
natural	entier non signé	généralement 16 ou 32	$0..2^{n}-1$	comme integer
positive	entier positif	généralement 16 ou 32	$1..2^{n}$	comme integer
character	caractère	8	*null*..'ÿ' correspondant à 0..255	'a', 'b', cr, lf…
boolean	booléen	généralement 8	true, false	true, false
float	flottant simple précision	généralement 32	voir norme IEEE 754	1.52E+2, -1.5E+1
delta	nombre point fixe	dépend de la définition	dépend de la définition	1.52E+2, -1.5E+1

Le fait que différents types aient une longueur dépendante de l'architecture sous-jacente proscrit l'utilisation de valeurs constantes lorsque l'on manipule des tailles de données. Le langage Ada définit des **attributs** permettant d'obtenir diverses informations notamment sur les types et sur les variables. Ainsi, par exemple, l'expression *Integer'Size* donne la taille des entiers, *Integer'First* donne le plus petit entier et *Integer'Last* le plus grand.
Notons que Ada propose un constructeur de types point fixe (*delta*).
Toute variable doit être définie avant son utilisation.

```
i: integer; -- Déclaration d'un entier i non initialisé
d:float :=1.51E+2; -- Déclaration d'un flottant initialisé à 151
a, b : natural; -- Déclaration de deux entiers non signés
c: character := character'succ('a');
-- Déclaration d'un caractère valant 'b' (caractère successeur de 'a')
```

Un **type enregistrement** est construit à l'aide du constructeur de types *record* :

```
type Complexe is record
  -- Un complexe est composé de deux entiers
reel : float;
imaginaire : float;
end record;
C : Complexe ; -- Déclaration d'une variable de type Complexe
```

Noter l'emploi obligatoire du mot clé *type* lors de la déclaration d'un nouveau type.
Un **type énuméré** se définit de la façon suivante :

```
type Couleur is (Trefle, Carreau, Coeur, Pique);
Coul : Couleur := Carreau;
```

Comme en langage C, une correspondance implicite est effectuée entre valeur de type énuméré et entier : *Trefle* correspond à 0, *Carreau* à 1, etc. Cependant, contrairement à C, Ada conserve une distinction réelle entre type énuméré et type entier, et l'expression suivante est refusée par un compilateur Ada :

```
Coul := 1; -- Refusé par le compilateur
```

Cependant, comme pour tout type discret, l'utilisation d'attributs permet d'opérer une conversion avec les entiers ou bien les chaînes de caractères :

```
Coul := Couleur'Val(1); -- Couleur'Val(1) vaut Carreau
Coul := Couleur'Value("Trefle"); -- Couleur'Value("Trefle") vaut
Trefle
I := Couleur'Pos(Coul); -- I vaut 0, position de Trefle
```

Il n'y a pas en langage Ada de constructeur de type union.
Le langage Ada propose des façons très flexibles de créer des types en contrôlant finement leur représentation binaire. La façon de procéder est présentée au paragraphe 6.1.2, p. 282.

☐ Langage LabVIEW

LabVIEW étant un langage graphique, c'est de façon graphique que l'on choisit le type d'une variable. Dans ce langage, tout est flot. L'origine d'un flot a un type, qui définit le type d'un flot. L'origine d'un flot peut être une « variable d'entrée » nommée **commande**, et la fin d'un flot, une « variable de sortie » du flot nommée **indicateur**.
Chaque programme, ou sous-programme LabVIEW s'appelle un **instrument virtuel** (*virtual instrument* ou ***vi***). La philosophie est de définir un instrument par son interface graphique, ou face avant, et son programme ou diagramme. La figure 6.1 montre un *vi* prenant en paramètre un entier *a*, lui appliquant un calcul $(a + 1)/2$ afin d'afficher le résultat *b*. *a* en tant que paramètre d'entrée s'appelle une **commande** (pour commande utilisateur, modifiable via l'interface graphique) et *b* en tant que paramètre de sortie visible par l'utilisateur, s'appelle un **indicateur**. Au début du programme, les constantes et les commandes produisent une valeur sur chacun des flots (fils) auxquels ils sont connectés. Dès qu'une valeur est disponible sur chacun des flots en entrée, le *vi* devient exécutable. Il est alors exécuté et produit ses valeurs en sortie, qui à leur tour rendent d'autre *vi* exécutables, etc. Dès qu'un flot est disponible en entrée d'un indicateur, la valeur est affichée sur l'élément correspondant de la face avant.
La figure 6.2 montre qu'un numérique, s'il est entier, peut être signé ou non (préfixe I pour signé, U pour non signé), sur 8, 16, ou 32 bits. Les nombres entiers peuvent donc être de type I8, I16, I32, U8, U16, ou U32. Les nombres réels, peuvent être représentés suivant la norme IEEE 754 par un flottant de 32, 64 ou 80 à 128 bits selon la plateforme.
Les littéraux numériques se voient graphiquement imposer un type, et peuvent être représentés en décimal, octal, hexadécimal ou binaire, etc. La couleur des flots des éléments graphiques a une sémantique. Ainsi, graphiquement, les flots de données

© Dunod – La photocopie non autorisée est un délit.

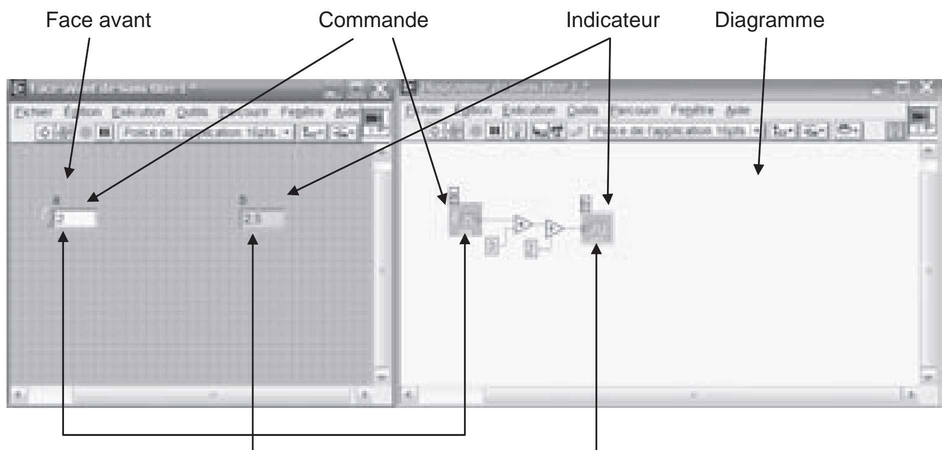


Figure 6.1 – Un programme simple en langage LabVIEW.

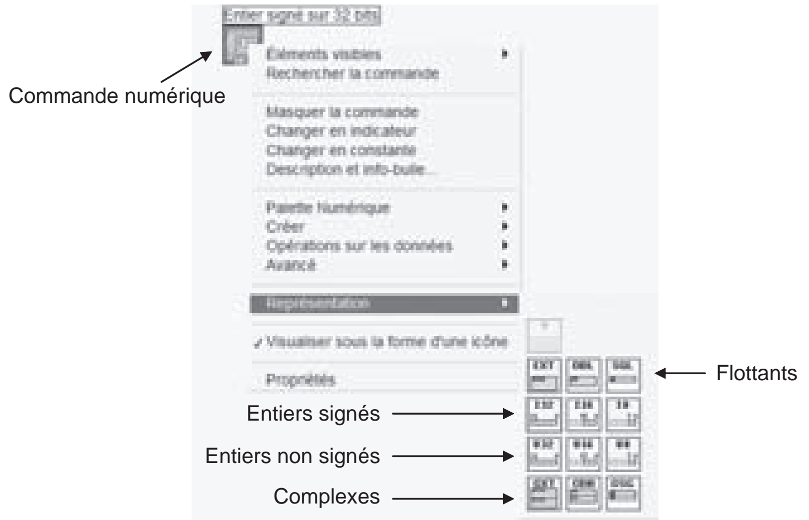


Figure 6.2 – Types numériques en langage LabVIEW.

d'entiers (signés ou non, quelle que soit leur taille) sont représentés en bleu. Les flots de données de réels sont représentés en orange.
Les booléens, normalement implémentés sur 8 bits, sont représentés graphiquement en vert. Les chaînes de caractères sont distinguées des tableaux en langage LabVIEW. Représentées en rose, celles-ci sont des types de base du langage.
La figure 6.3 montre une commande de type **enregistrement** (***cluster*** en langage LabVIEW), et la façon dont on y accède.

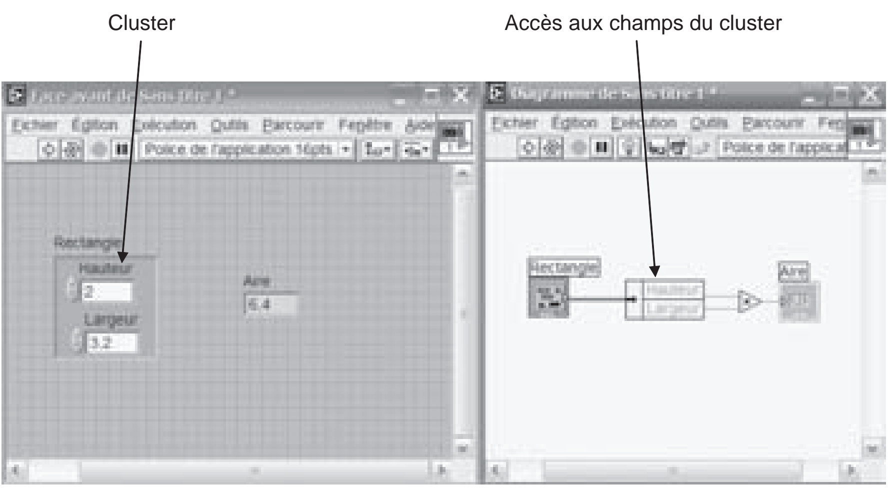


Figure 6.3 – Type enregistrement en langage LabVIEW.

Les flots de données de type *cluster* peuvent être de deux couleurs : marron ou rose. Ils sont marron lorsque tous les champs sont numériques, il est alors possible de réaliser des opérations arithmétiques champs à champs (sur la figure 6.3, on peut additionner deux *clusters* rectangles, et obtenir un *cluster* rectangle de hauteur la somme des hauteurs, et de largeur la somme des largeurs). Lorsqu'au moins un des champs est non numérique, le fil est de couleur rose, et n'autorise pas d'effectuer

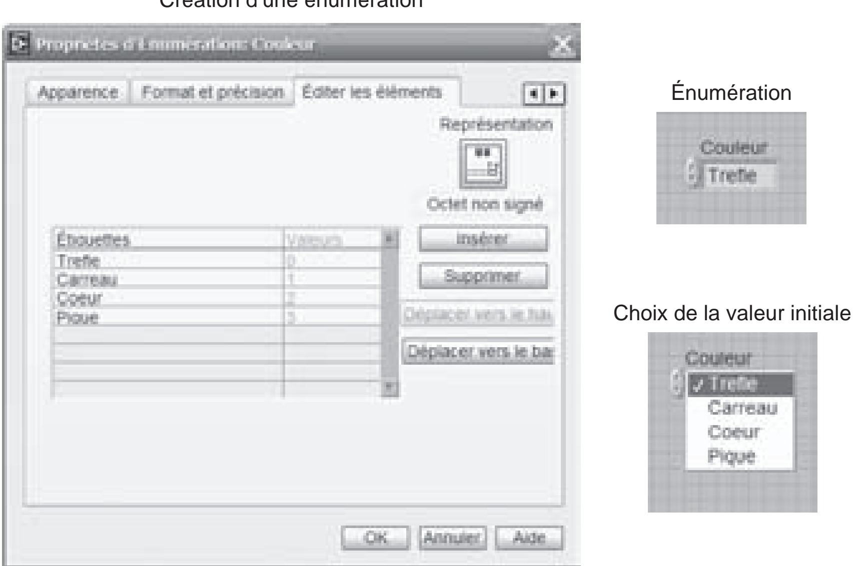


Figure 6.4 – Énumération en langage LabVIEW.

© Dunod – La photocopie non autorisée est un délit.

des opérations arithmétiques directement sur les *clusters*. La figure 6.4 montre une **énumération**.

Comme on le voit sur la figure 6.5, le typage est faible par défaut, comme en langage C, cependant les coercions (changements de types) implicites sont visibles sur le diagramme (point gris).

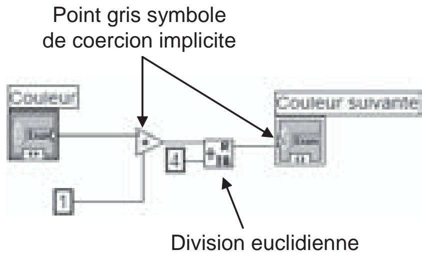


Figure 6.5 – Coercion implicite en langage LabVIEW.

■ Pointeurs, tableaux et mémoire dynamique

□ Langage C

Le langage C fait très peu de distinction entre pointeur et tableau. Un tableau est une zone mémoire contiguë permettant de stocker un certain nombre de valeurs du même type.

Un **tableau** est construit de la façon suivante :

```
int t1[3]; /* t1 est un tableau de 3 entiers non initialisés */
long T[5] = {2512,-15,42,56.32}; /* T est un tableau de 5 entiers
longs initialisé */
Complexe c, tc[5]; /* c est un complexe, tc est un tableau de 5
complexes */
```

Un tableau de taille n s'indexe de 0 à $n-1$.

```
t1[0]=3; /* La 1re case de t1 reçoit 3 */
tc[4]=C; /* La dernière case de tc reçoit la valeur de C */
C=tc[0]; /* C reçoit la valeur de la première case de tc */
```

Ainsi, la figure 6.6 présente l'organisation en mémoire du tableau *T* en supposant que l'adresse de début du tableau soit 0x22FF48. La valeur de *T* est l'adresse de *T[0]*, et l'adresse de *T[i]* vaut *T+i*sizeof(long)*.

La figure 6.6 montre que C ne conserve aucune information sur la taille du tableau. Il est donc tout à fait possible de déborder (*i.e.* accéder à une case non existante) du tableau sans s'en apercevoir pendant le déroulement d'un programme.

Il est très important de savoir qu'un tableau en lui-même n'est qu'une adresse en langage C, ainsi l'affectation de tableau nécessite une recopie mémoire :

```
long T[5] = {2512,-15,42,56.32};
long T2[5];
T2 = T ; /* T2 contient maintenant l'adresse de T (figure 6.7)*/
```

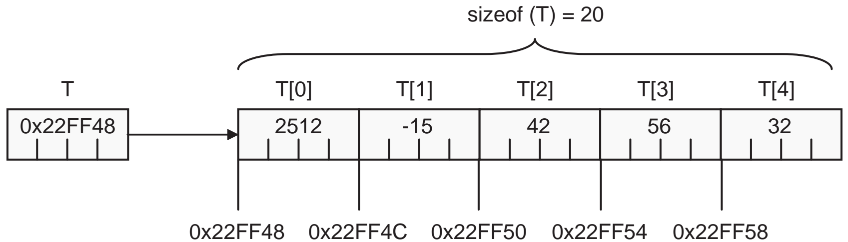


Figure 6.6 – Représentation mémoire d'un tableau en langage C.

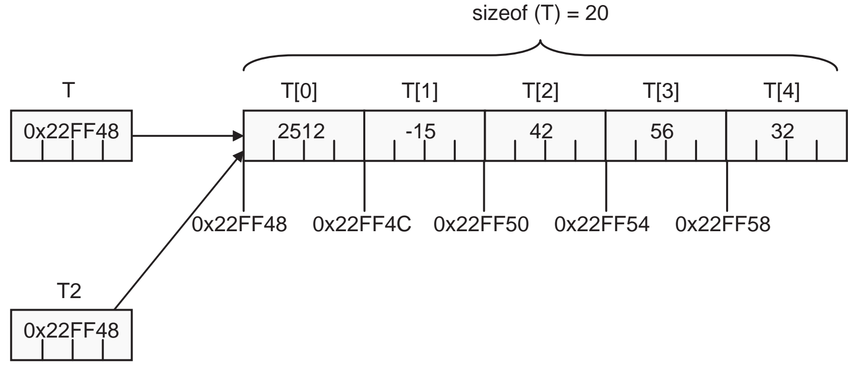


Figure 6.7 – L'affectation de tableau n'est qu'une affectation d'adresse en langage C.

La recopie de tableau nécessite une recopie mémoire :

```
long T[5] = {2512,-15,42,56.32};
long T2[5];
memcpy(T2, T, sizeof(T)) ; /* T2 contient maintenant une copie de T*/
```

De la même façon, la comparaison de tableau consiste à comparer l'adresse de base des tableaux, mais pas leur valeur. Afin de comparer deux tableaux de même taille, il faut effectuer une comparaison mémoire :

```
int identiques ;
identiques = memcmp(T2, T, sizeof(T)) ; /* vrai (différent de 0) si
les contenus de T et T2 sont identiques, octet par octet */
```

Les tableaux multidimensionnels sont déclarés de la façon suivante :

```
long T3[2][5]; /* tableau de 2*5 entiers longs */
long T4[][3] = {0,1,2,3,4,5}; /* la taille de la première dimension
est donnée implicitement */
long i=T4[0][2] ; /* i vaut 2 */
```

Un **pointeur** se définit en langage C à l'aide du caractère * :

```
long i, *pt; /* pt est un pointeur vers un entier long */
pt = &i; /* pt contient l'adresse de i */
*pt = 3; /* la valeur pointée par pt (c'est-à-dire i) vaut 3 */
```

© Dunod – La photocopie non autorisée est un délit.

L'opérateur « & » utilisé avant une variable est un opérateur de **référencement** (&*i* donne l'adresse de *i*). L'opérateur * placé avant une expression (ici **pt*) est un opérateur de **déréférencement**, c'est-à-dire qu'il permet d'accéder à l'adresse donnée par l'expression (ici, l'adresse de *i*).

```
long **pt2 ; /* ici pt2 est un pointeur de pointeur d'entier long */
pt2 = &pt; /* pt2 contient l'adresse de pt */
*(*pt2) = 4 ; /* i vaut maintenant 4 */
```

Pointeurs et tableaux sont extrêmement proches, puisqu'ils contiennent tous les deux une adresse. La différence fondamentale entre les deux est qu'un tableau voit sa mémoire allouée au moment de sa déclaration, ainsi, 20 octets sont alloués au tableau, dont l'adresse est le début de ces 20 octets lors de la déclaration de :

```
long T[5];
```

La taille d'un tableau est donc fixe. Les pointeurs se substituent aux tableaux lorsque l'on doit utiliser un tableau de taille **dynamique**. Pour cela, on alloue à la main l'espace nécessaire grâce aux fonctions *malloc* ou *calloc* :

```
long *pt;
pt = (long*)malloc(sizeof(long)); /* pt pointe vers 4 octets alloués*/
/* du code */
free(pt); /* libération de l'espace mémoire alloué à pt */
pt = (long*)calloc(10,sizeof(long)); /* pt pointe vers un tableau de
10 entiers longs initialisés à 0 */
pt[1]=3; /* la seconde case du tableau reçoit 3 */
/* du code */
free(pt); /* libération de l'espace mémoire alloué à pt */
```

On peut noter qu'en langage C, toute mémoire allouée doit être rendue (grâce à la fonction *free*). Notons que la différence entre *calloc* et *malloc*, mis à part le nombre de paramètres, est que *calloc* initialise chaque octet de mémoire allouée à 0.
Notons aussi qu'il est nécessaire de faire une coercion explicite de l'espace mémoire créé par *malloc* et *calloc*, de type *void ** (*void* est le type muet), en *long**.
Les **chaînes de caractères** sont des tableaux de caractères. Un caractère spécial, '\0', de code ASCII 0, signifie la fin d'une chaîne de caractères. Ainsi, une chaîne de caractères de *n* caractères occupe au moins $n + 1$ octets : les *n* caractères plus le caractère de fin de chaîne.

```
char ch[]="une chaine"; /* Déclaration d'une chaîne de caractères */
char *ch2 = "abc"; /* Pointeurs et tableaux peuvent presque
indifféremment être utilisés */
char *ch3=ch2; /* ch3 pointe sur la même chaîne que ch2 */
```

La figure 6.8 montre comment ces chaînes de caractères sont gérées en mémoire.
Les chaînes de caractères étant fondamentalement des tableaux, l'affectation d'une chaîne à une autre copie l'adresse de la chaîne, mais pas le contenu de la chaîne elle-même.

```
int taille_ch;
char ch[512], *copie_ch; /* Déclaration d'une chaîne de caractères de
512 caractères ch, et d'un pointeur */
printf("Entrez une chaîne"); /* Affichage de "Entrez une chaîne" */
scanf("%s",ch); /* lecture d'une chaîne au clavier */
```

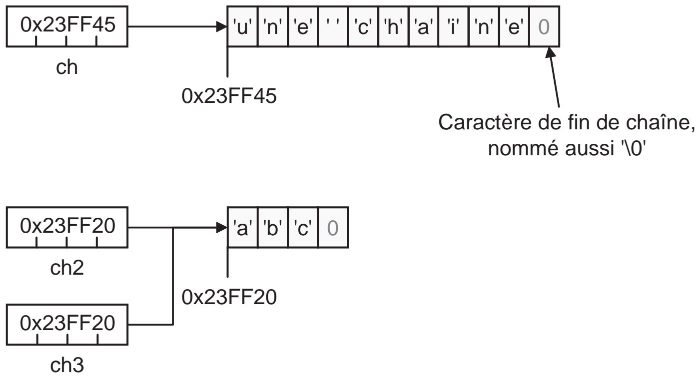


Figure 6.8 – Exemple de chaînes de caractères en langage C.

```
taille_ch = strlen(ch); /* taille_ch vaut la longueur de ch, sans
compter le caractère fin de chaîne */
copie_ch=(char *)malloc(sizeof(char)*(taille_ch+1)); /* Allocation
mémoire non initialisée (utilisation de malloc) de taille juste
suffisante (taille de ch + le caractère de fin de chaîne) pour copier
la chaîne lue */
strcpy(copie_ch,ch); /* recopie de la chaîne ch dans copie_ch */
/* code ...*/
free(copie_ch); /* libération de la mémoire allouée à copie_ch */
```

Les fonctions spécifiques aux chaînes de caractères parcourent la ou les chaînes jusqu'au caractère de fin de chaîne. Ainsi, *strcpy* copie les caractères un à un en incluant le caractère de fin de chaîne et termine. De même, la comparaison de chaînes, effectuée par *strcmp*, s'arrête au premier caractère de fin de chaîne rencontré.

□ Langage Ada

Ada gère de façon fondamentalement différente pointeurs et tableaux, afin de permettre un suivi sûr du déroulement des programmes.
Un type tableau se définit à l'aide du mot clé *array* :

```
type Tab is array (1..5) of Integer;
-- type tableau indicé de 1 à 5
T1: Tab := (1, 3, 5, 7, 9);
-- Déclaration de tableau initialisé
T2 : Tab := (others=>4);
-- Déclaration de tableau initialisé par agrégat
-- toutes les valeurs valent ainsi 4
```

Contrairement à C, l'affectation de tableaux effectue une recopie, et la comparaison compare effectivement le contenu des tableaux. L'affectation de deux tableaux de taille différente lève une exception (§ 6.1.2, p. 276).

```
T1(1) := 3; -- la 1re case de T1 vaut 3
T2 := T1;   -- T1 est recopié dans T2
```

Un tableau de taille **dynamique** est déclaré de la façon suivante :

© Dunod – La photocopie non autorisée est un délit.

```
type Tab is array (natural range <>) of Integer;
-- type tableau d'entiers de taille dynamique
T1: Tab(1..5) := (1, 3, 5, 7, 9);
-- Déclaration de tableau de taille 5 initialisé indicé de 1 à 5
T2 : Tab(0..3) := (others=>4);
-- Déclaration de tableau de taille 4 indicé de 0 à 3
```

Notons qu'en langage Ada, les indices de tableau sont dépendants de la déclaration du type. Notons aussi que tout type discret peut servir à indicer un tableau. Ainsi, le code suivant est correct :

```
type Bus_Externe is (RS232, Parallele, USB, FireWire, SCSI);
type Tableau_Debit is array (RS232..SCSI) of Integer;
-- Type tableau indicé par un type énuméré
Debits : Tableau_Debit;
```

On peut ainsi accéder par exemple à *Debits(RS232)*.
Les **chaînes de caractères** sont de type *String*, dont la définition est :

```
type String is array (Positive range <>) of Character;
```

Les **pointeurs** sont définis par le mot clé *access* :

```
type Pt_Entier is access Integer; -- Type pointeur sur entier
P : Pt_Entier; -- Pointeur initialisé à null
```

L'allocation dynamique se fait à l'aide de l'opérateur *new*, le déréférencement se fait en adjoignant *.all* à un pointeur :

```
P:= new Integer; -- allocation dynamique
P.all := 3; -- référencement du pointeur
```

Notons qu'il est possible d'affecter une valeur au pointeur au moment de l'allocation :

```
P:= new Integer'(3); -- allocation dynamique et initialisation
```

Ada, qui est un langage sûr, ne permet pas de référencer une variable quelconque. Il est nécessaire pour cela de dire explicitement que la variable peut être référencée par pointeur grâce au mot clé *aliased*, et de permettre au type pointeur le référencement (mot clé *access all*) :

```
type Pt_Entier is access all Integer;
I : aliased Integer;
P : Pt_Entier;
```

Ainsi, *P* peut référencer *I* :

```
P := I'access; -- P pointe sur I
```

Notons que contrairement à C, Ada est muni d'un **ramasse miettes** (***garbage collector***) chargé de récupérer la mémoire dynamique n'étant plus utilisée. Il n'est donc pas nécessaire, sauf cas exceptionnel, de libérer explicitement la mémoire allouée.

☐ Langage LabVIEW

LabVIEW permet de créer des tableaux de façon graphique (figure 6.9), mais n'a pas la notion explicite de pointeur. En effet, la notion de pointeur n'est pas compatible avec la philosophie flot de donnée. Les pointeurs sont cependant accessibles

pour certains objets graphiques via des références, ou des variables locales de façon à simplifier certains types de programmation.

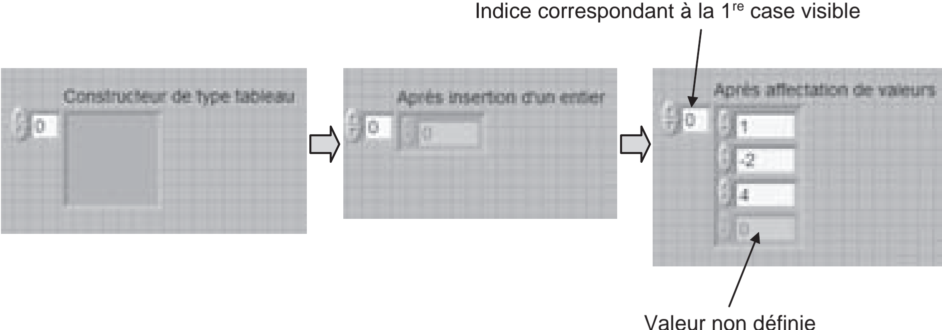


Figure 6.9 – Création d'une commande de type tableau en langage LabVIEW.

LabVIEW propose divers outils fonctionnant sur les tableaux (indexation, calcul de la taille, construction, concaténation…). Les tableaux sont indicés de 0 à $n - 1$ comme en langage C. Notons que les flots de données tableaux sont repérés par un trait épais conservant la couleur du type contenu dans le tableau. Il est ainsi possible, comme sur les clusters, d'effectuer directement des opérations arithmétiques sur des tableaux entiers.

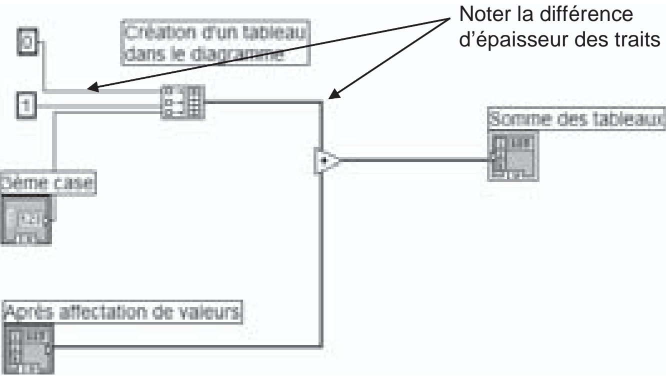


Figure 6.10 – Manipulation de tableaux en langage LabVIEW.

L'allocation mémoire est réalisée de façon transparente, et LabVIEW vérifie pendant les opérations sur les tableaux qu'il n'y a pas débordement. La lecture d'une case allant au-delà de la fin d'un tableau donne la valeur par défaut du type (0 pour les types numériques).

Les tableaux multidimensionnels sont créés de façon similaire.

© Dunod – La photocopie non autorisée est un délit.

■ Structures de contrôle

□ Langages C et Ada

La différence syntaxique entre structures de contrôle C et Ada réside surtout dans le fait qu'en langage C, une structure de contrôle contient une seule instruction ou bien un bloc (entre accolades) d'instructions, alors qu'en langage Ada, un bloc contient des instructions et se termine au(x) mot(s) clé(s) terminateurs de blocs (par exemple *if*, *end id*).

Tableau 6.3 – Structures de contrôle en langage C et Ada.

C	Ada

`int T[10];`
`int i,j;`
 	
`for (i=0;i<10;i++) {`
`      T[i]=i;`
`}`
`j=0;`
`while (j<10) {`
`      T[j]=j;`
`      j++;`
`}`
`j=0;`
`do {`
`      T[j]=j;`
`      j++;`
`} while (j<10);`
 	
`if (T[2]>=2) {`
`      T[3]=1;`
`} else if (T[2]<1) {`
`      T[3]=-1;`
`} else {`
`      T[3]=0;`
`}`
 	
`switch (T[2]) {`
`case 1:`
`      T[3]=1;`
`      T[4]=5;`
`      break;`
`case 2:`
`      T[3]=3;`
`      T[4]=6;`
`      break;`
`default:`
`      T[2]=4;`
`}`
	`T : array(1..10) of integer;`
	`j:integer;`
	
	`for i in 1..10 loop`
	`      T(i)=i-1;`
	`end loop;`
	`j:=1;`
	`while j<=10 loop`
	`      T(j):=j-1;`
	`      j:=j+1;`
	`end loop;`
	`j:=1;`
	`loop`
	`      T(j):=j-1;`
	`      j:=j+1;`
	`      exit when j>10;`
	`end loop;`
	`if T(3)>=2 then`
	`      T(4)=1;`
	`elsif T(3)<1 then`
	`      T(4)=-1;`
	`else`
	`      T(4)=0;`
	`end if;`
	
	`case T(2) is`
	`when 1=>`
	`      T(3):=1;`
	`      T(4):=5;`
	
	`when 2=>`
	`      T(3):=3;`
	`      T(4):=6;`
	
	`when others=>`
	`      T(2):=4;`
	`end case;`

Nous pouvons noter que les indices de boucle ne sont pas déclarés en langage Ada, et que les indices de tableaux ne commencent pas nécessairement à 0. Notons aussi l'emploi obligatoire de `break` à la fin de chaque alternative d'une structure de choix en langage C.
Notons aussi qu'en langage Ada, il est plus élégant de parcourir un tableau en utilisant les attributs de tableau : ainsi, l'expression T'range vaut T'first..T'last, c'est-à-dire 1..10 pour l'exemple donné sur le tableau 6.3. Donc, de façon plus élégante, la boucle « pour » du tableau 6.3 peut s'écrire :

```
for i in T'range loop
      T(i)=i-1;
end loop;
```

Notons aussi une subtilité lors de l'évaluation des booléens, ayant un impact sur les structures de choix : en langage C l'évaluation des booléens est une **évaluation paresseuse**, ainsi le code C suivant est correct :

```
char ch[512]; /* Ch est indicé de 0 à 511 */
scanf("%s",ch); /* Lecture au clavier de ch d'au plus 10 caractères */
int i=0;
/* Recherche du caractère '.' */
while (i<sizeof(ch) && ch[i]!='.' && ch[i]!=0) {
/* Tant que i est dans la chaîne et ch[i] différent de '.' et pas fin
de chaîne */
      i++;
}
if (i<sizeof(ch) && ch[i]=='.') {
/* Si on n'a pas dépassé la fin de la chaîne et que i est l'indice d'un
caractère '.' */
      printf("Il y a un '.' au caractère %d",i+1);
} else {
      printf("Il n'y a pas de point");
}
```

En effet, les expressions booléennes étant évaluées paresseusement, si la première condition est fausse, le reste n'est pas évalué. Ainsi, on n'essaie pas d'accéder au-delà de la chaîne *ch*.
Au contraire, en langage Ada, les expressions booléennes sont par défaut évaluées complètement, ainsi, le code Ada suivant est erroné, car même si *i* est au-delà de la fin de la chaîne, on accède à *Ch(i)* :

```
Ch: String(1..512); -- Ch est indicé de 1 à 512
I: Integer := 0;
Longueur: Integer;
…
Get_Line(Ch,Longueur); -- Lecture de Ch au clavier,
-- Longueur vaut la longueur de la chaîne lue
-- Recherche du caractère '.'
while (i<=Longueur and Ch(i)/='.') loop
   -- Tant que i est dans la chaîne et ch(i) différent de '.'
   i:=i+1;
end loop;
if (i<Longueur and ch(i)='.') then
-- Si on n'a pas dépassé la fin de la chaîne
-- et que i est l'indice d'un caractère '.'
   Put("Il y a un '.' au caractère "&Integer'Image(I+1));
```

© Dunod – La photocopie non autorisée est un délit.

```
   -- Integer'Image convertit un entier en chaîne de caractères
   -- & est la concaténation
else
   Put("Il n'y a pas de point");
end if;
```

Ada propose les opérateurs *and then* et *or else* permettant une évaluation paresseuse d'une expression booléenne.

☐ Langage LabVIEW

LabVIEW propose presque les mêmes structures de contrôle que C et Ada : la boucle « pour » (figures 6.11 et 6.12), la boucle « tant que » (figure 6.13), la structure conditionnelle (figure 6.14) qui ne se distingue de la structure de choix (figure 6.15) que par la condition – si c'est un booléen, c'est une structure conditionnelle, si c'est un autre type de base (entier, type énuméré, chaîne de caractère), alors c'est une structure de choix.

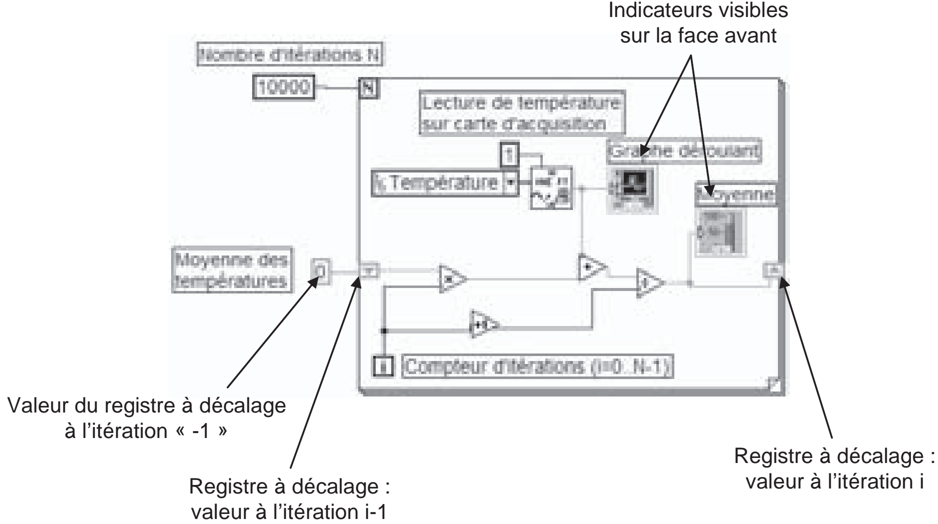


Figure 6.11 – Boucle *pour* en langage LabVIEW.

Notons sur la figure 6.11 la présence d'un registre à décalage, outil très souvent utilisé dans les boucles. En effet, d'après la philosophie du flot de données : un nœud ne peut s'exécuter que lorsque tous ses flots de données en entrée sont présents, il ne peut pas y avoir de cycle. Par conséquent, il ne serait pas possible de prendre un fil en sortie de boucle (ici, la moyenne des températures lues) et de le remettre en entrée de boucle. C'est pour cela que l'on utilise le registre à décalage.
Notons aussi qu'afin de faciliter l'utilisation de boucles « pour » sur les tableaux, LabVIEW fournit un mécanisme d'indexation/désindexation automatique de tableau. Ainsi, sur la figure 6.12, le tableau en entrée est indexé automatiquement (noter la présence de petits crochets sur le bord de la boucle) : la valeur prise dans le tableau

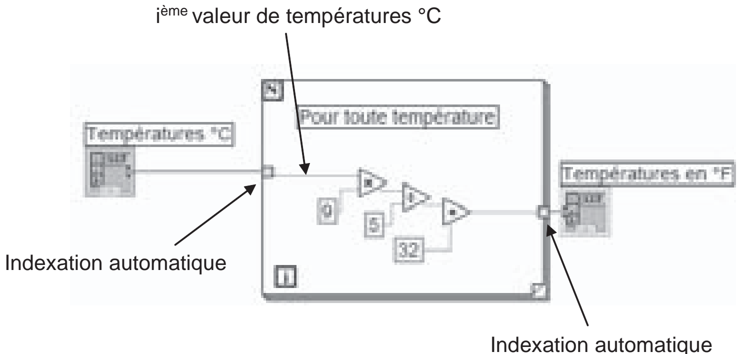


Figure 6.12 – Boucle *pour* avec indexation automatique en langage LabVIEW.

de températures à l'indice de boucle *i* est le contenu de la ième case du tableau. Dans ce cas, le programmeur n'a pas à donner le nombre d'itérations de boucle : celui-ci est implicitement la taille du tableau en entrée.
Sur la figure 6.12, un tableau est indexé automatiquement. Ce tableau, de même que tout flot sortant d'une structure de contrôle, **n'est créé qu'à la fin de la boucle**. Tout se passe comme si les valeurs étaient accumulées à l'intérieur de la boucle, et n'étaient disponibles qu'à la fin de celle-ci.

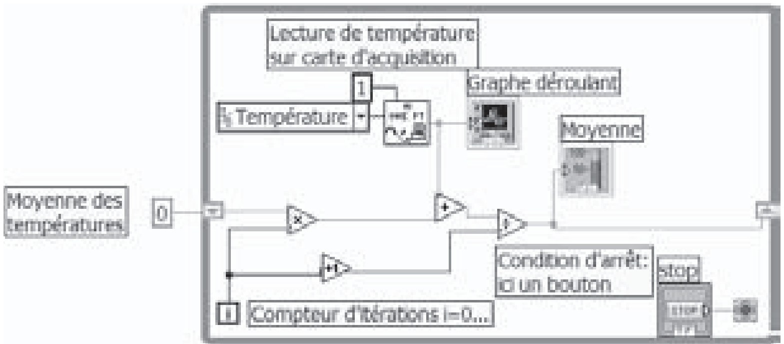


Figure 6.13 – Boucle *tant que* en langage LabVIEW.

Une boucle « tant que » est caractérisée par une condition d'arrêt, prenant un paramètre booléen : sur la figure 6.13, ce paramètre est en bas à droite, et sort de la boucle à la fin de la première itération pour laquelle le paramètre envoyé à la condition d'arrêt est vrai. En fait, la boucle « tant que » LabVIEW est une boucle « faire jusqu'à » lorsque la condition d'arrêt est affublée d'un point rouge, et une boucle « faire tant que » lorsque la condition d'arrêt ne possède pas de point rouge.
Noter sur la figure 6.14 que toute structure de contrôle est considérée comme un nœud dans LabVIEW : si un flot sort dans un cas (ici une chaîne de caractères), alors il doit sortir dans tous les cas.

© Dunod – La photocopie non autorisée est un délit.

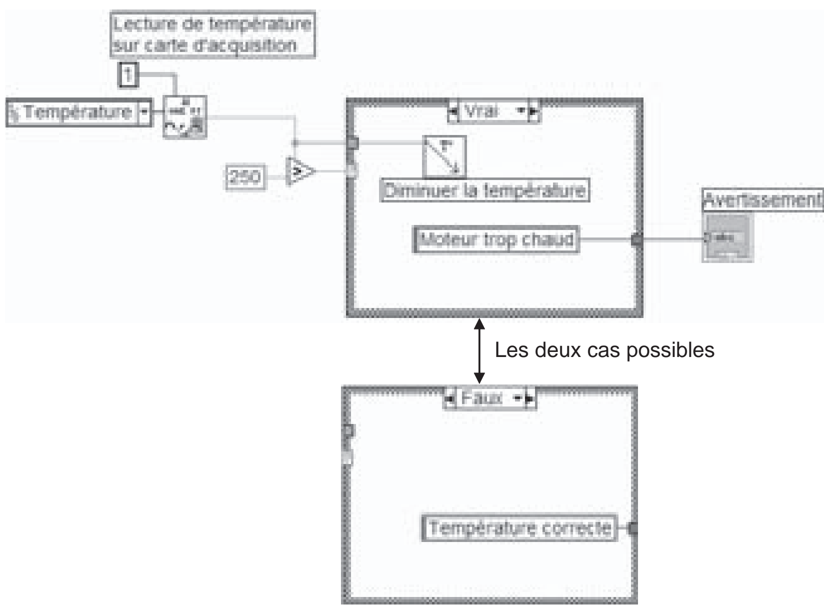


Figure 6.14 – Structure conditionnelle en langage LabVIEW.

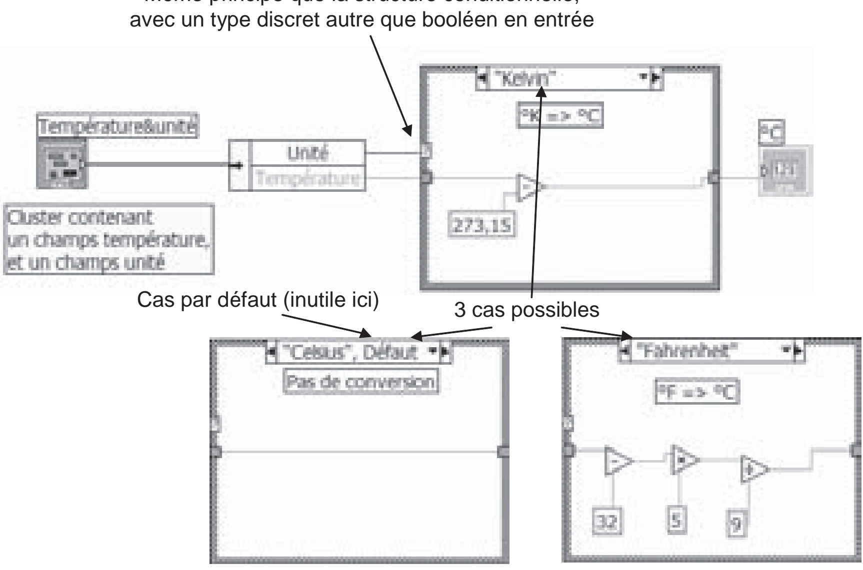


Figure 6.15 – Structure de choix en langage LabVIEW.

Notons l'existence d'une structure de contrôle inédite par rapport à C et Ada : la structure séquence. En effet, LabVIEW est naturellement parallèle car tous les nœuds prêts à s'exécuter peuvent être exécutés, dans n'importe quel ordre. Lorsque l'on souhaite forcer un ordre d'exécution, on pourra donc être amené à utiliser la structure séquence.

■ Sous-programmes

□ Langage C

En C, tout sous-programme est une fonction, prenant des paramètres en entrée uniquement et renvoyant une valeur. Le prototype d'une fonction C à deux arguments est :

```
type_de_retour nom_fonction (type1 argument1, type2 argument2) ;
```

Par exemple, la fonction calculant le plus grand commun diviseur de deux entiers a pour prototype :

```
int pgcd(int a, int b);
/*
Entrée: a,b deux entiers
Retourne: le plus grand commun diviseur de a et b
Convention: pgcd(0,b) = pgcd(a,0) = 1
*/
```

L'équivalent d'une procédure dans d'autres langages (sous-programme ne renvoyant pas de valeur) se fait en spécifiant un type de retour *void*. Ainsi, le prototype d'une fonction d'affichage d'entier peut être :

```
void afficher(int a);
/*
Entrée: a entier à afficher
Entraîne: affiche a
*/
```

Étant donné que tous les arguments sont passés par valeur (les valeurs des arguments sont recopiées directement sur la pile au moment de l'appel), afin de passer un paramètre en sortie ou en entrée sortie, de sorte à le modifier dans la fonction, on passe son adresse. Voici le prototype d'une fonction transformant un mot en son mot miroir (rappelons qu'un tableau est fondamentalement un pointeur) :

```
void miroir(char * ch);
/*
Entrée/Sortie: ch
Entraîne: ch' = miroir('ch)
*/
```

Le problème est qu'il est alors difficile de distinguer, typiquement lorsqu'on passe un tableau ou un pointeur en paramètre à une fonction, si celui-ci est un paramètre d'entrée, ou bien d'entrée/sortie. Pour distinguer cela, observons sur le prototype de la fonction de recopie de chaîne de caractères l'emploi du mot clé *const* (constant) :

```
char *strcpy(char *destination, const char *source)
/*
Entrée: source
```

© Dunod – La photocopie non autorisée est un délit.

```
Sortie: destination
Entraîne: copie de source dans destination
Nécessite: sizeof(destination)>=strlen(source)+1
Retourne: l'adresse de destination
*/
```

Les sous-programmes se déclarent « à plat », c'est-à-dire qu'en langage C, il y a un contexte global, dans lequel on déclare tous les types globaux, variables globales, et constantes globales, ainsi que tous les sous-programmes. Un sous-programme particulier, nommé ***main*** (principal), est le programme principal, c'est-à-dire le point d'entrée du programme.
Voici un exemple de sous-programme et son utilisation dans un programme principal :

```
#include <stdio.h> /* inclusion des en-têtes définissant scanf et
printf */
#include <math.h> /* inclusion des en-têtes définissant abs */
int pgcd (int A, int B) {
/*
Entrée: A,B deux entiers
Retourne: le plus grand commun diviseur de A et B
Convention: pgcd(0,B) = B, pgcd(A,0) = A
*/
    int r;
    int a=abs(A); /* Nous travaillons en valeur absolue */
    int b=abs(B);
    if (b>a) {
        return pgcd(b,a);
    }
    r = a % b; /* Reste de la division entière */
    if (r==0) {
        return b;
    } else {
        return pgcd(b,r);
    }
}
void main() {
int a,b;
printf("Donnez deux entiers:"); /* Affichage de texte */
scanf("%d %d",&a,&b); /* Lecture de a et b */
/* a et b sont passés par adresse car modifiés par scanf */
printf("%d",pgcd(a,b));/*Affichage du résultat de la fonction */
}
```

Remarque

Généralement, les types de retour de fonction sont des types scalaires. L'utilisation de tableaux comme types de retour est à proscrire, car rappelons que l'affectation de tableaux ne fait qu'une copie d'adresse et pas de contenu. Ainsi, un tableau créé à l'intérieur d'une fonction est perdu dès la fin de celle-ci. En effet, en tant que variable locale, un tableau est créé dans la pile lors de l'appel d'une fonction, et la pile est libérée au retour de celle-ci (§ 4.2.3, p. 154).

□ Langage Ada

En Ada, il existe deux types de sous-programmes : les fonctions qui renvoient une et une seule valeur, et ne prenant que des paramètres en entrée (*IN*), et les procédures, ne renvoyant pas de valeur mais pouvant prendre des paramètres en entrée ou en

entrée/sortie (*IN OUT*). Le prototype d'une fonction en langage C à deux arguments est :

```
function Nom_Fonction(Arg1:Type1 ; Arg2 :Type2) return Type_Retour;
```

Le prototype d'une procédure prenant un argument en entrée, et l'autre en entrée/sortie est :

```
procedure Nom_Procedure(Arg1:Type1 ; Arg2 : in out Type2);
```

Remarquons qu'implicitement, un argument est en entrée.
Par exemple, la fonction calculant le plus grand commun diviseur de deux entiers a pour prototype :

```
function Pgcd(a: Integer; b: Integer) return Integer;
-- Entrée: a,b deux entiers
-- Retourne: le plus grand commun diviseur de a et b
-- Convention: Pgcd(0,b) = Pgcd(a,0) = 1

Le prototype d'une procédure d'affichage d'entier peut être :

procedure Afficher(a : Integer);
-- Entrée: a entier à afficher
-- Entraîne: affiche a
```

Les sous-programmes peuvent se déclarer à l'intérieur d'autres sous-programmes dans quelques cas très rares. Contrairement au langage C, on travaille rarement en contexte global en langage Ada. Généralement, tout sous-programme, type, variable, constante est déclaré dans des **paquetages** (modules). Un fichier au moins contient une procédure seule : c'est le programme principal (qui ne s'appelle pas nécessairement *main*).
Voici un exemple de sous-programme et son utilisation dans un programme principal. Pour plus de simplicité, avant la présentation des paquetages, nous utilisons, fait rare, un sous-programme interne à une procédure :

```
with Ada.Text_Io, Ada.Integer_Text_Io;
use Ada.Text_Io, Ada.Integer_Text_Io;
-- utilisation des paquetages définissant put_line et get d'entiers
procedure Test_Pgcd is -- Procédure principale
   function Pgcd (I : Integer; J : Integer) return Integer is
      -- Entrée: I,J deux entiers
      -- Retourne: le plus grand commun diviseur de I et J
      -- Convention: pgcd(0,J) = J, pgcd(I,0) = I
      R,A,B : Integer;
   begin
      A:=abs(I); -- Travail en valeur absolue
      B:=abs(J);
      if B>A then
         return Pgcd(B,A);
      else
         R := A mod B; -- Reste de la division entière
         if R=0 then
            return B;
         else
            return Pgcd(B,R);
         end if;
      end if;
```

© Dunod – La photocopie non autorisée est un délit.

```
   end; -- Fin de la fonction Pgcd
   -- Variables de la procédure principale
   A,
   B : Integer;
begin
   Put_Line("Donnez deux entiers"); -- Affichage de texte
   Get(A);
   Get(B); -- Lecture des deux entiers au clavier
   Put(Pgcd(A,B)); -- Affichage du résultat à l'écran
end;
```

□ Langage LabVIEW

Le langage LabVIEW est inégalable pour les tests unitaires de sous-programmes car chaque programme est potentiellement un sous-programme. En effet, lorsqu'on conçoit un *vi*, les commandes sont les entrées utilisateur, et les indicateurs les sorties utilisateur. Le principe de l'encapsulation LabVIEW permet de passer les commandes d'un *vi* par un autre *vi*, qui devient ainsi *vi* appelant, et récupère les indicateurs comme paramètres de sortie.
Afin d'illustrer ce concept, considérons le *vi* donné sur la figure 6.15 qui transforme une température donnée dans une unité quelconque en degrés Celsius. Il peut être exécuté pendant la phase de développement afin de le tester. Ensuite, il est prêt à être utilisé en tant que sous-programme. Pour cela, il suffit de se placer sur sa face avant et de visualiser les connecteurs sur l'icône du *vi* (figure 6.16). Là, il ne reste plus qu'à identifier le fait que le *vi* possède un paramètre d'entrée et un paramètre de sortie pour le transformer en sous-programme.

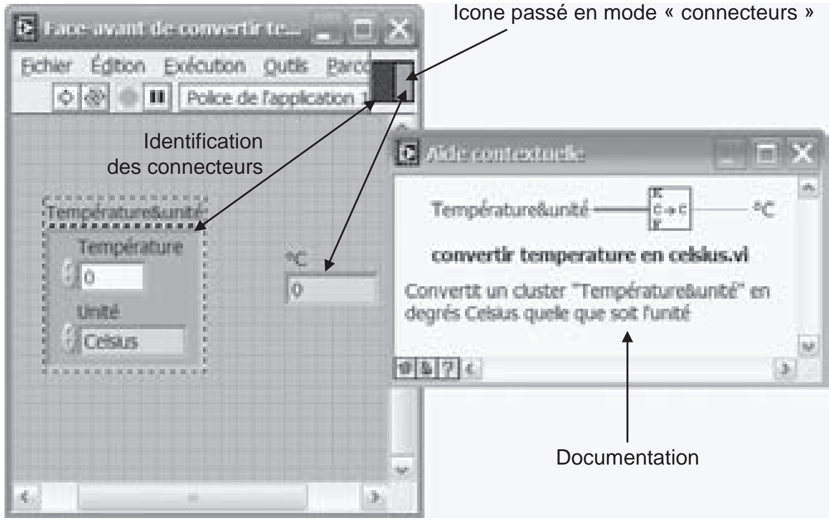


Figure 6.16 – Sous-programme en langage LabVIEW.

Afin de rendre le *vi* plus simple à reconnaître dans les *vi* qui l'utilisent, il reste à dessiner un icône personnalisée, et à le documenter (propriétés du *vi*, documentation).

Ainsi, l'aide contextuelle de LabVIEW permet d'obtenir la documentation du *vi* lorsque le pointeur de la souris est dessus. Pour l'insérer dans un autre *vi*, un glisser-déposer du fichier ou de l'icône suffit.

Notons une limitation de LabVIEW : le langage ne permet pas la récursivité (voir le calcul du plus grand commun diviseur sur la figure 6.17). Cela n'est que très exceptionnellement gênant, tout algorithme récursif pouvant s'exprimer de façon non récursive. De plus, les algorithmes récursifs mettent en jeu une utilisation très importante de la pile d'appel et doivent être proscrits dans les applications à haut niveau de sûreté.

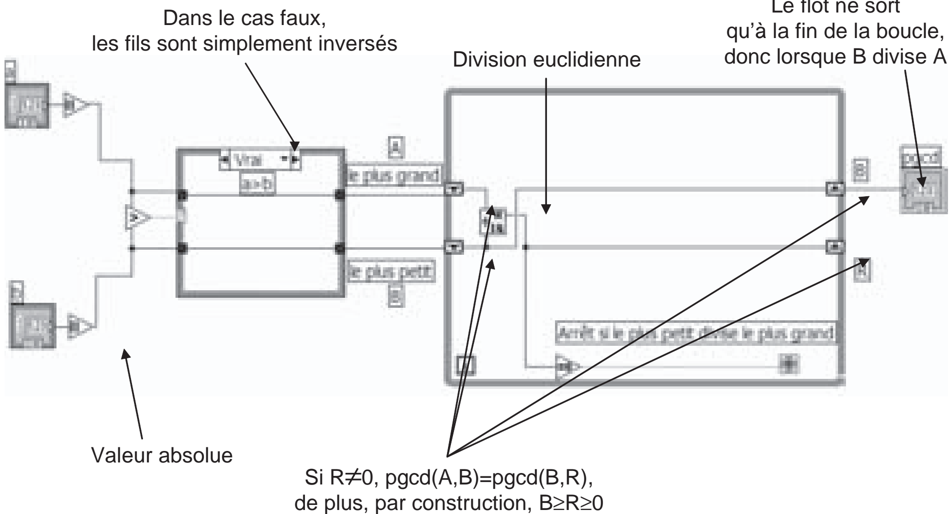


Figure 6.17 – Pas de récursivité en langage LabVIEW.

■ Modularité et visibilité

□ Langage C

La figure 6.18 montre une hiérarchie typique liée à l'élaboration d'un programme C.

Nous distinguons sur cette figure différents types d'éléments :

– le **fichier exécutable** (extension « .exe » sur une plateforme Microsoft, ou sans extension spécifique sur des plateformes Unix ou Macintosh) ;

– les **objets** (extension « .o » ou « .obj ») permettent de représenter en code intermédiaire très proche du langage machine, les sous-programmes et les variables globales ou spécifiques à un module ;

– les **modules**, entourés sur la figure 6.19, représentent un arbre de dépendance lors de la compilation. Un module définit un ensemble de sous-programmes et de variables globales ou spécifiques au module ;

© Dunod – La photocopie non autorisée est un délit.

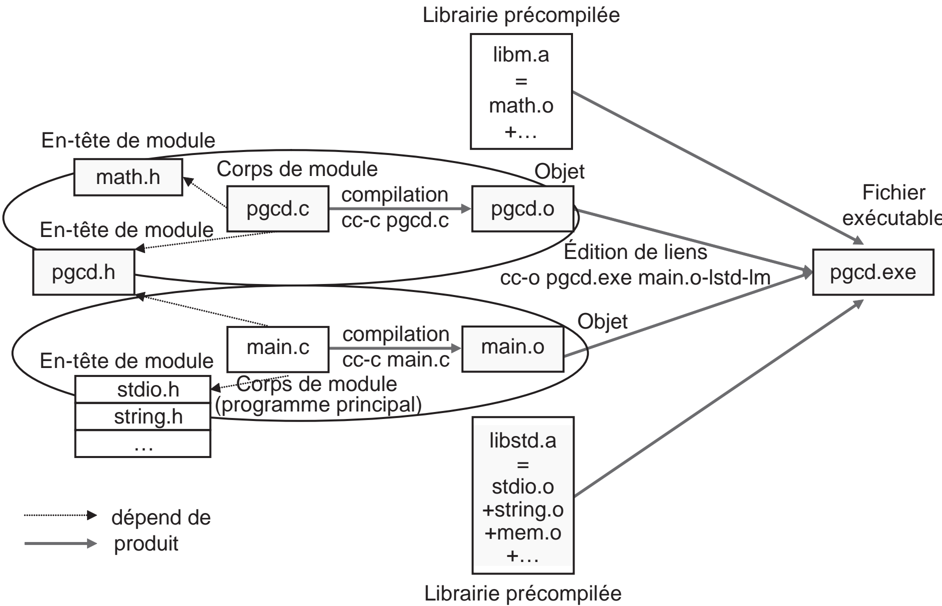


Figure 6.18 – Hiérarchie lors de l'élaboration d'un programme écrit en langage C.

– généralement, hormis le module intégrant le programme principal, les modules sont composés de deux parties : un **corps de module** (extension « .c » donnant l'implémentation du module et définissant les variables et constantes de module ou globales), et un fichier d'**en-têtes de module**, donnant la spécification de l'interface du module. Typiquement, cette spécification consiste en la définition de signatures des sous-programmes fournis par le module, ainsi que de constantes et de variables déclarées dans le fichier « .c » ;

– des librairies de modules précompilés permettent de regrouper différents objets en un seul fichier. Les fichiers en-têtes définissant les interfaces des modules restent quant à eux sous la forme de fichiers « .h ».

Un programme est donc généré à partir de la compilation des modules, indépendamment les uns des autres. Sur la figure 6.19, nous utilisons la commande « cc –c » qui compile un module en un objet. Ce type de commande est typique du monde Unix/Linux, ainsi que des compilateurs libres (GNU) que l'on peut trouver sur la plupart des plateformes.

Il reste ensuite à élaborer le fichier exécutable : il est composé d'un sous-ensemble de l'union de tous les sous-programmes, variables et constantes utilisées lors de l'exécution de la fonction *main*. C'est au moment de l'édition de lien que la cohérence est vérifiée : tous les symboles utilisés (sous-programmes, variables, constantes) sont-ils implémentés ? Pour trouver chaque symbole, l'éditeur de liens puise les définitions et implémentations requises dans les objets et librairies fournis. Pendant l'élaboration, il vérifie qu'il n'y a aucun symbole défini plusieurs fois.

Bien entendu, ces phases sont gérées de façon transparente par la plupart des environnements de développement intégrés, mais il est important de connaître le déroule-

ment précis du processus afin de comprendre et d'éviter les erreurs dues à une hiérarchie de compilation mal construite.
La compilation d'un module s'effectue en deux phases : un **préprocesseur** interprète un langage macro afin de remplacer toutes les macros par leur valeur. Le fichier ainsi « *préprocessé* », est compilé en un objet (figure 6.19). Le langage macro commence chacune de ses instructions par le caractère #. Ainsi, l'instruction `#define` permet d'associer un nom logique à une suite d'instructions. Typiquement, ce type de définition est utilisé pour les constantes de programme et les macros. Ainsi, sur la figure 6.19, les constantes *MAX_TEMP* et *MIN_TEMP* sont substituées par leur valeur par le préprocesseur.

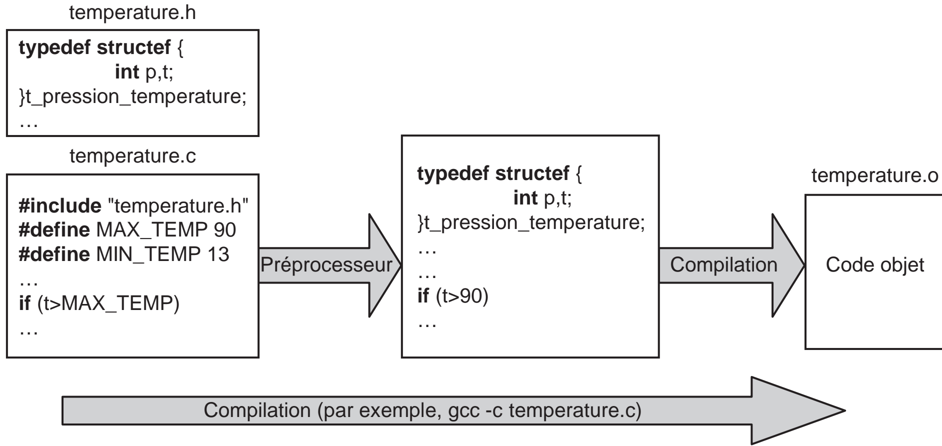


Figure 6.19 – Préprocesseur C.

C'est aussi une instruction préprocesseur qui permet de faire référence à des fichiers d'en-têtes. Le préprocesseur se contente d'insérer leur contenu au fichier *préprocessé*. Lors de la compilation du fichier ainsi obtenu, le compilateur ne voit donc aucune différence entre corps de module (le fichier .c) et en-têtes de module. Il en résulte que l'édition de lien du code suivant génère une erreur :

```
/* Fichier temperature.h
Module contenant des définitions de fonctions de gestion de la
température courante */
float temperature_courante;
/* Variable globale
température courante en °C */

float temperature_K();
/* retourne la température courante en °K */

float temperature_F();
/* retourne la température courante en °F */

void mettre_a_jour_temperature();
/* Met à jour la température courante en °C */
```

© Dunod – La photocopie non autorisée est un délit.

```
/* Fichier temperature.c
Corps de module contenant l'implémentation des fonctions de gestion de
la température courante */
#include "temperature.h"
/* implementation des fonctions déclarées dans
l'en-tête de module */
float temperature_K()
{
...
}
...

/* Fichier main.c contenant le programme principal */
#include "temperature.h"
void main() {
/* programme principal utilisant le module temperature */
...
}
```

En effet, lors de la compilation du fichier `temperature.c`, la variable `temperature_courante` est déclarée. Lors de la compilation du fichier `main.c`, la même variable est déclarée de nouveau. Par conséquent, lors de l'édition de lien, une erreur vient du fait que cette variable est déclarée plusieurs fois.
Par conséquent, il est indispensable de pallier ce type de problème lorsque l'on programme de façon modulaire en langage C. L'une des techniques les plus utilisées consiste à ne jamais faire de déclaration de variable dans un en-tête, et à protéger un en-tête contre une insertion multiple dans le même fichier. Sur l'exemple de code précédent, la façon de procéder devient alors :

```
#ifndef _TEMPERATURE_H_
/* Ce qui suit est ignoré jusqu'à l'instruction #endif
si le symbole _TEMPERATURE_H_ est déjà défini */
#define _TEMPERATURE_H_
/* Définition préprocesseur empêchant le fichier
d'être inséré plusieurs fois dans la compilation
du même fichier .c : le symbole _TEMPERATURE_H_ est
défini lors de la première inclusion de ce fichier
lors de la phase de préprocessing*/

extern float temperature_courante;
/* Référence externe à une variable globale,
la variable globale n'est pas déclarée, mais on dit
explicitement au compilateur qu'elle est déclarée quelque
part dans le programme. En l'occurrence, elle est déclarée
dans le corps du module */

float temperature_K();
/* retourne la température courante en °K */

float temperature_F();
/* retourne la température courante en °F */

void mettre_a_jour_temperature();
/* Met à jour la temperature courante en °C */
#endif
/* Fin du fichier d'en-tête */
```

```
/* Fichier temperature.c
Corps de module contenant l'implémentation des fonctions de gestion de
la température courante */
#include "temperature.h"
/* implementation des fonctions déclarées dans
l'en-tête de module */
float temperature_courante;
/* déclaration effective de la variable globale */
float temperature_K()
{
...
}
...

/* Fichier main.c contenant le programme principal */
#include "temperature.h"
void main() {
/* programme principal utilisant le module temperature */
...
}
```

□ Langage Ada

Contrairement aux compilateurs C, les compilateurs Ada différencient corps de module et en-tête de module. Un en-tête de module, ou spécification de module, est un fichier d'extension *.ads* (pour *Ada specification*). Un corps de module est implémenté dans un fichier d'extension *.adb* (pour *Ada body*). Un module Ada est appelé paquetage ou ***package***. Il y a trois sortes d'unités de compilation en langage Ada : les spécifications de paquetages, les corps de paquetage et les procédures ou fonctions pouvant se trouver directement dans une unité de compilation (extension *.adb*). Typiquement dans un programme modulaire, on trouve un certain nombre de paquetages et une unité de compilation contenant une procédure : le programme principal.
La figure 6.20 présente une hiérarchie de programme modulaire en langage Ada.

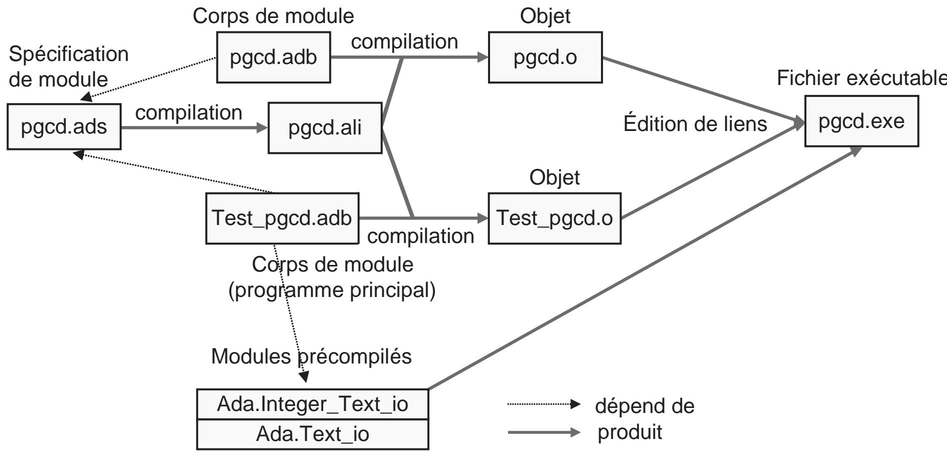


Figure 6.20 – Hiérarchie lors de l'élaboration d'un programme écrit en langage Ada.

© Dunod – La photocopie non autorisée est un délit.

Sur cette figure, nous distinguons un module pgcd (paquetage pgcd) décomposé en un corps (fichier *pgcd.adb*) et une spécification (fichier *pgcd.ads*) dont le contenu serait le suivant :

```
package Pgcd is
      function Calcul_Pgcd (I : Integer; J : Integer) return Integer;
      -- Entrée: I,J deux entiers
      -- Retourne: le plus grand commun diviseur de I et J
      -- Convention: pgcd(0,J) = J, pgcd(I,0) = I
end Pgcd;
```

On peut noter l'emploi du mot clé `package` permettant de définir une unité de compilation de type spécification de module.
Le code source du corps de paquetage, défini par les mots clés `package body`, est donné ci-après :

```
package body Pgcd is
   -- Noter que contrairement à C, il est inutile de préciser
   -- que le corps de paquetage utilise la spécification: cela est
   -- implicite
   function Calcul_Pgcd (I : Integer; J : Integer) return Integer is
      -- Entrée: I,J deux entiers
      -- Retourne: le plus grand commun diviseur de I et J
      -- Convention: pgcd(0,J) = J, pgcd(I,0) = I
      R,A,B : Integer;
   begin
      A:=abs(I); -- Travail en valeur absolue
      B:=abs(J);
      if B>A then
         return Calcul_Pgcd(B,A);
      else
         R := A mod B; -- Reste de la division entière
         if R=0 then
            return B;
         else
            return Calcul_Pgcd(B,R);
         end if;
      end if;
   end; -- Fin de la fonction Calcul_Pgcd
end Pgcd; -- Fin du paquetage
```

La procédure principale, placée dans le fichier *Test_pgcd.adb*, est :

```
with Ada.Text_Io, Ada.Integer_Text_Io,pgcd;
use Ada.Text_Io, Ada.Integer_Text_Io,pgcd;
-- utilisation des paquetages prédéfinis définissant les
-- entrées/sorties de type texte et entier,
-- ainsi que du paquetage pgcd
procedure Test_Pgcd is -- Procédure principale
   A,
   B : Integer;
begin
   Put_Line("Donnez deux entiers"); -- Affichage de texte
   Get(A); -- Lecture des deux entiers au clavier
   Get(B);
   Put(Calcul_Pgcd(A,B)); -- Affichage du résultat
end;
```

L'emploi des mots clés `with` et `use`, placés généralement en début d'unité de compilation, permet de faire référence aux spécifications de paquetage concernées. L'emploi de la clause `with` *M* donne à une unité de compilation la visibilité des types, variables, sous-programmes, etc. définis dans la spécification du module *M*. Cependant, l'emploi de ces éléments doit alors être préfixé par le nom du module. Ainsi, pour utiliser la fonction *F* définie dans le module *M*, il faut écrire son nom complet, soit *M.F.* On peut utiliser la clause `use` *M*, qui permet d'obtenir une visibilité directe, ce qui permet d'appeler la fonction *F* sans préfixer son nom par *M*.
Ada définit en fait des **espaces de noms** liés aux modules. Ainsi, toute ambiguïté peut être levée si une unité de compilation utilise plusieurs modules définissant des éléments ayant le même nom.
Notons enfin que dans tous les cas, il est fortement recommandé d'utiliser un nom de fichier identique au nom du paquetage ou du sous-programme contenu (*i.e.* si un paquetage s'appelle *temperature*, alors les fichiers l'implémentant s'appellent *temperature.ads* et *temperature.adb*).
De nombreux concepts permettant de gérer finement la visibilité des types et sous-programmes contenus dans les paquetages, la généricité, les modules imbriqués, etc., ne sont pas explicités ici.

□ Langage LabVIEW

Le langage LabVIEW permet une modularité par sous-programme (*vi*) : en effet, lorsqu'un sous-*vi* est créé, il peut être intégré dans un autre *vi* directement par nom de fichier (par glisser/déposer du fichier sur le diagramme du *vi* par exemple). Généralement, on organise les *vi* par types dans des dossiers du système de fichiers, en les regroupant par thème.

■ Traitement des erreurs

□ Langage C

La gestion des erreurs en langage C est relativement basique : une partie des fonctions susceptibles de renvoyer une erreur (accès à un fichier, au réseau, conversions chaîne de caractères/nombre, etc.) renvoient un entier représentant le statut d'erreur. Généralement, une valeur de retour nulle est renvoyée s'il n'y a pas eu d'erreur, et -1 ou un code d'erreur est renvoyé en cas d'erreur. Dans le cas où le code d'erreur lui-même n'est pas renvoyé par la fonction, celle-ci modifie une variable globale, nommée `errno` (numéro d'erreur), qui contient le code de l'erreur venant de survenir.
Les fonctions renvoyant un pointeur (allocation mémoire, ouverture de fichier, etc.) renvoient un pointeur à *null* (valant 0) lorsqu'une erreur est survenue. Là encore, la variable `errno` peut être consultée afin de connaître le code de l'erreur.
Par conséquent un code source C traitant les erreurs est une succession de structures conditionnelles : la valeur de retour de chaque fonction susceptible de générer une erreur est testée. Si l'on s'aperçoit d'une erreur, celle-ci est immédiatement traitée. Ce type de mécanisme diminue la lisibilité des programmes.
Nous présentons ci-dessous un exemple de code C gérant les erreurs : dans le cas normal, un fichier est ouvert, un traitement non détaillé ici a lieu, puis le fichier est fermé.

© Dunod – La photocopie non autorisée est un délit.

```
#include <stdio.h>
/* Inclusion de la librairie système d'entrées/sorties standard
Noter l'emploi de <nomfichier> au lieu de "nomfichier"*/
#include <errno.h>

int main () {
      FILE *f; /* f est un descripteur de fichier, destiné à pointer
sur une structure représentant un fichier */
      int retval;/* variable utilisé lorsque l'on veut conserver une
valeur de retour */
      if (!(f=fopen("monfichier","r"))) {/* Ouverture d'un fichier en
lecture. */
            /* Si fopen renvoie null */
            /* Noter l'emploi des doubles parenthèses :
en langage C, l'affectation renvoie une valeur, ainsi, l'expression
(a=b), en plus d'affecter b à a, renvoie la valeur de b. Cette valeur
peut alors être testée : est-elle nulle ? */
            printf("Erreur %d lors de l'ouverture du fichier",errno);
            /* affichage du numéro d'erreur */
            return(errno); /* Terminaison du programme en renvoyant le
code d'erreur */
      }
      /* Traitements divers */
      if ((retval=fclose(f))) { /* Fermeture du fichier. */
/* Si fclose renvoie une valeur non nulle, c'est un code d'erreur */
            printf("Erreur %d lors de l'ouverture du fichier",retval);
            /* affichage du numéro d'erreur */
            return(retval); /* Terminaison du programme en renvoyant
le code d'erreur */
      }
      return 0; /* Normalement, un programme se terminant sans erreur
devrait renvoyer la valeur 0 */
}
```

Cet exemple met en évidence l'un des écueils liés à l'utilisation du C : le traitement des erreurs rend le code difficile à lire.
Notons enfin que le mécanisme utilisant `errno` n'est pas compatible avec le multitâche : en effet, rien n'empêche une autre tâche de modifier le contenu de la variable `errno` entre le moment où l'erreur survient dans une tâche et le moment où celle-ci lit la valeur du code d'erreur. Par conséquent, certains exécutifs, comme VxWorks® par exemple, définissent une variable `errno` par tâche.

☐ Langage Ada

Ada gère les erreurs suivant un mécanisme **d'exception**. Le concept d'exception est proche du concept d'interruption logicielle présenté au chapitre 5. Chaque bloc d'instructions peut être muni d'une partie « traite exception ». Si une erreur a lieu, une exception correspondant au type d'erreur est levée. L'exécution du bloc en cours est alors interrompue. Si le bloc contient un traitement pour cette exception, alors ce traitement est exécuté, le bloc se termine, et le programme continue au niveau du bloc appelant ou englobant. Si le bloc ne contient pas de traitement pour cette exception, alors l'exception est propagée au bloc appelant ou englobant. Si une exception n'est pas traitée, elle arrive au bloc de plus haut niveau (programme principal, ou bien tâche) et ce bloc arrête son exécution. Si c'est dans le programme principal

lui-même que l'exception est propagée, alors le programme s'arrête (un message d'erreur est alors affiché).

Ce mécanisme permet de gérer les erreurs sans nuire à la lisibilité d'un programme, contrairement à la gestion des erreurs en langage C. Ainsi, l'exemple traité en langage C au paragraphe 6.1.2, p. 276 est repris ci-après en langage Ada :

```
with Ada.Text_Io;
use Ada.Text_Io;
-- Paquetage standard permettant les entrées/sorties
procedure Test_Fichier is
   F: File_Type;
   -- Descripteur de fichier
begin
   Open(F,In_File,"monfichier");
   -- Ouverture d'un fichier en lecture
   -- Actions diverses sur le fichier
   Close(F);
   -- Fermeture du fichier
exception
   when Name_Error => Put_Line("Erreur de nom de fichier");
   -- Traitement spécifique de l'exception Name_Error
   when others => Put_Line("Erreur sur le fichier");
   -- Traitement pour toute exception
end;
```

Remarquons le traitement d'erreurs séparé du code, et la lisibilité améliorée. Cependant, notons qu'il est parfois difficile de savoir qu'est-ce qui est à l'origine de l'exception. Il est possible de générer soi-même une exception. Notons qu'une des limitations du langage est qu'une exception ne peut pas transporter de valeur, ce qui est assez dommage.

☐ Langage LabVIEW

La figure 6.21 présente un exemple de gestion d'erreur en langage LabVIEW.

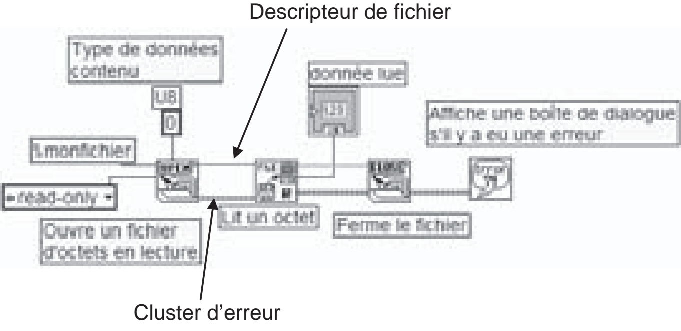


Figure 6.21 – Gestion des erreurs en langage LabVIEW.

Un *vi* pouvant lever une erreur possède une sortie de type cluster d'erreur (figure 6.22) généralement nommée `error out`. La plupart des *vi* susceptibles de lever des erreurs possèdent une entrée et une sortie d'erreur. En cas d'erreur dans un *vi*, sa sortie

© Dunod – La photocopie non autorisée est un délit.

d'erreur contient les informations d'erreur. Un *vi* ayant une erreur sur son entrée d'erreur ne fait rien, et se contente de relayer l'erreur sur sa propre sortie d'erreur. Ainsi, si une chaîne de traitement voit se lever une erreur, les *vi* s'exécutent sans rien faire, si ce n'est relayer l'erreur au reste de la chaîne. Généralement, on traite donc l'erreur *a posteriori*, en vérifiant l'état de la dernière sortie d'erreur d'une chaîne de traitement.

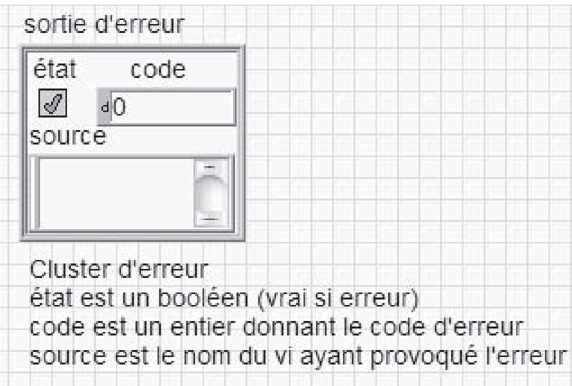


Figure 6.22 – Représentation d'une erreur en langage LabVIEW.

Sur la figure 6.21, le traitement d'erreur consiste à afficher une boîte de dialogue décrivant celle-ci. Notons que depuis la version 7 de LabVIEW, une erreur non traitée entraîne l'affichage d'une boîte de dialogue de description de l'erreur (ce mode de fonctionnement peut être inhibé afin de garantir une compatibilité ascendante au niveau des versions). Ce type de traitement se rapproche alors de l'exception.

■ Programmation bas niveau

□ Introduction

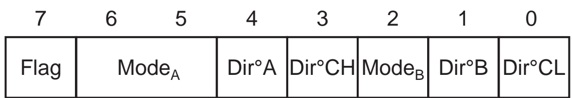
Sur les microcontrôleurs et parfois sur les microprocesseurs, il est nécessaire de programmer au niveau des registres en particulier, à cause de l'absence de pilotes de périphérique pour certains systèmes d'exploitation ou exécutifs. La programmation bas niveau est généralement effectuée en langage C, bien qu'Ada le permette (§ 6.1.2, p. 282). LabVIEW, quant à lui, ne propose que très peu d'outils de très bas niveau, mais il possède un grand nombre de pilotes de périphériques. Si un périphérique n'est pas supporté par LabVIEW, un pilote peut être écrit en langage C et facilement interfacé avec LabVIEW.

Dans cette section, nous considérons l'exemple suivant : la carte *National Instruments PCI-DIO-24/PnP* est une carte d'acquisition très simple au format PCI, avec 3 ports de 8 entrées/sorties numériques, capable de déclencher des interruptions. Supposons qu'il n'existe pas de pilote de périphérique pour l'exécutif utilisé avec cette carte. Il est alors nécessaire de programmer directement la carte au niveau registres.

Les registres permettant à l'unité centrale de communiquer avec un périphérique d'entrées/sorties sont généralement liés à une adresse, dite d'entrée/sortie, en mémoire.

Lire ou modifier une adresse d'entrée/sortie correspond à lire ou modifier un registre du dispositif d'entrées/sorties correspondant.

Si l'on ne se sert pas de la possibilité de génération d'interruptions matérielles, la carte *PCI-DIO-24/PnP* est caractérisée par 4 registres d'un octet. Le 4e octet est le registre de configuration de la carte (figure 6.23). Afin de simplifier sa programmation, nous nous limitons au mode simple pour les trois ports : toutes les lignes vont dans le même sens sur le port A, il en est de même pour le port B, alors que le port C est coupé en deux parties (de 4 bits) pouvant chacunes être configurées dans un sens.



Flag : mode de programmation du registre
(0 : configuration du mode, 1 : configuration bits)
Mode$_A$: sélection du mode du port A et des lignes 4 à 7 du port C
(00 : mode simple, 01 : mode synchronisé, 10 et 11 : mode bidirectionnel)
Dir°A : direction des 8 lignes du port A (0 : sortie, 1 : entrée)
Dir°CH : direction des lignes 4 à 7 du port C (0 : sortie, 1 : entrée)
Mode$_B$: sélection du mode du port B et des lignes 0 à 3 du port C
(0 : mode simple, 1 : mode synchronisé)
Dir°B : direction des 8 lignes du port B (0 : sortie, 1 : entrée)
Dir°CL : direction des lignes 0 à 3 du port C (0 : sortie, 1 : entrée)

Figure 6.23 – Registre de configuration d'une carte d'acquisition.

Il est généralement possible de configurer l'**adresse de base** liée aux registres d'un dispositif d'entrées/sorties : ainsi, par exemple, en fonction de la configuration matérielle, les adresses des registres de la carte pourront se trouver en 0x210-0x213. Dans ce cas, le premier registre, dont les 8 bits correspondent au port A de la carte, se trouve en 0x210, le registre correspondant au port B en 0x211, le registre correspondant au port C en 0x212 et le registre de configuration en 0x213.

□ Langage C

La configuration de la carte peut se faire de deux façons : soit en utilisant directement un octet créé pour correspondre à la configuration désirée, soit en définissant une fonction de configuration, réutilisable pour cette carte d'acquisition.

Dans le premier cas, si les ports doivent être configurés de sorte à ce que le port A et les lignes 0 à 3 du port C soient en sortie, et le reste en entrée, on peut créer l'octet de configuration : en binaire b00001010, soit en hexadécimal 0x0A. L'initialisation de la carte consisterait donc à écrire 0x0A à l'adresse 0x213 (par exemple en exécutant une instruction d'écriture à une adresse d'entrée/sortie comme `OutByte(0x213,0x0A)`). Cette solution est simple mais il est nécessaire de relire plusieurs pages de documentation pour chaque changement de configuration.

La seconde solution, plus élégante, consiste à créer un pilote de périphérique (*driver*) simplifié. Pour cela, C permet de définir assez finement un type facilitant les opérations bit à bit sur le registre de configuration :

© Dunod – La photocopie non autorisée est un délit.

```
#ifndef BYTE
#define BYTE unsigned char /* Octet non signé */
#endif
typedef union { /* Correspond au registre de configuration : on peut y
accéder indifféremment par octet (champ config) ou par bits */
    BYTE config; /* Octet complet */
    struct { /* Définit chaque champ (sur 1 ou 2 bits en fonction du
champ) */
        BYTE DirCL:1;
        BYTE DirB: 1;
        BYTE ModeB:1;
        BYTE DirCH:1;
        BYTE DirA:1;
        BYTE ModeA:2;
        BYTE Flag:1;
    } ;
} t_DIO24_config;
```

Noter la façon de donner dans un type enregistrement (`struct`) la taille en bits de chaque champ. Grâce à l'utilisation d'une `union`, on peut accéder de deux façons à l'octet : directement en tant qu'octet, ou bien champ par champ. Ainsi, par exemple, le champ `DirB` correspond au deuxième bit (en partant du bit de poids faible) de l'octet.

Si l'on souhaite créer un pilote de périphérique à même de gérer plusieurs cartes de ce type, chacune à une adresse de base, on peut créer une fonction d'initialisation comme celle qui suit (remarquer l'utilisation de l'union définissant l'accès au registre de contrôle) :

```
/* Décalage des registres par rapport à l'adresse de base */
typedef enum {PORTA=0, PORTB=1, PORTC=2, CONFIG=3} Port_DIO24;

/* Définition d'un type représentant la carte DIO24 en mode simple */
typedef struct {
    int base; /* Adresse de base de la carte */
   BYTE valeurs[3]; /* Mémorise les dernières valeurs écrites sur les
ports, sert à fournir des opérations d'écriture avec masque binaire */
    t_DIO24_config config;
} t_DIO24; /* Représente l'état d'une carte DIO24 */
void ConfigDIO24(int adresse_base, char DirectionA, char DirectionB,
char DirectionCH, char DirectionCL, t_DIO24* DIO24) {
/* Entrées: adresse_base est l'adresse de base de la carte. La
direction des ports est 1 pour sortie, 0 pour entrée
   Sorties: DIO24 est une structure renseignée dans cette fonction,
est utilisée dans les fonctions d'entrées/sorties sur la carte*/
    (*DIO24).base=adresse_base;
    (*DIO24).config.config=0; /* l'octet passe à 0, donc tous les
champs valent 0 dans l'union représentant le registre de configuration
*/
    (*DIO24).config.DirA= !DirectionA; /* Si Direction vaut 0
(entrée), le bit correspondant doit être mis à 1 */
    (*DIO24).config.DirB= !DirectionB;
    (*DIO24).config.DirCH= !DirectionCH;
    (*DIO24).config.DirCL= !DirectionCL;
    /* Par défaut, les ports initialisés en sortie sont mis à 0 */
   memset((*DIO24).valeurs,0,3); /* Mise à 0 du tableau des dernières
valeurs écrites */
```

```
    outByte(adresse_base+CONFIG,(*DIO24).config.config); /* Ecriture
de la configuration à l'adresse du port de contrôle */
}
```

Sur un microprocesseur, lorsqu'on écrit à une adresse d'entrée/sortie, il est généralement nécessaire d'utiliser une fonction spécifique, comme `outByte` dans l'exemple. Sur un microcontrôleur, l'instruction `outByte(adresse,valeur)` serait généralement remplacée directement par `(*adresse)=valeur`.
Afin de donner un exemple de manipulation binaire sur un cas concret (voir § 4.1.3, p. 123), étudions la manière de programmer l'écriture sur un port de la carte (soit d'un octet avec masque, soit d'une seule ligne), ainsi que la lecture.

```
void EcrirePort(t_DIO24 *DIO24,Port_DIO24 port,BYTE valeur,BYTE
masque) {
/* Modifie un port configuré en sortie : utilise la technique du
masque binaire comme vu sur la figure 4.10
  Nécessite: DIO24 préalablement configuré par ConfigDIO24 */

(*DIO24).valeurs[port]=(valeur&masque)|(~masque&(*DIO24).valeurs[port
]); /* Calcul de la nouvelle valeur en fonction de l'ancienne et du
masque */
    outByte((*DIO24).base+port,(*DIO24).valeurs[port]); /* Ecriture à
l'adresse d'entrée/sortie correspondant au port */
}

void EcrireLigne(t_DIO24 *DIO24, Port_DIO24 port, BYTE ligne, BYTE
valeur) {
/* Modifie une ligne d'un port configuré en sortie
  Nécessite: DIO24 préalablement configuré par ConfigDIO24 */
    if (valeur) /* Mise à 1 */
        (*DIO24).valeurs[port]|=(1<<ligne);
    else /* Mise à 0*/
        (*DIO24).valeurs[port]&= ~(1<<ligne);
    outByte((*DIO24).base+port,(*DIO24).valeurs[port]); /* Ecriture à
l'adresse d'entrée/sortie correspondant au port */
}

BYTE LirePort(t_DIO24 *DIO24,Port_DIO24 port) {
/* Lit l'état d'un port configuré en entrée
   Nécessite: DIO24 préalablement configuré par ConfigDIO24 */
    return inByte((*DIO24).base+port);/* Lecture de l'adresse
d'entrée/sortie correspondant au port */
}

BYTE LireLigne(t_DIO24 *DIO24,Port_DIO24 port, BYTE ligne) {
/* Lit l'état d'une ligne d'un port configuré en entrée
   Nécessite: DIO24 préalablement configuré par ConfigDIO24 */
    return (inByte((*DIO24).base+port)>>ligne)&1; /* Lecture du port,
décalage de bit pour obtenir la valeur de la ligne sur le bit de poids
faible, suppression des autres 1 éventuels de la valeur renvoyée */
}
```

Remarquons la façon dont le masque binaire est géré pour l'écriture d'un octet (figures 4.9 et 4.10). Une erreur classique est d'utiliser le « non » logique (point d'exclamation) à la place du complément binaire (tilde). Notons aussi que pour simplifier, les erreurs (tentative d'écriture sur un port configuré en lecture, etc.) ne sont pas gérées.

© Dunod – La photocopie non autorisée est un délit.

Voici un exemple d'utilisation de ce pilote de périphérique :

```
  t_DIO24 DIO; /* Structure représentant une carte configurée */
  ConfigDIO24(0x210, 1, 0, 0, 1, &DIO); /* Configuration : adresse
0x210, A et CL en écriture, B et CH en lecture */
  EcrireLigne(&DIO,PORTA,0,1); /* Mise à 1 de la ligne 0 du port A */
  EcrirePort(&DIO,PORTA,0x8F,0xF0); /* Mise à 1 de la ligne 7, et à 0
des lignes 6, 5, 4 du port A */
```

☐ **Langage Ada**

Comme C, Ada permet de gérer très finement les représentations binaires associées à un type. Par exemple, la définition d'un type octet peut s'effectuer de la façon suivante :

```
type Octet is mod 256; -- Intervalle de représentation du type
                       -- correspondant à un octet
for Octet'Size use 8; -- Taille en bits utilisée pour la représentation
```

Ada permet de maîtriser la taille associée à un type, et son intervalle de représentation. Ainsi, le type suivant pourrait permettre d'accéder champ par champ à l'octet de contrôle de la carte d'acquisition PCI-DIO-24/PNP (figure 6.23).

```
type Bit is range 0..1; -- Domaine de valeur de 1 bit
type Bitx2 is range 0..3; -- Domaine de valeur de 2 bits
type Direction_Port is (sortie,entree); -- Type enuméré utilisé pour
                                        -- définir la direction d'un port
type T_Control_Dio24 is record -- Définition d'un enregistrement
                               -- correspondant au registre de contrôle
   Dir_CL : Direction_Port;
   Dir_B : Direction_Port;
   Mode_B : Bit;
   Dir_Ch : Direction_Port;
   Dir_A : Direction_Port;
   Mode_A : Bitx2;
   Mode : Bit;
end record;
for T_Control_Dio24 use record -- Clause de représentation: le type
                               -- se plaque sur l'octet de contrôle
   Dir_CL at 0 range 0..0;
   Dir_B  at 0 range 1..1;
   Mode_B  at 0 range 2..2;
   Dir_Ch at 0 range 3..3;
   Dir_A at 0 range 4..4;
   Mode_A at 0 range 5..6;
   Mode  at 0 range 7..7;
end record;
```

La syntaxe utilisée pour les clauses de représentation, bien que plus flexible que celle du langage C, n'en demeure pas moins plus lourde.
Afin de représenter un port configuré comme cela est effectué en langage C dans la section précédente, le type enregistrement suivant peut être défini :

```
type T_Port_DIO24 is (A, B, C); -- Valeurs des ports, noter que la
                                -- position de A vaut 0, celle de B 1
                                -- et celle de C 2
type T_Valeurs_De_Port is array(A..C) of Octet;
type T_Dio_24 is record -- Type utilisé pour représenter la carte
                        -- DIO24 (en mode simple)
```

```
   Base: System.Address; -- Adresse de base
   Valeurs: T_Valeurs_De_Port; -- Dernière valeur écrite sur chaque
                               -- port
   Config: T_Control_DIO24; -- Octet de configuration
end record;
```

La procedure de configuration de la carte donnée ci-après montre comment associer une adresse mémoire à une variable. Noter qu'il est nécessaire d'utiliser le paquetage `System`.

```
procedure Config_Dio24(Base: System.Address; Dir_A,
Dir_B,Dir_CH,Dir_CL: Direction_Port;Dio24: out T_Dio_24) is
-- Entrées: base est l'adresse de base de la carte. Les paramètres
-- Dir_x donnent la direction des ports
-- Sorties: DIO24 est créé: il est utilisé dans les fonctions
-- d'entrées/sorties sur la carte
   Control_Reg: T_Control_DIO24; -- Correspond au registre de contrôle
                                 -- (sur 1 octet)
   for Control_Reg'Address use Base+3; -- Adresse du registre de
                                       -- contrôle=Base+3
begin
   Dio24.Base:=Base;
   for Port in T_Port_Dio24'range loop
      Dio24.Valeurs(Port):=0;
   end loop;
   Dio24.Config.Mode:=0; -- Tous les champs de mode valent 0
   Dio24.Config.Mode_A:=0;
   Dio24.Config.Mode_B:=0;
   Dio24.Config.Dir_A:=Dir_A; -- Les champs de direction correspondent
                              -- à 0 pour sortie, 1 pour entrée
   Dio24.Config.Dir_B:=Dir_B;
   Dio24.Config.Dir_CH:=Dir_CH;
   Dio24.Config.Dir_Cl:=Dir_Cl;
   Control_Reg:=Dio24.Config; -- Mise à jour effective du registre de
                              -- contrôle
end;
```

La lecture et l'écriture des ports utilisent le même principe. Noter que ce sont les mêmes opérateurs que pour les booléens qui opèrent en mode binaire lorsque les types manipulés sont de type `mod` comme le type `octet`.

```
procedure Ecrire_Port(Dio24: in out T_Dio_24; Port: T_Port_Dio24;
Valeur, Masque: Octet) is
-- Modifie un port configuré en sortie : utilise la technique du
masque binaire comme vu sur la figure 4.10
   -- Nécessite: DIO24 préalablement configuré par Config_DIO24
   Port_Reg: Octet; -- Correspond à l'adresse d'entrée/sortie du
                    -- registre lié au port à modifier
   for Port_Reg'Address use Dio24.Base+(T_Port_Dio24'Pos(Port));
-- Par construction, la position du port dans le type énuméré est le
-- décalage par rapport à l'adresse base
begin
   Dio24.Valeurs(Port):=(Valeur and Masque) or ((not Masque) and
Dio24.Valeurs(Port)); -- Calcul de la nouvelle valeur en fonction de
                      -- l'ancienne et du masque
   Port_Reg:=Dio24.Valeurs(Port); -- Mise à jour effective du registre
                                  -- du port
end;
procedure Ecrire_Ligne(Dio24: in out T_Dio_24; Port: T_Port_Dio24;
Ligne: Integer; Valeur: Boolean) is
```

© Dunod – La photocopie non autorisée est un délit.

```
-- Modifie une ligne d'un port configuré en sortie
-- Nécessite: DIO24 préalablement configuré par Config_DIO24
   Port_Reg: Octet;
   for Port_Reg'Address use Dio24.Base+(T_Port_Dio24'Pos(Port));
begin
   if Valeur then
      Dio24.Valeurs(Port):=Dio24.Valeurs(Port) or 2**Ligne;
-- Mise à 1 du bit ligne, 2**i correspond en binaire à un nombre
-- composé d'un 1 sur le bit i
   else
      Dio24.Valeurs(Port):=Dio24.Valeurs(Port) and not (2**Ligne);
-- Mise à 0 du bit ligne
   end if;
   Port_Reg:=Dio24.Valeurs(Port);
end;
function Lire_Port(Dio24: T_Dio_24; Port: T_Port_Dio24) return Octet
is
-- Lit l'état d'un port configuré en entrée
-- Nécessite: DIO24 préalablement configuré par Config_DIO24
   Port_Reg: Octet;
   for Port_Reg'Address use Dio24.Base+(T_Port_Dio24'Pos(Port));
begin
   return Port_Reg;
end;
function Lire_Ligne(Dio24: T_Dio_24; Port: T_Port_Dio24; Ligne:
Integer) return Boolean is
-- Lit l'état d'une ligne d'un port configuré en entrée
-- Nécessite: DIO24 préalablement configuré par Config_DIO24
   Port_Reg: Octet;
   for Port_Reg'Address use Dio24.Base+(T_Port_Dio24'Pos(Port));
begin
   if (Port_Reg and 2**Ligne)/=0 then
      return True;
   else
      return False;
   end if;
end;
```

Étant donné qu'Ada est fortement typé, l'utilisation du pilote de périphérique ainsi programmé nécessite l'emploi de la fonction de conversion d'entier en adresse, située dans le paquetage `System.Storage_Elements`.

```
Dio24 : T_Dio_24;
begin
   Config_Dio24(To_Address(16#210#),Sortie, Entree, Sortie,
Entree,Dio24);
   Ecrire_Ligne(Dio24,A,0,True);
   Ecrire_Port(Dio24,A,16#8F#, 16#F0#);
end;
```

Cet exemple montre qu'Ada permet la programmation bas niveau. Il faut cependant jongler avec les types et les conversions, tout en ayant conscience de la représentation binaire de ce que l'on manipule.

Dans les faits, la programmation bas niveau est souvent effectuée en langage C. Il est donc très fréquent qu'un pilote de périphérique soit disponible en langage C, mais pas en langage Ada. Dans ce cas, plutôt que de réécrire totalement le pilote, un *binding* (interface entre deux langages) Ada est créé.

6.2 Programmation multitâche en langage C

Comme cela a été notifié précédemment, le langage C n'offre pas de support natif pour le multitâche. Il existe donc différentes normes (POSIX, OSEK/VDX, etc.) ou bibliothèques propriétaires (VxWorks®, RTEMS, etc.) permettant une programmation multitâche.
Ce sous-chapitre montre donc comment on peut implémenter une conception DARTS dans une norme (POSIX) et une bibliothèque propriétaire (VxWorks®).

6.2.1 Implémentation en tâches POSIX

■ Interface POSIX 1003.1c et 1003.1j

L'interface multitâche (*pthread*) définie par la norme POSIX se trouve dans les amendements 1003.1c (intégrée dans 1003.1 depuis 2001) et 1003.1j. L'apport principal de l'amendement 1003.1j est l'ajout de la possibilité de spécifier l'utilisation de différentes horloges (notamment CLOCK_MONOTONIC).

■ Implémentation des éléments DARTS

□ Tâches

Une tâche (*pthread*) est représentée par une fonction pouvant posséder au plus un paramètre de type `void *`. Ce paramètre peut être utilisé pour passer une donnée de la taille d'un pointeur (généralement, on peut y passer un entier, ou bien n'importe quel pointeur).
Chaque tâche se lance dynamiquement grâce à la fonction `pthread_create`. La signature de cette fonction est :

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine)(void*), void *arg);
```

Le paramètre `thread` est un paramètre de sortie, qui contiendra l'identificateur de la tâche nouvellement créée. Cette valeur pourra être utilisée dans d'autres fonctions de manipulation de tâche.
Le paramètre `start_routine` est un pointeur vers la fonction implémentant la tâche. Le dernier paramètre, `arg`, est le paramètre éventuel passé à cette fonction.
Le paramètre le plus complexe est `attr` : en effet, ce paramètre sert de « fourre-tout ». On y trouve des paramètres de configuration d'une tâche comme :

– le fait qu'elle soit détachée ou pouvant être attachée (par défaut). Dans le cas où une tâche est détachée, il n'est pas possible d'attendre directement sa terminaison. Si une tâche est attachée à une autre, ou bien au programme principal, alors il est possible d'attendre sa terminaison ;
– la taille et l'adresse de sa pile ;
– son type d'ordonnancement (la façon dont on ordonnance les tâches filles de la tâche créée, ce qui importe peu dans le cas où toutes les tâches sont au même niveau hiérarchique), sa priorité, et le fait qu'elle soit ordonnancée localement (à l'intérieur du processus père) ou de façon globale.

© Dunod – La photocopie non autorisée est un délit.

Enfin, la valeur de retour est 0 si la tâche a bien été créée, un code d'erreur sinon.
Il est possible de créer une hiérarchie de tâches, mais dans la méthode DARTS, que nous implémentons, toute tâche est fille directe du programme principal. Par conséquent, au lancement de la tâche, son masque de signaux est le même que celui du programme principal. Il pourra ensuite être modifié.
Un exemple très simple de programme multitâche POSIX est donné ci-après :

```
#include <pthread.h> /* Librairie POSIX threads */
#include <stdio.h> /* Entrées/Sorties (printf) */

void f(int no) { /* Fonction implémentant une tâche */
    int i;
    for (i=0;i<10;i++) {
        printf("Tache %d : iteration %d\n",no,i);
        usleep(1000000); /* Attente d'au moins une seconde */
    }
}

void main() {
    pthread_t tache1, tache2;/* identificateurs de tâches */
    pthread_create(&tache1, 0,f, (void *)1);
    /* Création et lancement d'une instance de la tâche f. Observer
l'utilisation du paramètre pour passer un entier. Noter qu'en passant
la valeur nulle au paramètre d'attribut, les attributs par défaut sont
choisis, et notamment, la tâche peut être attachée */
    usleep(2000000); /* Attente d'au moins 2 secondes */
    pthread_create(&tache2, 0,f, (void *)2); /*Création de la seconde
tâche, implémentée par la même fonction*/
    pthread_join(tache1,NULL); /* Le programme principal s'attache à
la tâche 1. Il ne se terminera que lorsque cette tâche se terminera */
}
```

Cet exemple montre le lancement de deux tâches implémentées par la même fonction. Il illustre le fonctionnement en mode attaché/détaché de la façon suivante : seule la tâche 1 est attachée au programme principal. Dès qu'elle se termine, et que le programme principal a la main, celui-ci arrive sur la dernière instruction et se termine avant que la tâche 2, créée au moins deux secondes après la tâche 1, n'ait le temps de se terminer. Le résultat de cette exécution peut être :

```
Tache 1 : iteration 0
Tache 1 : iteration 1
Tache 2 : iteration 0
Tache 1 : iteration 2
Tache 2 : iteration 1
Tache 1 : iteration 3
Tache 2 : iteration 2
Tache 1 : iteration 4
Tache 2 : iteration 3
```

Après la terminaison de la tâche 1, la tâche 2 obtient la main avant le programme principal et a le temps d'effectuer une 4e itération avant la terminaison du programme principal.
Nous pouvons noter que toutes les fonctions manipulant les tâches commencent par `pthread_`.

Si l'on souhaite qu'une tâche fonctionne de façon spécifique, il faut lui donner des attributs spécifiques. Ainsi, dans le code ci-après, la tâche 1 se voit ordonnancée (si le système le supporte) de façon globale, alors que la tâche 2 l'est de façon locale. On obtient donc un ordonnancement mixte.

```
    pthread_attr_t attributs;
    /* Attribut utilisé lors de la création des tâches */

    pthread_attr_init(&attributs);
    /*Initialisation des attributs aux valeurs par défaut*/
    pthread_attr_setscope(&attributs, PTHREAD_SCOPE_SYSTEM);
    /* L'ordonnancement des tâches créées avec cet attribut s'effectue
au niveau global */
    pthread_create(&tache1, &attributs,f, (void *)1);
    /* La tâche 1 est ordonnancée au niveau global */
    usleep(2000000);
    pthread_attr_setscope(&attributs, PTHREAD_SCOPE_PROCESS);
    /* L'ordonnancement des tâches créées avec cet attribut s'effectue
au niveau local */
    pthread_create(&tache2, &attributs,f, (void *)2);
    /*Création de la seconde tâche, ordonnancée localement*/
    pthread_attr_destroy(&attributs);
    pthread_join(tache1,NULL);
    /* Le programme principal s'attache à la tâche 1. Il ne poursuivra
son exécution jusqu'à sa terminaison que lorsque cette tâche se
terminera */
```

Nous pouvons noter que toutes les fonctions manipulant les attributs de tâches commencent par `pthread_attr` suivi du nom de l'attribut à modifier. Il est possible de modifier (respectivement interroger) la plupart des attributs d'une tâche pendant son exécution avec des fonctions dont le préfixe est `pthread_set_` (respectivement `pthread_get_`).
Il faut souligner que plusieurs compilateurs présents sur des systèmes d'exploitation généralistes proposent une interface POSIX 1003.1c, mais que celle-ci est malheureusement souvent incomplète (impossibilité d'attacher une tâche, gestion des priorités et de l'ordonnancement absents). Par conséquent, il convient de tester les valeurs de retour des fonctions POSIX afin d'y déceler des erreurs (notamment « fonctionnalité non implémentée »).

□ Ordonnancement et priorités

Les attributs de tâches permettent de leur affecter une priorité dès leur lancement. Il est aussi possible de modifier la priorité d'une tâche au cours de son exécution grâce à la fonction `pthread_set_schedparam`.

```
#include <pthread.h>
void f(int no) {
    int i;
    for (i=0;i<5;i++) {
        printf("Tache %d : iteration %d\n",no,i);
        usleep(1000000);
    }
}

void main() {
```

© Dunod – La photocopie non autorisée est un délit.

```
    pthread_t taches[5]; /* Tableau d'identificateurs de tâches */
    struct sched_param schedparam; /* Paramètre servant à affecter une
priorité aux tâches */
    pthread_attr_t attributs; /* Attributs utilisés lors de la
création des tâches */
    int i;

    schedparam.sched_priority = 30;
    pthread_setschedparam(pthread_self(),SCHED_RR,&schedparam); /*
Modifie l'ordonnancement du programme principal, et passe sa priorité
à 30 */
    pthread_attr_init(&attributs); /*Initialisation des attributs aux
valeurs par défaut*/
   pthread_attr_setinheritsched(&attributs, PTHREAD_EXPLICIT_SCHED);
/* Force la prise en compte des paramètres d'ordonnancement de
l'attribut: sans cela, ils seraient hérités du programme principal */
    pthread_attr_setschedpolicy(&attributs, SCHED_RR); /*
Ordonnancement à tourniquets par niveaux de priorités  */
    for (i=0;i<5;i++) {
        schedparam.sched_priority = 2+i; /* Priorité de la prochaine
tâche à créer */
        pthread_attr_setschedparam(&attributs, &schedparam); /* La
priorité est placée dans les attributs */
        pthread_create(&(taches[i]), &attributs,f, (void *)i);
    }
    pthread_attr_destroy(&attributs);
    for (i=0;i<5;i++) {
        pthread_join(taches[i],NULL); /* Attente de terminaison des
tâches */
    }
}
```

☐ Modules de données

Les modules de données DARTS peuvent être implémentés à l'aide d'un sémaphore d'exclusion mutuelle (*mutex*). Ainsi, une partie du code correspondant à la figure 6.24 est donnée ci-après.

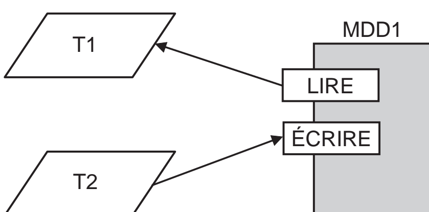


Figure 6.24 – Module de données DARTS.

```
#include <pthread.h>

/* Type utilisé pour le module de données */
typedef struct {
    float Temperature;
    float Pression;
} t_Temperature_Pression;
```

```
t_Temperature_Pression MDD1; /*Déclaration du module de données MDD1*/
pthread_mutex_t s_MDD1; /* Mutex protégeant MDD1 */

void T1() {
/* Tâche 1 lisant le module de données */
    t_Temperature_Pression TP;
    while (1) { /* Faire toujours */
        /* Exclusion mutuelle sur le module de données */
        pthread_mutex_lock(&s_MDD1);
        /* Lecture du MDD */
        TP=MDD1;
        pthread_mutex_unlock(&s_MDD1);
        /* etc. */
    }
}

void T2() {
/* Tâche 2 modifiant le module de données */
    float T,P;
    while (1) { /* Faire toujours */
        /* Lecture des capteurs en dehors de la section critique */
        P=Lire_Capteur_Pression();
        T=Lire_Capteur_Temperature();
        /* Exclusion mutuelle sur le module de données */
        pthread_mutex_lock(&s_MDD1);
        /* Modification du MDD */
        MDD1.Temperature=T;
        MDD1.Pression=P;
        pthread_mutex_unlock(&s_MDD1);
        /* etc. */
    }
}

void main() {
  pthread_t t1,t2; /* Identificateurs de tâches */
  pthread_mutex_init(&s_MDD1,NULL); /*Initialisation du mutex de MDD1
*/
  /* Initialisation de MDD1 */
  MDD1.Temperature=Lire_Capteur_Temperature();
  MDD1.Pression=Lire_Capteur_Pression();
  /* Lancement des tâches */
  pthread_create(&t1, NULL,T1,NULL);
  pthread_create(&t2, NULL,T2,NULL);
  /* Attente de terminaison */
  pthread_join(t1,NULL);
  pthread_join(t2,NULL);
}
```

On peut remarquer que le sémaphore est déclaré de façon globale, ce qui le rend visible aux deux tâches concernées. Notons qu'il est créé et initialisé avant le lancement des tâches, par exemple dans le programme principal.
Il est nécessaire de réduire la durée des sections critiques, par conséquent, sur cet exemple, l'accès au matériel via les fonctions de lecture de capteurs, très longues, doit s'effectuer à l'extérieur de la section critique d'accès au module de données.
Si l'on souhaite s'occuper des priorités, afin de pouvoir effectuer une étude d'ordonnançabilité, il convient de définir des priorités pour les tâches, de définir une prio-

© Dunod – La photocopie non autorisée est un délit.

rité plafond pour le sémaphore (la plus forte des priorités des tâches l'utilisant), et de lui affecter le protocole à priorité plafond.
Ainsi, le programme principal devient :

```
void main() {
  pthread_t t1,t2; /* Identificateurs de tâches */
  pthread_attr_t task_attr; /* Attributs de tâches */
  pthread_mutexattr_t mutex_attr; /* Attributs de mutex */
  struct sched_param schedparam; /* Paramètre servant à affecter une
priorité aux tâches */

    pthread_mutexattr_init(&mutex_attr); /* Initialisation des
attributs du mutex */
  pthread_mutexattr_setprioceiling(&mutex_attr,40); /* Initialisation
de la priorité plafond du mutex à max_priorité(t1,t2) */
  pthread_mutexattr_setprotocol(&mutex_attr,PTHREAD_PRIO_PROTECT); /*
Utilisation du protocole à priorité plafond. Ce protocole est
malheureusement optionnel dans les implémentations POSIX */

  pthread_mutex_init(&s_MDD1,&mutex_attr); /* Initialisation du mutex
de MDD1 */
  MDD1.Temperature=Lire_Capteur_Temperature();/* Initialisation de
MDD1 */
  MDD1.Pression=Lire_Capteur_Pression();
  schedparam.sched_priority = 41;
  pthread_setschedparam(pthread_self(),SCHED_RR,&schedparam);  /*
Modifie l'ordonnancement du programme principal, et passe sa priorité
à 41 */
  pthread_attr_init(&task_attr); /*Initialisation des attributs aux
valeurs par défaut*/
  pthread_attr_setinheritsched(&task_attr, PTHREAD_EXPLICIT_SCHED); /
* Force la prise en compte des paramètres d'ordonnancement de
l'attribut: sans cela, ils seraient hérités du programme principal */
  pthread_attr_setschedpolicy(&task_attr, SCHED_RR); /*
Ordonnancement à tourniquets par niveaux de priorités  */
  schedparam.sched_priority = 30; /* Priorité de t1 */
  pthread_attr_setschedparam(&task_attr, &schedparam); /* La priorité
de t1 est placée dans les attributs */
  pthread_create(&t1, &task_attr,T1,NULL); /* Lancement de t1 */
  schedparam.sched_priority = 40; /* Priorité de t2 */
  pthread_attr_setschedparam(&task_attr, &schedparam); /* La priorité
de t2 est placée dans les attributs */
  pthread_create(&t2, NULL,T2,NULL); /* Lancement de t1 */

  /* Libération de la mémoire utilisée par les attributs */
  pthread_mutexattr_destroy(&mutex_attr);
  pthread_attr_destroy(&task_attr);
  pthread_join(t1,NULL);
  pthread_join(t2,NULL);
}
```

Nous voyons ici l'un des inconvénients de POSIX : la puissance et la souplesse de la norme impliquent un effort de code important pour arriver à une solution intégrant toutes les contraintes désirées.
Il serait aussi possible de protéger les modules de données grâce à un verrou de type *rwlock* (lecteur/écrivain) lorsqu'il y a plusieurs lecteurs. Cependant, cet outil ne permet pas l'utilisation du protocole à priorité plafond.

Boîtes aux lettres

POSIX ne définit malheureusement aucun outil de communication par message spécifique aux tâches. L'amendement 1003.1b, orienté processus, définit quant à lui les boîtes aux lettres nommées permettant à deux processus de communiquer. Étant donné leur champ d'application, leur utilisation est lourde. De plus, cet outil ne propose que des boîtes aux lettres sans écrasement, ce qui n'est pas suffisant pour implémenter tous les éléments de communication par message de la méthode DARTS. Par conséquent, afin d'implémenter une communication par messages au niveau tâches uniquement, nous définissons ici un module, nommé *BaLs*, fournissant différents types de boîtes aux lettres utilisant les mutex et les variables conditionnelles (voir le concept au § 5.2.2, p. 189). L'utilisation de ce couple d'outils permet en effet de définir des moniteurs, à partir desquels tout outil de communication et/ou de synchronisation peut être créé.

Boîtes aux lettres de taille 1 avec écrasement

Le premier type de boîte aux lettres implémenté correspond aux boîtes aux lettres de taille 1 avec écrasement, que nous nommerons `bal_ecr`.
La spécification de ce type abstrait est :

```
/* Boîte aux lettres avec écrasement de taille 1 (i.e. réception
bloquante, envoi non bloquant) */
typedef struct s_bal_ecr {
      char * buf; /* Zone dans laquelle le message est stocké, mémoire
allouée DYNAMIQUEMENT par bal_ecr_init */
      unsigned char vide;/* booléen permettant de savoir si la boîte
est vide ou pleine */
      pthread_mutex_t mutex;/* Sémaphore d'exclusion mutuelle
garantissant qu'une seule tâche manipule la structure */
      pthread_cond_t pas_vide;/* Variable conditionnelle utilisée
pour signaler un ajout d'élément susceptible de réveiller une tâche en
attente */
      unsigned taille_element;/* taille d'un élément de boîte aux
lettres */
} *bal_ecr;

bal_ecr bal_ecr_init(const unsigned taille_element);
/* Crée et initialise une boîte aux lettres à écrasement vide
pouvant contenir un élément de taille_element octets
Nécessite: taille_element>0
Renvoie: la BaL si succès, 0 sinon */
int bal_ecr_recevoir(bal_ecr bal, char *buf);
/* Attend un message et reçoit un message de la BaL b
Primitive bloquante
Nécessite: b initialisée par bal_ecr_init
           taille(buf) >= taille_element donné lors de bal_ecr_init
Entraine: buf contient le message reçu
Renvoie: taille_element */
int bal_ecr_envoyer(bal_ecr bal, const char *buf);
/* Envoie un message dans la BaL b en écrasant si besoin le message
présent
Primitive non bloquante
Nécessite: b initialisée par bal_ecr_init
           taille(buf) >= taille_element donné lors de bal_ecr_init
Entraine: la boîte aux lettres contient le message buf
```

© Dunod – La photocopie non autorisée est un délit.

```
Renvoie: taille_element */
void bal_ecr_delete(bal_ecr bal);
/* Supprime la BaL b
Nécessite: b initialisée par bal_ecr_init
Entraine: b est supprimée */
```

L'implémentation des primitives manipulant les boîtes aux lettres à écrasement est donnée ci-après :

```
bal_ecr bal_ecr_init(const unsigned taille_element) {
    bal_ecr bal;
      if (!(bal=(bal_ecr)malloc(sizeof(struct s_bal_ecr)))) return
0;/* Allocation de la structure */
      if (!((*bal).buf=(char *)malloc(taille_element))) return 0;/*
Allocation du buffer contenant un message */
      (*bal).vide=1;/* Initialement la bal est vide */
pthread_mutex_init(&((*bal).mutex),0);/* Création du sémaphore
garantissant l'exclusion mutuelle des accès à la structure */
      (*bal).taille_element=taille_element;
pthread_cond_init(&((*bal).pas_vide), 0);/* Variable conditionnelle
qui sera déclenchée lorsqu'un message est ajouté */
      return bal;
}

int bal_ecr_recevoir(bal_ecr bal, char *buf) {
    pthread_mutex_lock (&(*bal).mutex);
    /* Exclusion mutuelle */
      while ((*bal).vide) {
      /* Tant que la boîte aux lettres est vide */
            pthread_cond_wait (&(*bal).pas_vide, &(*bal).mutex);/* On
attend que la bal contienne un message */
      }
      memcpy(buf,(*bal).buf,(*bal).taille_element);/* Copie du
message dans buf */
      (*bal).vide=1;/* La boîte est maintenant vide */
      pthread_mutex_unlock (&(*bal).mutex);/* Fin de l'exclusion
mutuelle */
      return (*bal).taille_element;
}

int bal_ecr_envoyer(bal_ecr bal, const char *buf) {
    pthread_mutex_lock (&(*bal).mutex);
    /* Exclusion mutuelle */
      memcpy((*bal).buf,buf,(*bal).taille_element);/* Le message
écrase un éventuel message présent dans la bal */
      (*bal).vide=0;/* La boîte contient un message */
      pthread_mutex_unlock (&(*bal).mutex);/* Fin de l'exclusion
mutuelle */
      pthread_cond_signal (&(*bal).pas_vide);/* On réveille
l'éventuel thread en attente d'un message */
      return (*bal).taille_element;
}

void bal_ecr_delete(bal_ecr bal) {
    pthread_cond_destroy(&(*bal).pas_vide);
    pthread_mutex_lock(&(*bal).mutex);
    free((*bal).buf);
    pthread_mutex_unlock(&(*bal).mutex);
    free(bal);
}
```

Notons la façon dont on utilise la variable conditionnelle : une tâche se met en attente en appelant `pthread_cond_wait(&(*bal).pas_vide, &(*bal).mutex)`. Cela a pour effet de libérer le mutex, et d'attendre un signal sur la variable conditionnelle `&(*bal).pas_vide`. La tâche déposant un message signale cette variable afin de réveiller les tâches en attente, qui peuvent alors vérifier si un message est présent. Lorsqu'au maximum une tâche est en attente, le signal de la variable s'effectue avec `pthread_cond_signal`, alors que lorsque plusieurs tâches sont susceptibles d'être en attente, le signal s'effectue à l'aide de la fonction `pthread_cond_broadcast`.

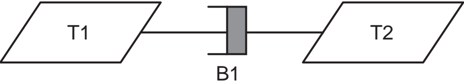


Figure 6.25 – Communication par boîte aux lettres de taille 1 avec écrasement.

L'implémentation partielle du schéma DARTS donné sur la figure 6.25 est donc :

```
#include <pthread.h>
/* Inclusion du module de boîtes aux lettres */
#include "BaLs.h"

/* Type utilisé pour la boîte aux lettres */
typedef struct {
    float Temperature;
    float Pression;
} t_Temperature_Pression;

bal_ecr B1; /* Déclaration de la boîte aux lettres */

void T1() {
/* Tâche envoyant des messages dans B1 */
    t_Temperature_Pression TP;
    while (1) { /* Faire toujours */
        /* Lecture des capteurs en dehors de la section critique */
        TP.Pression=Lire_Capteur_Pression();
        TP.Temperature=Lire_Capteur_Temperature();
        bal_ecr_envoyer(B1,(char*)&TP); /* Envoi du message */
       /* Noter la perte de l'information de type lors du passage dans
la boîte aux lettres */
        /* etc. */
    }
}

void T2() {
/* Tâche en attente de messages dans B1 */
    t_Temperature_Pression TP;
    while (1) { /* Faire toujours */
        bal_ecr_recevoir(B1,(char*)&TP); /* Attente du message */
        /* Noter la coercion dans le type attendu */
        /* etc. */
    }
}

void main() {
```

© Dunod – La photocopie non autorisée est un délit.

```
  pthread_t t1,t2; /* Identificateurs de tâches */
  B1=bal_ecr_init(sizeof(t_Temperature_Pression));/* Initialisation
de B1 */
  /* Lancement des tâches */
  pthread_create(&t1, NULL,T1,NULL);
  pthread_create(&t2, NULL,T2,NULL);
  /* Attente de terminaison */
  pthread_join(t1,NULL);
  pthread_join(t2,NULL);
}
```

Boîtes aux lettres de taille 1 sans écrasement

Pour les boîtes aux lettres sans écrasement (figure 6.26), la démarche adoptée est similaire à celle utilisée pour les boîtes aux lettres avec écrasement. La différence vient du fait que dans ce cas, les tâches émettrices peuvent être bloquées. On utilise donc deux variables conditionnelles : l'une dédiée à signaler l'arrivée de message (`pas_vide`), l'autre dédiée au signalement de suppression de message (`pas_plein`). L'implémentation est de type producteur/consommateur.

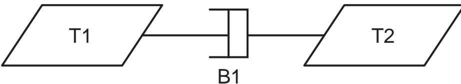


Figure 6.26 – Communication par boîte aux lettres de taille 1 sans écrasement.

```
/* Boîte aux lettres de taille 1 sans écrasement (i.e. envoi et
réception bloquants) */
typedef struct s_bal {
      char * buf;/* Zone dans laquelle le message est stocké, mémoire
allouée DYNAMIQUEMENT par bal_init */
      unsigned char vide;/* booléen permettant de savoir si la boîte
est vide ou pleine */
      pthread_mutex_t mutex;/* Sémaphore d'exclusion mutuelle
garantissant qu'une seule tâche manipule la structure */
      pthread_cond_t pas_plein,pas_vide;/* Variables conditionnelles
utilisées pour signaler un ajout/retrait d'élément susceptible de
réveiller une tâche en attente */
      unsigned taille_element;/* taille d'un élément de boîte aux
lettres */
} *bal;

bal bal_init(const unsigned taille_element);
/* Crée et initialise une boîte aux lettres vide
pouvant contenir un élément de taille_element octets
Nécessite: taille_element>0
Renvoie: la BaL si succès, 0 sinon */
int bal_recevoir(bal b, char *buf);
/* Attend un message et reçoit un message de la BaL b
Primitive bloquante
Nécessite: b initialisée par bal_init
           taille(buf) >= taille_element donné lors de bal_init
Entraine: buf contient le message reçu
Renvoie: taille_element */
int bal_envoyer(bal b, const char *buf);
```

```
/* Envoie un message dans la BaL b (peut attendre que la BaL soit
vide)
Primitive bloquante
Nécessite: b initialisée par bal_init
           taille(buf) >= taille_element donné lors de bal_init
Entraine: la boîte aux lettres contient le message buf
Renvoie: taille_element */
void bal_delete(bal b);
/* Supprime la BaL b
Nécessite: b initialisée par bal_init
Entraine: b est supprimée */
```

Le corps de ce module est donné ci-après :

```
bal bal_init(const unsigned taille_element) {
    bal b;
      if (!(b=(bal)malloc(sizeof(struct s_bal)))) return 0; /*
Allocation de la structure */
      if (!((*b).buf=(char *)malloc(taille_element))) return 0; /*
Allocation du buffer contenant un message */
(*b).vide=1; /* Initialement la bal est vide */
      pthread_mutex_init(&((*b).mutex),0); /* Création du sémaphore
garantissant l'exclusion mutuelle des accès à la structure */
      (*b).taille_element=taille_element;
      pthread_cond_init (&((*b).pas_vide), 0); /* Variable
conditionnelle qui sera déclenchée lorsqu'un message est ajouté */
      pthread_cond_init (&((*b).pas_plein), 0); /* Variable
conditionnelle qui sera déclenchée lorsque la bal est vidée */
      return b;
}
int bal_recevoir(bal b, char *buf) {
    pthread_mutex_lock (&(*b).mutex);
    /* Exclusion mutuelle */
      while ((*b).vide) {
            pthread_cond_wait (&(*b).pas_vide, &(*b).mutex);
            /* On attend que la bal contienne un message */
      }
      memcpy(buf,(*b).buf,(*b).taille_element); /* Copie du message
dans buf */
      (*b).vide=1; /* La boîte est maintenant vide */
      pthread_mutex_unlock (&(*b).mutex); /* Fin de l'exclusion
mutuelle */
      pthread_cond_broadcast (&(*b).pas_plein); /* On réveille
d'éventuelles tâches en attente sur la boîte pleine */
      return (*b).taille_element;
}
int bal_envoyer(bal b, const char *buf) {
    pthread_mutex_lock (&(*b).mutex);
    /* Exclusion mutuelle */
      while (!(*b).vide) {
            pthread_cond_wait (&(*b).pas_plein, &(*b).mutex);
            /* On attend que la bal soit vide */
      }
      memcpy((*b).buf,buf,(*b).taille_element); /* On copie le
message dans la bal */
      (*b).vide=0; /* La boîte contient un message */
      pthread_mutex_unlock (&(*b).mutex); /* Fin de l'exclusion
mutuelle */
      pthread_cond_signal (&(*b).pas_vide); /* On réveille
l'éventuelle tâche en attente d'un message */
```

© Dunod – La photocopie non autorisée est un délit.

```
    return (*b).taille_element;
}

void bal_delete(bal b) {
    pthread_cond_destroy(&(*b).pas_vide);
    pthread_cond_destroy(&(*b).pas_plein);
    pthread_mutex_lock(&(*b).mutex);
    free((*b).buf);
    pthread_mutex_unlock(&(*b).mutex);
    free(b);
}
```

L'implémentation de tâches communiquant par boîtes aux lettres est presque identique à celle présentée pour l'utilisation de boîtes aux lettres avec écrasement. La seule différence réside dans les noms de fonctions utilisés. Notons qu'il aurait été possible de **surcharger** les fonctions, c'est-à-dire de créer des fonctions de même nom (`bal_init`, `bal_envoyer`, `bal_recevoir` et `bal_delete`) manipulant des types de boîtes aux lettres différents. Il en a été décidé autrement afin de faciliter la lisibilité du code, et parce que l'implémentation suit normalement la conception, pendant laquelle le type de boîte aux lettres utilisé a été choisi.

Autres boîtes aux lettres

Le lecteur trouvera en annexe C l'extension des boîtes aux lettres de taille 1 avec et sans écrasement au cas des boîtes aux lettres de taille *n* (figure 6.27). La file de messages est implémentée par un tableau circulaire. La prise en compte d'une priorité de messages peut s'effectuer de deux façons : soit le nombre de priorités est élevé, dans ce cas la gestion de la file d'attente consiste en un tri par insertion des messages, ce qui est coûteux, soit il n'y a que deux niveaux de priorité (normal et urgent,) et on peut créer deux files de messages.

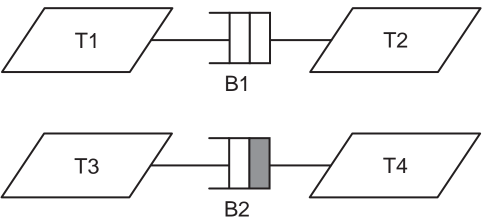


Figure 6.27 – Communication par boîte aux lettres de taille *n*.

☐ Synchronisation

Généralement, on implémente la synchronisation (figure 6.28) à l'aide de sémaphores à compte (§ 5.2.2, p. 201). Cependant, nous avons vu au chapitre 6 que les amendements 1003.1c et 1003.1j ne proposent pas cet outil de synchronisation. Les sémaphores à compte sont définis dans la partie 1003.1b orientée processus. Cependant, il est possible, à la création d'un sémaphore à compte 1003.1b, de forcer celui-ci à ne pas être partagé par plusieurs processus.

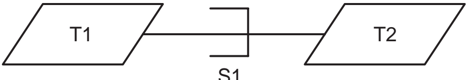


Figure 6.28 – Synchronisation.

```
#include <pthread.h>
#include <semaphore.h>

sem_t S1; /* Sémaphore de synchronisation */

void T1() {
/* Tâche déclenchant une synchronisation S1 */
    while (1) { /* Faire toujours */
        /*etc. */
        sem_post(&S1); /* Déclenchement de la synchronisation */
        /* etc. */
    }
}

void T2() {
/* Tâche en attente sur la synchronisation S1 */
    while (1) { /* Faire toujours */
        sem_wait(&S1); /* Attente de synchronisation */
        /* etc. */
    }
}

void main() {
  pthread_t t1,t2; /* Identificateurs de tâches */
  sem_init(&S1,0,0); /* Sémaphore non partagé par des processus
initialisés à 0 */
  /* Lancement des tâches */
  pthread_create(&t1, NULL,T1,NULL);
  pthread_create(&t2, NULL,T2,NULL);
  /* Attente de terminaison */
  pthread_join(t1,NULL);
  pthread_join(t2,NULL);
}
```

☐ **Tâches périodiques**

Il n'est pas trivial de rendre une tâche périodique en POSIX. Certains systèmes proposent des primitives spécifiques non POSIX (suffixe `_np` au nom de la fonction) permettant de gérer la périodicité stricte des tâches. Ainsi, RTLinux fournit une fonction `pthread_make_periodic_np` permettant de rendre une tâche périodique. Cependant, la non portabilité de cette fonction est gênante. Il existe aussi des solutions lourdes, basées sur la programmation d'horloges générant des signaux récupérés par les tâches périodiques. Le problème est que différentes implémentations partielles de POSIX ne les supportent pas. Une autre solution, non satisfaisante, consisterait à endormir une tâche pendant la durée de sa période (fonctions `nanosleep` ou `usleep`), mais elle expose la tâche au problème de la dérive des horloges (§ 5.2.4). La solution retenue ici, fonctionnant sur la plupart des implémentations POSIX, même partielles, consiste à utiliser la fonction `pthread_cond_timedwait`, attente

© Dunod – La photocopie non autorisée est un délit.

bornée sur une variable conditionnelle, car cette fonction utilise une date absolue de *timeout*. L'idée est de créer une variable conditionnelle jamais signalée dans chaque tâche périodique : l'attente de la prochaine date d'activation s'effectue par le biais du *timeout* de `pthread_cond_timedwait`.
Ainsi, l'implémentation des tâches de la figure 6.29 est donnée ci-après.

Figure 6.29 – Deux tâches périodiques.

```
#include <pthread.h>

void ajouter_microsecondes(struct timespec *time, long us) {
/* Modifie la structure time afin d'y ajouter us microsecondes */
    (*time).tv_nsec+=(us%1000000)*1000; /* Ajout des microsecondes au
champ en nanosecondes */
    (*time).tv_sec+=(us/1000000)+((*time).tv_nsec/1000000000); /*
Ajout des secondes entières au champ en secondes */
    (*time).tv_nsec%=1000000000; /* Si il y avait débordement des
nanosecondes sur les secondes, il a été ajouté aux secondes dans
l'instruction précédente */
}

void Periodique(long periode_us) {
/* Tâche périodique affichant sa période à chaque période */
/* periode_us : période en microsecondes */
    struct timespec horloge; /* consituée d'un champ en secondes, et
d'un autre en nanosecondes */
    pthread_cond_t Reveil; /* Variable conditionnelle utilisée par les
tâches périodiques afin de se réveiller sur timeout */
    /* Cette variable n'est jamais signalée */
    pthread_mutex_t sReveil; /* Mutex lié à la variable conditionnelle
*/
    pthread_mutex_init(&sReveil,NULL); /* Initialisation du mutex */
    pthread_cond_init(&Reveil,NULL); /* Initialisation de la variable
conditionnelle */
    clock_gettime(CLOCK_REALTIME, &horloge); /* heure courante de
l'horloge au démarrage de la tâche */
    while (1) {
        printf("Periode %d\n",periode_us);

        ajouter_microsecondes(&horloge,periode_us);/* Calcul de la
date du prochain reveil */
        pthread_mutex_lock(&sReveil);
        pthread_cond_timedwait(&Reveil, &sReveil, &horloge);
        /* Cette variable n'étant pas signalée, c'est au timeout que
cette instruction se termine */
    }
}
```

```
void main() {
    pthread_t periodique1,periodique2;
    pthread_create(&periodique1, NULL,Periodique, (void*)1000); /*
Lancement de la tâche avec une période d'1 ms */
    pthread_create(&periodique2, NULL,Periodique, (void*)1500); /*
Lancement de la tâche avec une période d'1,5 ms */
    pthread_join(periodique1,NULL); /* Attente de terminaison */
    pthread_join(periodique2,NULL); /* Attente de terminaison */
}
```

Une solution proche, plus simple, consiste à utiliser les sémaphores de l'amendement 1003.1, qui intègrent aussi une primitive d'attente bornée dans le temps par une date. L'implémentation de la figure 6.29 deviendrait alors :

```
#include <pthread.h>
#include <semaphore.h>

void ajouter_microsecondes(struct timespec *time, long us) {
/* Modifie la structure time afin d'y ajouter us microsecondes */
    (*time).tv_nsec+=(us%1000000)*1000; /* Ajout des microsecondes au
champ en nanosecondes */
    (*time).tv_sec+=(us/1000000)+((*time).tv_nsec/1000000000); /*
Ajout des secondes entières au champ en secondes */
    (*time).tv_nsec%=1000000000; /* Si il y avait débordement des
nanosecondes sur les secondes, il a été ajouté aux secondes dans
l'instruction précédente */
}

void Periodique(long periode_us) {
/* Tâche périodique affichant sa période à chaque période */
/* periode_us : période en microsecondes */
    struct timespec horloge; /* constituée d'un champ en secondes, et
d'un autre en nanosecondes */
    sem_t Reveil;
    sem_init(&Reveil,0,0); /* Création d'un sémaphore non partagé
entre processus initialisé à 0 */
    clock_gettime(CLOCK_REALTIME, &horloge); /* heure courante de
l'horloge au démarrage de la tâche */
    while (1) {
        printf("Periode %d\n",periode_us);

        ajouter_microsecondes(&horloge,periode_us);/* Calcul de la
date du prochain réveil */
        sem_timedwait(&Reveil,&horloge); /* Le sémaphore étant
toujours nul, cette instruction a pour effet d'endormir la tâche
jusqu'à sa prochaine date de réveil */
    }
}

void main() {
    pthread_t periodique1,periodique2;
    pthread_create(&periodique1, NULL,Periodique, (void*)1000); /*
Lancement de la tâche avec une période d'1 ms */
    pthread_create(&periodique2, NULL,Periodique, (void*)1500); /*
Lancement de la tâche avec une période d'1,5 ms */
    pthread_join(periodique1,NULL); /* Attente de terminaison */
    pthread_join(periodique2,NULL); /* Attente de terminaison */
}
```

© Dunod – La photocopie non autorisée est un délit.

La seconde solution proposée présente l'avantage de nécessiter moins d'outils (sémaphore contre mutex et variable conditionnelle), et sera choisie si l'interface POSIX utilisée implémente la primitive `sem_timedwait`.

□ Tâches réveillées par interruption

La gestion des interruptions étant très liée à l'architecture sous-jacente, et à l'exécutif ou au système d'exploitation utilisé, POSIX ne définit pas d'interface normalisée pour la gestion des interruptions. Cela signifie que la programmation d'une routine de traitement d'interruption est liée au système sous-jacent. Si le traitement de l'interruption doit être effectué par une tâche, alors on utilise la technique du traitement différé (DSR, voir § 5.2.3, p. 206) : un code très court, appelé ISR est lié à l'interruption traitée. Son action consiste à signaler par synchronisation (§ 6.2.1, p. 324) à la tâche de traitement l'occurrence de l'interruption.

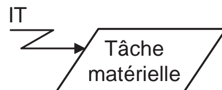


Figure 6.30 – Tâche matérielle.

Ainsi, le mécanisme de réveil de la tâche matérielle présentée sur la figure 6.30 s'implémente simplement par synchronisation sur sémaphore.

```
#include <pthread.h>
#include <semaphore.h>

sem_t declenche_dsr; /* Sémaphore de synchronisation entre l'ISR et la
tâche de traitement */
void mon_isr(/* Paramètres spécifiques au système */) {
/* Fonction appelée sur l'occurrence de l'interruption traitée */
/* Appels spécifiques permettant d'acquiescer l'interruption */
      sem_post(&declenche_dsr); /* Déclenche la tâche de traitement */
}

void Tache_Materielle() {
/* Tâche déclenchée sur interruption */
    while (1) {
        sem_wait(&declenche_dsr); /* Attente d'interruption */
/* Traitement d'une interruption */
/* etc. */
    }
}

void main() {
    pthread_t tache_materielle;
    sem_init(declenche_dsr,0,0); /* Initialisation du sémaphore de
synchronisation ISR-]-DSR */
/* Appel d'une primitive système branchant l'ISR à l'interruption
traitée */
/* Démasquage de l'interruption traitée */
    pthread_create(&tache_materielle, NULL,Tache_Materielle, NULL);
    pthread_join(tache_materielle,NULL); /* Attente de terminaison */
}
```

■ Exemple

La mise en œuvre des différents outils présentés pour l'implémentation d'éléments DARTS en POSIX 1003.1 est faite sur l'exemple de la « gestion de la sécurité d'une mine » dont le diagramme DARTS est donné sur la figure 3.27. Afin d'en faciliter la lecture, le système de commande est présenté à plat, sans utilisation de modules spécifiques, autres que ceux liés à la communication par boîte aux lettres. Pour la même raison, les attributs de tâches (§ 6.2.1, p. 287) permettant de jouer sur l'ordonnancement du système, et les attributs de mutex (§ 6.2.1, p. 288) permettant l'utilisation du protocole à priorité plafond sont omis.

```
#include <pthread.h>
#include <semaphore.h>
#include "BaLs.h" /* Module de boites aux lettres */
#include "procede.h" /* Fonctions d'accès aux capteurs et actionneurs
*/

/* Définition de constantes */
#define LLS 5.3
/* Valeur sous laquelle le niveau est considéré trop bas */
#define HLS 17.8
/* Valeur au-dessus de laquelle le niveau est considéré trop haut */
#define MS_L1 128
#define MS_L2 180
/* Seuils d'alerte de méthane*/

/* Définition des outils de communication/synchronisation */
/* Module de données Niveau_Eau */
float Niveau_Eau; /* Module de données */
pthread_mutex_t sNiveau_Eau; /* Mutex du module de données */

bal_ecr Niveau_Methane; /* Boîte aux lettres de taille 1 à écrasement
*/
bal_ecr Vitesse_Pompe; /* Boîte aux lettres de taille 1 à écrasement
*/
sem_t Evt_Alarme; /* Synchronisation */

void ajouter_microsecondes(struct timespec *time, long us) {
/* Modifie la structure time afin d'y ajouter us microsecondes */
    (*time).tv_nsec+=(us%1000000)*1000; /* Ajout des microsecondes au
champ en nanosecondes */
    (*time).tv_sec+=(us/1000000)+((*time).tv_nsec/1000000000); /*
Ajout des secondes entières au champ en secondes */
    (*time).tv_nsec%=1000000000; /* Si il y avait débordement des
nanosecondes sur les secondes, il a été ajouté aux secondes dans
l'instruction précédente */
}

void Acquerir_Niveau_Methane() {
/* Tâche périodique d'acquisition */
    struct timespec horloge; /* Date du prochain réveil */
    sem_t Reveil; /* Sémaphore utilisé pour le réveil périodique */
    float niveau_methane;
    sem_init(&Reveil,0,0);
    clock_gettime(CLOCK_REALTIME, &horloge); /* heure courante de
l'horloge au démarrage de la tâche */
    while (1) {
```

© Dunod – La photocopie non autorisée est un délit.

```
        niveau_methane=Lire_Capteur_Methane(); /* Lecture du capteur
*/
        bal_ecr_envoyer(Niveau_Methane,(char*)&niveau_methane);/*
Envoi du message */

        /* Périodicité */
        ajouter_microsecondes(&horloge,500000);/* Calcul de la
prochaine date de réveil (période de 500 ms) */
        sem_timedwait(&Reveil,&horloge); /* Attente de la prochaine
date de réveil */
    }
}

void Acquerir_Capteur_Eau() {
/* Tâche périodique d'acquisition */
    struct timespec horloge; /* Date du prochain réveil */
    sem_t Reveil; /* Sémaphore utilisé pour le réveil périodique */
    float niveau_eau;
    sem_init(&Reveil,0,0);
    clock_gettime(CLOCK_REALTIME, &horloge); /* heure courante de
l'horloge au démarrage de la tâche */
    while (1) {
        niveau_eau=Lire_Capteur_Eau(); /* Lecture du capteur */

        /* Ecriture dans le module de données */
        pthread_mutex_lock(&sNiveau_Eau);
        Niveau_Eau=niveau_eau; /* Noter que l'on n'appelle pas
l'acquisition dans la section critique afin de minimiser la durée de
celle-ci */
        pthread_mutex_unlock(&sNiveau_Eau);

        /* Périodicité */
        ajouter_microsecondes(&horloge,5000000);/* Calcul de la
prochaine date de réveil (période de 5 s) */
        sem_timedwait(&Reveil,&horloge); /* Attente de la prochaine
date de réveil */
    }
}

void Afficher_Alarme () {
    unsigned char etat_alarme=0; /* Etat courant de l'alarme (0
éteinte, 1 allumée) */
    while (1) {
        sem_wait(&Evt_Alarme); /* Attente de synchronisation */
        etat_alarme=1-etat_alarme;
        Piloter_Alarme(etat_alarme); /* Fonction actionnant ou coupant
l'alarme */
    }
}

void Commander_Pompe() {
    float vitesse;
    while(1) {
        bal_ecr_recevoir(Vitesse_Pompe,(char *)&vitesse); /* Attente
de message */
        Actionner_Pompe(vitesse); /* Fonction de pilotage de la pompe
*/
    }
}
```

```
void Controler_Mine() {
    float niveau_methane;
    float vitesse_pompe;
    float niveau_eau;
    typedef enum {nominal, pompe, alerte_et_pompe,  alerte}
t_etat_controle;
    t_etat_controle etat=nominal;
    while (1) {
        bal_ecr_recevoir(Niveau_Methane,(char *)&niveau_methane); /*
Attente sur boîte aux lettres */
        /* Lecture du module de données */
        pthread_mutex_lock(&sNiveau_Eau);
        niveau_eau=Niveau_Eau;
        pthread_mutex_unlock(&sNiveau_Eau);
        /* Implémentation du diagramme Etats/Transitions donné sur la
figure 2.33 */
        switch (etat) {
        case nominal:
            if (niveau_methane >= MS_L1) {
            /* Seuil d'alerte */
                sem_post(&Evt_Alarme); /* Alarme */
                etat = alerte;
            } else if (niveau_eau >= HLS) {
            /* Allumage de la pompe, pour simplifier, nous supposons
que nous la pilotons de façon proportionnelle */
                vitesse_pompe = niveau_eau-LLS;
                bal_ecr_envoyer(Vitesse_Pompe,(char*)&vitesse_pompe);
                etat = pompe;
            }
            break;
        case pompe:
            if (niveau_methane >= MS_L1) {
            /* Seuil d'alerte */
                sem_post(&Evt_Alarme); /* Alarme */
                etat = alerte_et_pompe;
            } else if (niveau_eau <= LLS) {
                vitesse_pompe = 0;
                bal_ecr_envoyer(Vitesse_Pompe,(char*)&vitesse_pompe);
                etat = nominal;
            } else {
            /* Pilotage proportionnel */
                vitesse_pompe = niveau_eau-LLS;
                bal_ecr_envoyer(Vitesse_Pompe,(char*)&vitesse_pompe);
}
            break;
        case alerte:
            if (niveau_methane < MS_L1) {
            /* Seuil d'alerte */
                sem_post(&Evt_Alarme); /* Extinction de l'alarme */
                etat = nominal;
            } else if (niveau_methane < MS_L2 && niveau_eau > HLS) {
            /* Allumage de la pompe, pour simplifier, nous supposons
que nous la pilotons de façon proportionnelle */
                vitesse_pompe = niveau_eau-LLS;
                bal_ecr_envoyer(Vitesse_Pompe,(char*)&vitesse_pompe);
                etat = alerte_et_pompe;
            }
            break;
        case alerte_et_pompe:
```

© Dunod – La photocopie non autorisée est un délit.

```
            if (niveau_methane >= MS_L2 || niveau_eau <= LLS) {
            /* Alerte MS_L2 ou niveau bas => extinction de la pompe */
                vitesse_pompe = 0;
                bal_ecr_envoyer(Vitesse_Pompe,(char*)&vitesse_pompe);
                etat = alerte;
            } else if (niveau_methane < MS_L1) {
            /* Niveau de méthane sous le seuil d'alerte */
                sem_post(&Evt_Alarme); /* Extinction de l'alarme */
                etat = pompe;
            } else {
            /* Pilotage proportionnel */
                vitesse_pompe = niveau_eau-LLS;
                bal_ecr_envoyer(Vitesse_Pompe,(char*)&vitesse_pompe);
}
            break;
        };
    }
}

void main() {
    pthread_t acquerir_niveau_methane, afficher_alarme,
acquerir_capteur_eau, commander_pompe, controler_mine;
    /* Initialisation du matériel (capteurs, actionneurs) */
    /* etc. */

    /* Initialisation du module de données */
    Niveau_Eau=(HLS-LLS)/2+LLS; /* La valeur initiale est choisie de
sorte à ne pas lancer de commande : on suppose ici que le niveau est
au milieu de HLS et LLS */
    /* On pourrait aussi choisir de lire une première fois le
capteur : Niveau_Eau = Lire_Capteur_Eau() ; */
    pthread_mutex_init(&sNiveau_Eau,NULL);
    /* Initialisation des outils de communication et synchronisation
*/
    Niveau_Methane=bal_ecr_init(sizeof(float));
    Vitesse_Pompe=bal_ecr_init(sizeof(float));
    sem_init(&Evt_Alarme,0,0); /* Sémaphore de synchronisation */

pthread_create(&acquerir_niveau_methane,NULL,Acquerir_Niveau_Methane,
NULL);

pthread_create(&acquerir_capteur_eau,NULL,Acquerir_Capteur_Eau,NULL);
    pthread_create(&afficher_alarme,NULL,Afficher_Alarme,NULL);
    pthread_create(&commander_pompe,NULL,Commander_Pompe,NULL);
    pthread_create(&controler_mine,NULL,Controler_Mine,NULL);
    pthread_join(acquerir_niveau_methane,NULL);
    pthread_join(acquerir_capteur_eau,NULL);
    pthread_join(afficher_alarme,NULL);
    pthread_join(commander_pompe,NULL);
    pthread_join(controler_mine,NULL);
}
```

Remarquons l'implémentation du diagramme états/transitions défini pour le contrôle de la mine.

6.2.2 Implémentation sur exécutif VxWorks

■ Interface de programmation

VxWorks® est l'exécutif temps réel le plus répandu sur le marché. Dans ses versions 5.x, il implémente une grande partie des interfaces POSIX 1003.1b et 1003.1c. Il est donc possible de programmer un système VxWorks® comme un système POSIX. Cependant, afin de donner un autre exemple de programmation d'applications de contrôle-commande en langage C, nous présentons ici la programmation basée sur l'interface propriétaire de VxWorks®.

■ Implémentation des éléments DARTS

□ Tâches

Une tâche (*task*) est représentée par une fonction pouvant posséder au plus dix paramètres de type `int`. Un paramètre peut être utilisé pour passer une donnée de la taille d'un entier, on peut donc y passer typiquement un entier, ou un pointeur. Chaque tâche se lance dynamiquement grâce à la fonction `taskSpawn` ou bien à la combinaison des deux fonctions `taskInit` et `taskResume`. La signature de cette fonction est :

```
int taskSpawn(char *nom, int priority, int options, int
stackSize,FUNCPTR entryPoint,int arg1,…,int arg10);
```

La valeur de retour de cette fonction est l'identificateur de la tâche créée, `nom` est le nom alphanumérique de la tâche, `priority` est sa priorité (attention, par rapport à POSIX, l'ordre est inversé, et 0 est la priorité la plus forte), `stackSize` est la taille de sa pile, `entryPoint` est la fonction implémentant la tâche, et les paramètres `arg1` à `arg10` sont les paramètres passés à cette fonction.
Enfin, le paramètre `options` permet de spécifier différentes particularités au système, comme l'utilisation ou non des calculs à virgule flottante afin de déterminer quels registres doivent être sauvegardés lors d'un changement de contexte.
Les tâches s'exécutent en mode détaché (§ 5.4.1, p. 235). Notons qu'il n'y a pas de hiérarchie de tâches.
Un exemple très simple de programme multitâche VxWorks® est donné ci-après :

```
#include <taskLib.h> /* Librairie tâches */
#include <stdio.h> /* Entrées/Sorties (printf) */

void f(int no) {
/* Fonction implémentant une tâche */
    int i;
    for (i=0;i<10;i++) {
        printf("Tache %d : iteration %d\n",no,i);
        taskDelay(1); /* Attente d'au moins un tick */
    }
}

void LancerTaches() {
taskSpawn("tache1",90,0,20000,f,1,0,0,0,0,0,0,0,0,0); /* Lancement de
la tâche 1, de priorité 90, avec 20000 octets de pile, implémentée par
la fonction f, avec le 1er paramètre valant 1 */
```

© Dunod – La photocopie non autorisée est un délit.

```
taskSpawn("tache2",80,0,10000,f,2,0,0,0,0,0,0,0,0,0); /* Lancement de
la tâche 2, de priorité 80, avec 10000 octets de pile, implémentée par
la fonction f, avec le 1er paramètre valant 1 */
}
```

Nous pouvons noter que le système se base sur les *ticks* pour spécifier le temps. D'autre part, l'une des particularités des exécutifs embarqués est que le programmeur ne définit pas de programme principal : lorsque le code est compilé et chargé sur la cible, la fonction `LancerTaches` peut être lancée à l'aide d'un *shell*.

☐ Modules de données

Les modules de données DARTS peuvent être implémentés à l'aide d'un sémaphore d'exclusion mutuelle. Ainsi, une partie du code correspondant à la figure 6.24 est donnée ci-après.

```
#include <taskLib.h>
#include <semLib.h> /* Module définissant les sémaphores d'exclusion
mutuelle */

/* Type utilisé pour le module de données */
typedef struct {
    float Temperature;
    float Pression;
} t_Temperature_Pression;

t_Temperature_Pression MDD1; /*Déclaration du module de données MDD1*/
SEM_ID s_MDD1; /* Mutex protégeant MDD1 */

void T1() {
/* Tâche 1 lisant le module de données */
    t_Temperature_Pression TP;
    while (1) { /* Faire toujours */
        /* Exclusion mutuelle sur le module de données */
        semTake(s_MDD1,WAIT_FOREVER);
        /* Lecture du MDD */
        TP=MDD1;
        semGive(s_MDD1);
        /* etc. */
    }
}

void T2() {
/* Tâche 2 modifiant le module de données */
    float T,P;
    while (1) { /* Faire toujours */
        /* Lecture des capteurs en dehors de la section critique */
        P=Lire_Capteur_Pression();
        T=Lire_Capteur_Temperature();
        /* Exclusion mutuelle sur le module de données */
        semTake(s_MDD1,WAIT_FOREVER);
        /* Modification du MDD */
        MDD1.Temperature=T;
        MDD1.Pression=P;
        semGive(s_MDD1);
        /* etc. */
    }
}
```

```
void LancerTaches() {
    s_MDD1=semMCreate(SEM_Q_PRIORITY|SEM_INVERSION_SAFE); /
*Initialisation du mutex de MDD1 : La file d'attente des tâches est
gérée par niveaux de priorité, et le protocole à priorité héritée est
utilisé */
    /* Initialisation de MDD1 */
    MDD1.Temperature=Lire_Capteur_Temperature();
    MDD1.Pression=Lire_Capteur_Pression();
    /* Lancement des tâches */
taskSpawn("T1",90,0,20000,T1,0,0,0,0,0,0,0,0,0,0);
taskSpawn("T2",100,0,20000,T2,0,0,0,0,0,0,0,0,0,0);
}
```

On peut remarquer que le sémaphore est déclaré de façon globale, ce qui le rend visible aux deux tâches concernées. Notons qu'il est créé et initialisé avant le lancement des tâches, par exemple dans le programme de lancement des tâches.
Il est nécessaire de réduire la durée des sections critiques, par conséquent, sur cet exemple, l'accès au matériel via les fonctions de lecture de capteurs, très longues, doit s'effectuer à l'extérieur de la section critique d'accès au module de données. Rappelons que VxWorks® ne propose pas le protocole à priorité plafond.

☐ Boîtes aux lettres

VxWorks® définit des boîtes aux lettres bornées sans écrasement, avec une gestion des priorités de message à deux niveaux (normal, urgent). Ainsi, l'implémentation du schéma DARTS donné sur la figure 6.24 est donnée ci-après.

```
#include <taskLib.h>
#include <msgQLib.h> /* Module définissant les boîtes aux lettres */

/* Type utilisé pour le module de données */
typedef struct {
      float Temperature;
      float Pression;
} t_Temperature_Pression;

MSG_Q_ID B1; /*Déclaration de la boîte aux lettres*/

void T1() {
/* Tâche 1 en attente de messages sur la boîte aux lettres */
      t_Temperature_Pression TP;
      while (1) { /* Faire toujours */
            msgQReceive(B1,(char*)&TP,sizeof(TP),WAIT_FOREVER);
            /* etc. */
      }
}

void T2() {
/* Tâche 2 envoyant des messages */
      t_Temperature_Pression TP;
      while (1) { /* Faire toujours */
            /* Lecture des capteurs en dehors de la section critique*/
            TP.Pression=Lire_Capteur_Pression();TP.Temperatu
   re=Lire_Capteur_Temperature();
            msgQSend(B1, (char*)&TP, sizeof(TP), WAIT_FOREVER,
MSG_PRI_NORMAL); /* Envoi du message en priorité normale, si la boîte
est pleine, cette instruction est bloquante */
```

© Dunod – La photocopie non autorisée est un délit.

```
            /* etc. */
        }
}

void LancerTaches() {
        B1=msgQCreate(1,sizeof(t_Temperature_Pression),MSG_Q_PRIORITY);
/*Initialisation de la boîte aux lettres B1 : tampon de 1 message. Les
tâches en attente sont gérées par niveau priorité */
        /* Lancement des tâches */
        taskSpawn("T1",90,0,20000,T1,0,0,0,0,0,0,0,0,0,0);
        taskSpawn("T2",100,0,20000,T2,0,0,0,0,0,0,0,0,0,0);
}
```

□ Synchronisation

Comme en POSIX, la synchronisation peut être implémentée à l'aide d'un sémaphore à compte. Ainsi, l'implémentation du diagramme DARTS donné sur la figure 6.28 est donnée ci-après.

```
#include <taskLib.h>
#include <semLib.h> /* Module définissant les sémaphores à compte */

SEM_ID S1; /*Déclaration du sémaphore de synchronisation */

void T1() {
/* Tâche déclenchant la synchronisation */
        while (1) { /* Faire toujours */
            /* etc. */
            semGive(S1); /* Déclenchement de la synchronisation */
            /* etc. */
        }
}

void T2() {
/* Tâche en attente de synchronisation */
        while (1) { /* Faire toujours */
            semTake(S1,WAIT_FOREVER); /* Attente de synchronisation
*/
            /* etc. */
        }
}
void LancerTaches() {
        S1=semCCreate(SEM_Q_FIFO,0); /*Initialisation du sémaphore de
synchronisation à 0, la file d'attente des tâches est FIFO pure (il
n'est pas nécessaire de gérer les priorités puisqu'il ne doit y avoir
qu'une tâche en attente de cette synchronisation) */
        /* Lancement des tâches */
        taskSpawn("T1",90,0,20000,T1,0,0,0,0,0,0,0,0,0,0);
        taskSpawn("T2",100,0,20000,T2,0,0,0,0,0,0,0,0,0,0);
}
```

Remarquons que les sémaphores à compte ne se différencient, au niveau du programme, des sémaphores d'exclusion mutuelle qu'à la création (utilisation de `semCCreate` pour les premiers, et de `semMCreate` pour les seconds).

☐ Tâches périodiques

Comme nous l'avons vu précédemment, la base de temps de VxWorks est le *tick*. Le nombre de *ticks* par seconde est donné par la fonction `sysClkRateGet`, et peut être modifié à l'aide de `sysClkRateSet`. Un intervalle de temps est donc forcément multiple de 1/`sysClkRateGet()`. On pourrait donc être tenté d'augmenter la fréquence des *ticks* afin d'obtenir une meilleure granularité temporelle. Cependant, étant donné que VxWorks® est un exécutif dirigé par le temps, le système prend la main à chaque *tick*. Par conséquent, augmenter la fréquence des *ticks* augmente l'*overhead* (surcoût d'utilisation processeur due à l'exécutif). Il est donc conseillé de ne pas descendre l'intervalle entre deux *ticks* en dessous d'une milliseconde, bien qu'il soit possible de diminuer cet intervalle aux alentours de quelques dizaines de microsecondes (ce qui entraîne naturellement un *overhead* important).

L'inconvénient majeur de l'interface de programmation native de VxWorks® est qu'il n'existe aucune primitive permettant d'attendre jusqu'à une certaine date (en *ticks*). Par conséquent, le seul moyen purement logiciel de déclencher périodiquement une tâche consiste à attendre un certain nombre de *ticks*, ce qui entraîne une dérive des horloges (§ 5.3.1, p. 216). Ainsi, l'implémentation des tâches de la figure 6.29 est donnée ci-après.

```
#include <taskLib.h>

void Periodique(int periode_ticks) {
/* Tâche périodique (avec dérive des horloges) */
/* periode_ticks est sa période désirée en ticks */
      while (1) { /* Faire toujours */
            /* etc. */
            taskDelay(periode_ticks); /* Attente d'au moins
periode_ticks ticks */
      }
}

void LancerTaches() {
      /* Lancement des tâches */
taskSpawn("Periodique1",90,0,20000,Periodique,sysClkRateGet()/
10,0,0,0,0,0,0,0,0,0); /* Periode de 100 ms */
taskSpawn("Periodique2",100,0,20000,Periodique1,sysClkRateGet()/
18,0,0,0,0,0,0,0,0,0); /* Periode de 55.555… ms */
}
```

Notons que la période est arrondie au *tick* inférieur, ainsi par exemple, si la fréquence des *ticks* est de 300 *ticks* par seconde (un *tick* dure 3,33… millisecondes), la période de la tâche `Periodique1` est de 10 *ticks*, soit 33,33 millisecondes, et celle de `Periodique2` est de 18 *ticks*, soit 60 millisecondes.

☐ Tâches réveillées par interruption

La gestion des interruptions est très simple et se base comme POSIX sur le concept d'ISR (fonction appelée lors de l'interruption) s'exécutant dans un contexte spécifique aux interruptions et n'étant pas autorisée à appeler des primitives bloquantes ou suspensives, et de DSR, tâche classique de traitement, déclenchée par synchronisation par l'ISR (§ 5.2.3, p. 206). Ainsi, le mécanisme de réveil de la tâche maté-

© Dunod – La photocopie non autorisée est un délit.

rielle présente sur la figure 6.30 s'implémente simplement par synchronisation sur sémaphore.

```
#include <semLib.h>
#include <taskLib.h>

SEM_ID declencheT1; /* Sémaphore de synchronisation */

void traiteIntr(int dummy) {
/* ISR déclenchant la tâche de traitement par synchronisation */
      semGive(declencheT1);
}

void TacheMaterielle () {
/* Tâche de traitement de l'interruption */
      while (1) {
            semTake(declencheT1,WAIT_FOREVER); /* Attente de
synchronisation */
            …
      }
}

void LancerTaches() {
      declencheT1 = semCCreate(SEM_Q_FIFO,0); /* Initialisation du
sémaphore de synchronisation */
      intConnect(INUM_TO_IVEC(7),traiteIntr,0); /* La fonction
traiteIntr est appelée sur l'interruption IRQ7 */
      taskSpawn("TacheMaterielle",90,0,20000,TacheMaterielle,0,0,0,0,
0,0,0,0,0,0);
}
```

■ **Exemple**

La mise en œuvre des différents outils présentés pour l'implémentation d'éléments DARTS sur VxWorks® avec la librairie native est faite sur l'exemple de la « gestion de la sécurité d'une mine » dont le diagramme DARTS est donné sur la figure 3.27. Afin d'en faciliter la lecture, le système de commande est présenté à plat, sans utilisation de modules spécifiques.

```
#include <taskLib.h>
#include <msgQLib.h>
#include <semLib.h>
#include "procede.h" /* Fonctions d'accès aux capteurs et actionneurs
*/

/* Définition de constantes */
#define LLS 5.3
/* Valeur sous laquelle le niveau est considéré trop bas */
#define HLS 17.8
/* Valeur au-dessus de laquelle le niveau est considéré trop haut */
#define MS_L1 128
#define MS_L2 180
/* Seuils d'alerte de méthane*/

/* Définition des outils de communication/synchronisation */
/* Module de données Niveau_Eau */
float Niveau_Eau; /* Module de données */
SEM_ID sNiveau_Eau; /* Mutex du module de données */
```

```
MSG_Q_ID Niveau_Methane; /* Boîte aux lettres de taille 1 à écrasement
*/
MSG_Q_ID Vitesse_Pompe; /* Boîte aux lettres de taille 1 à écrasement
*/
SEM_ID Evt_Alarme; /* Synchronisation */

void Acquerir_Niveau_Methane() {
/* Tâche périodique d'acquisition */
    float niveau_methane;
    while (1) {
        niveau_methane=Lire_Capteur_Methane(); /* Lecture du capteur
*/
        msgQSend(Niveau_Methane, (char*)&niveau_methane,
sizeof(niveau_methane), WAIT_FOREVER, MSG_PRI_NORMAL);/* Envoi du
message */
            taskDelay(sysClkRateGet()/2); /* Periode de 500 ms avec
dérive des horloges */
    }
}

void Acquerir_Capteur_Eau() {
/* Tâche périodique d'acquisition */
    float niveau_eau;
    while (1) {
        niveau_eau=Lire_Capteur_Eau(); /* Lecture du capteur */

        /* Ecriture dans le module de données */
        semTake(sNiveau_Eau);
        Niveau_Eau=niveau_eau; /* Noter que l'on n'appelle pas
l'acquisition dans la section critique afin de minimiser la durée de
celle-ci */
        semGive(sNiveau_Eau);
taskDelay(sysClkRateGet()*5); /* Periode de 5 s */
    }
}

void Afficher_Alarme () {
    unsigned char etat_alarme=0; /* Etat courant de l'alarme (0
éteinte, 1 allumée) */
    while (1) {
        semTake(Evt_Alarme); /* Attente de synchronisation */
        etat_alarme=1-etat_alarme;
       Piloter_Alarme(etat_alarme); /* Fonction actionnant ou coupant
l'alarme */
    }
}

void Commander_Pompe() {
    float vitesse;
    while(1) {

msgQReceive(Vitesse_Pompe,(char*)&vitesse,sizeof(vitesse),WAIT_FOREVER
); /* Attente de message */
      Actionner_Pompe(vitesse); /* Fonction de pilotage de la pompe */
    }
}
void Controler_Mine() {
    float niveau_methane;
    float vitesse_pompe;
```

© Dunod – La photocopie non autorisée est un délit.

```
    float niveau_eau;
    typedef enum {nominal, pompe, alerte_et_pompe,  alerte}
t_etat_controle;
    t_etat_controle etat=nominal;
    while (1) {
        msgQReceive(Niveau_Methane,(char
*)&niveau_methane,sizeof(niveau_methane),WAIT_FOREVER); /* Attente
sur boîte aux lettres */
        /* Lecture du module de données */
        semTake(sNiveau_Eau);
        niveau_eau=Niveau_Eau;
        semGive(sNiveau_Eau);
        /* Implémentation du diagramme Etats/Transitions donné sur la
figure 2.33 */
        switch (etat) {
        case nominal:
            if (niveau_methane >= MS_L1) {
            /* Seuil d'alerte */
                semGive(Evt_Alarme); /* Alarme */
                etat = alerte;
            } else if (niveau_eau >= HLS) {
            /* Allumage de la pompe, pour simplifier, nous supposons
que nous la pilotons de façon proportionnelle */
                vitesse_pompe = niveau_eau-LLS;

msgQSend(Vitesse_Pompe,(char*)&vitesse_pompe,sizeof(vitesse_pompe),WA
IT_FOREVER, MSG_PRI_NORMAL);
                etat = pompe;
            }
            break;
        case pompe:
            if (niveau_methane >= MS_L1) {
            /* Seuil d'alerte */
                semGive(Evt_Alarme); /* Alarme */
                etat = alerte_et_pompe;
            } else if (niveau_eau <= LLS) {
                vitesse_pompe = 0;
                msgQSend(Vitesse_Pompe,(char*)&vitesse_pompe,
sizeof(vitesse_pompe), WAIT_FOREVER, MSG_PRI_NORMAL);
                etat = nominal;
            } else {
            /* Pilotage proportionnel de la pompe */
                vitesse_pompe = niveau_eau-LLS;
                msgQSend(Vitesse_Pompe,(char*)&vitesse_pompe,
sizeof(vitesse_pompe), WAIT_FOREVER, MSG_PRI_NORMAL);
}
            break;
        case alerte:
            if (niveau_methane < MS_L1) {
            /* Seuil d'alerte */
                semGive(Evt_Alarme); /* Extinction de l'alarme */
                etat = nominal;
            } else if (niveau_methane < MS_L2 && niveau_eau > HLS) {
            /* Allumage de la pompe, pour simplifier, nous supposons
que nous la pilotons de façon proportionnelle */
                vitesse_pompe = niveau_eau-LLS;
                msgQSend(Vitesse_Pompe,(char*)&vitesse_pompe,
         sizeof(vitesse_pompe),WAIT_FOREVER, MSG_PRI_NORMAL);
                etat = alerte_et_pompe;
```

```
            }
            break;
        case alerte_et_pompe:
            if (niveau_methane >= MS_L2 || niveau_eau <= LLS) {
            /* Alerte MS_L2 ou niveau bas => extinction de la pompe */
                vitesse_pompe = 0;
                msgQSend(Vitesse_Pompe,(char*)&vitesse_pompe,
         sizeof(vitesse_pompe),WAIT_FOREVER, MSG_PRI_NORMAL);
                etat = alerte;
            } else if (niveau_methane < MS_L1) {
            /* Niveau de méthane sous le seuil d'alerte */
                semGive(Evt_Alarme); /* Extinction de l'alarme */
                etat = pompe;
            } else {
            /* Pilotage proportionnel */
                vitesse_pompe = niveau_eau-LLS;
                msgQSend(Vitesse_Pompe,(char*)&vitesse_pompe,
         sizeof(vitesse_pompe),WAIT_FOREVER, MSG_PRI_NORMAL);
}
            break;
        };
    }
}

TASK_ID acquerir_niveau_methane, afficher_alarme,
acquerir_capteur_eau, commander_pompe, controler_mine;
/* Afin de conserver la valeur des identificateurs de tâches, ceux-ci
sont déclarés de façon globale */

void LancerTaches() {
    /* Initialisation matérielle du système (capteurs, actionneurs) */
    /* etc. */

    /* Initialisation du module de données */
    Niveau_Eau=(HLS-LLS)/2+LLS; /* La valeur initiale est choisie de
sorte à ne pas lancer de commande : on suppose ici que le niveau est
au milieu de HLS et LLS */
    /* On pourrait aussi choisir de lire une première fois le
capteur : Niveau_Eau = Lire_Capteur_Eau() ; */

sNiveau_Eau=semMCreate(SEM_Q_PRIORITY|SEM_INVERSION_SAFE|SEM_DELETE_S
AFE); /* Constater l'ajout de SEM_DELETE_SAFE */
    /* Initialisation des outils de communication et synchronisation
*/
    Niveau_Methane=msgQCreate(1,sizeof(float),MSG_Q_FIFO); /* Noter
que la gestion des files d'attente d'envoie et de réception est FIFO,
car une seule tâche émet, une seule tâche lit */
    Vitesse_Pompe=msgQCreate(1,sizeof(float), MSG_Q_FIFO);
    Evt_Alarme=semCCreate(SEM_Q_FIFO,0); /* Sémaphore de
synchronisation */
    acquerir_niveau_methane=taskSpawn("AcqMethane",70,0,20000,
Acquerir_Niveau_Methane,0,0,0,0,0,0,0,0,0,0);
    acquerir_capteur_eau=taskSpawn("AcqEau",71,0,20000,
Acquerir_Capteur_Eau,0,0,0,0,0,0,0,0,0,0);
    afficher_alarme=taskSpawn("Alarme",40,0,20000,
Afficher_Alarme,0,0,0,0,0,0,0,0,0,0);
    commander_pompe=taskSpawn("CmdPompe",45,0,20000,
Commander_Pompe,0,0,0,0,0,0,0,0,0,0);
```

© Dunod – La photocopie non autorisée est un délit.

```
    controler_mine=taskSpawn("CtrlMine",48,0,20000,
Controler_Mine,0,0,0,0,0,0,0,0,0,0);
}

void StopperTaches() {
/* Stoppe toutes les tâches et détruit les outils de communication/
synchronisation, ce qui permet pendant la phase de mise au point, sans
redémarrer la cible, de stopper les tâches, modifier leur code source,
le recharger sur la cible, et de lancer à nouveau les tâches.
L'éventualité d'un arrêt des tâches explique le fait que les
sémaphores d'exclusion mutuelle soient créés avec l'option
SEM_DELETE_SAFE */
taskDelete(acquerir_niveau_methane);
taskDelete(acquerir_capteur_eau);
taskDelete(afficher_alarme);
taskDelete(commander_pompe);
taskDelete(controler_mine);
semDelete(sNiveau_Eau);
semDelete(Evt_Alarme);
msgQDelete(Niveau_Methane);
msgQDelete(Vitesse_Pompe);
}
```

L'un des avantages de VxWorks® est que l'environnement de développement lié (Tornado®) permet le chargement et le remplacement de modules sans redémarrer le système cible. Il en résulte que pour faciliter la mise au point, on intègre généralement une fonction de destruction des tâches et des éléments de communication et de synchronisation. Par conséquent, afin d'éviter qu'une tâche ne soit détruite lorsqu'elle possède un sémaphore d'exclusion mutuelle, on peut être amené à utiliser l'option `SEM_DELETE_SAFE` lors de la création des sémaphores d'exclusion mutuelle. Cette option a pour effet de protéger une tâche de la suppression tant qu'elle détient un sémaphore d'exclusion mutuelle.
Ce code exemple, une fois compilé et chargé sur la cible, permet, à partir du *shell* de lancer directement les fonctions C. Ainsi, après chargement, on exécute au niveau du *shell* la fonction `LancerTaches`. Et pour arrêter le système, on exécutera toujours au niveau du *shell* la fonction `StopperTaches`.

6.3 Programmation multitâche en langage Ada

6.3.1 Les tâches Ada

La possibilité de créer des tâches en langage Ada est native. Il existe un constructeur de type tâche (à l'instar des `array`, `record`, etc.) nommé `task type`. Ainsi, on peut définir des types de tâches, et les instancier pour créer des tâches.

■ La création des tâches Ada

Lorsqu'une tâche est instanciée de façon globale (variable de paquetage), comme c'est souvent le cas lors d'une implémentation DARTS, cette tâche est lancée automatiquement au démarrage du programme principal. Lorsqu'une tâche est instanciée dynamiquement (tâche déclarée dans un bloc, ou allouée dynamiquement par `new`), elle est lancée au moment de l'instanciation.

Les tâches sont toujours attachées automatiquement au programme principal : tant qu'il existe une tâche en fonctionnement, le programme principal ne se termine pas, et reste « bloqué » juste avant son `end` final.
La déclaration d'un type tâche s'effectue en deux parties : une spécification de type (`task type`), et un corps de type (`task body`). Ainsi, le code suivant définit un type de tâche et l'instancie deux fois.

```
with Ada.Text_Io; use Ada.Text_Io;
procedure Deux_Taches is
   -- Crée un type tâche et l'instancie deux fois
   task type T_Matache;  -- Spécification du type tâche
   task body T_Matache is
   -- Corps de la tâche exécuté lorsqu'elle est instanciée
   begin
      for I in 1..10 loop
         Put_Line("Iteration "&Integer'Image(I)); -- Affichage du
numéro d'itération
      end loop;
   end; -- Fin de la tâche
   T1, T2 : T_Matache; -- Deux instanciations, T1 et T2 sont deux
tâches
begin -- Démarrage du programme principal, T1 et T2 sont
automatiquement lancées
   null; -- Le programme principal ne fait rien
end; -- La fin n'est atteinte qu'à la terminaison des tâches lancées
```

Le type `T_Matache` étant justement un type, il est possible de construire d'autres types, comme des tableaux de tâches de ce type, des types enregistrement, etc. Un type de tâche est un type « limité », c'est-à-dire que toute affectation ou comparaison de deux éléments est impossible.
Souvent les tâches ne sont destinées à être instanciées qu'une seule fois dans le programme. Une facilité d'écriture consiste alors à instancier des tâches de type anonyme, ce qui consiste à définir un type sans le nommer pour l'instancier. Ainsi, le code ci-après définit un type de tâche et l'instancie : noter la présence du mot clé « type » dans la spécification.

```
with Ada.Text_Io;use Ada.Text_Io;
with System;use System;
procedure une_tache_typee is
   task type T_Matache is
      -- Noter la présence du mot clé "type" dans la déclaration
      pragma Priority(Default_Priority);
   end T_Matache;
   task body T_Matache is
   begin
      for I in 1..10 loop
         Put_Line("Iteration de tache");
      end loop;
   end;
   Matache : T_Matache; -- Instanciation
begin -- La tâche instanciée est lancée
   null;
end; -- ne se termine que lorsque la tâche est terminée
```

© Dunod – La photocopie non autorisée est un délit.

Le code ci-après est équivalent, et définit une tâche de type anonyme : noter l'absence du mot clé « type » dans la spécification.

```
with Ada.Text_Io;use Ada.Text_Io;
with System;use System;
procedure une_tache_anonyme is
   task Matache is
      -- Noter l'absence du mot clé "type" dans la déclaration
      pragma Priority(Default_Priority);
   end Matache;
   task body Matache is
   begin
      for I in 1..10 loop
         Put_Line("Iteration de tache");
      end loop;
   end;
   -- La tâche Matache a été instanciée en même temps qu'elle a été
déclarée
begin -- La tâche instanciée est lancée
   null;
end; -- ne se termine que lorsque la tâche est terminée
```

Priorités des tâches

Les attributs d'un type tâche, comme sa priorité par exemple, sont placés dans la spécification du type. On peut passer des paramètres à une tâche lors de son instanciation par la technique des discriminants. Le type d'un discriminant est forcément un type discret (entier, caractère, type énuméré, etc.) ou un pointeur. Ainsi, le code ci-après crée un type de tâche ayant un numéro et une priorité passés en paramètre. Notons qu'il est possible de changer la priorité d'une tâche au cours de son exécution grâce à la fonction `Set_Priority`.

```
with Ada.Text_Io; use Ada.Text_Io;
with System;use System; -- Pour l'utilisation du type Priority
procedure Deux_Taches_Discriminant is
   task type T_Matache(Priorite: Integer;Numero:Integer) is
      -- Le discriminant permet de passer des paramètres scalaires
lors de l'instanciation
      pragma Priority(Priorite); -- Définit la priorité de la tâche
   end T_Matache;
   task body T_Matache is
   begin
      for I in 1..10 loop
         Put_Line("Iteration "&Integer'Image(I)&" de la tache
"&Integer'Image(Numero));
         -- Affichage du numéro d'itération et du numéro de la tâche
         -- Noter qu'il provient du discriminant
      end loop;
   end; -- Fin de la tâche
   T1 : T_Matache(Priority'First+10,1);
   -- T1 a une priorité 10 niveaux au-dessus du niveau minimal, et a
le numéro 1
   T2 : T_Matache(Priority'Last-10,2);
   -- T1 a une priorité 10 niveaux au-dessous du niveau maximal, et a
le numéro 2
begin -- Démarrage du programme principal, T1 et T2 sont
automatiquement lancées
```

```
   null; -- Le programme principal ne fait rien
end; -- La fin n'est atteinte qu'à la terminaison des tâches lancées
```

Le paquetage `system` définit un type `Any_Priority` comme un intervalle d'au moins 31 valeurs. Ce type est ensuite décomposé en deux sous-types :

– les priorités normales (type `priority`) définies comme un intervalle d'`Any_Priority`, ce qui permet au passage l'utilisation des attributs (comme par exemple `priority'first` qui donne la valeur minimale de priorité) comme sur tout type discret. La norme Ada impose au moins 30 niveaux dans ce type. La priorité par défaut d'une tâche est celle du niveau qui l'a créé. La priorité par défaut du programme principal est `Default_Priority`, se trouvant au milieu de l'intervalle de priorités (`(priority'last-priority'first)/2+priority'first`) ;

– les priorités d'interruption (`Interrupt_Priority`), strictement supérieures aux priorités normales, correspondent aux plus hauts niveaux de priorité du système. Typiquement, elles ne sont utilisées que pour les routines de traitement d'interruption (§ 6.3.2, p. 326). La norme impose au moins un niveau de priorité d'interruption. La priorité `Interrupt_Priority'Last` est donc la plus grande priorité que l'on puisse trouver sur le système.

Communication et synchronisation

Dans sa première version, en 1983, Ada n'intégrait que le rendez-vous (§ 5.2.2, p. 198). Divers inconvénients apparaissaient à l'utilisation, comme l'obligation de créer des tâches pour implémenter des moniteurs (qui sont des objets passifs), l'impossibilité de communiquer de façon asynchrone, et la difficulté d'étudier le comportement temporel de ces outils de communications. Par conséquent, cet outil n'est pas utilisé par la suite.

Depuis Ada95, un deuxième outil a été introduit : le moniteur de haut niveau (objet protégé). De par sa nature, le moniteur Ada permet d'implémenter tout type de synchronisation et de communication, synchrone et asynchrone, comme cela est montré par la suite.

Comme pour les tâches, il existe un constructeur de types d'objet protégé (`protected type`) et il est possible de définir des objets protégés anonymes (`protected`).

La spécification d'un objet protégé contient une partie privée, correspondant aux variables internes au moniteur, et un ensemble de primitives. Il y a trois types de primitives :

– les procédures (`procedure`) sont des primitives non réentrantes par rapport aux autres primitives de l'objet protégé (*i.e.* une procédure ne peut pas avoir lieu tant qu'une autre primitive de l'objet protégé est exécutée). Cependant, elles sont non bloquantes, c'est-à-dire qu'elles n'intègrent pas la possibilité d'attendre un événement du moniteur (de faire l'équivalent d'une instruction `wait` sur le moniteur, voir § 5.2.2, p. 186). Les procédures peuvent avoir des paramètres en entrée ou en sortie ;

– les procédures gardées (`entry`) sont des procédures dont l'accès est conditionné (gardé), c'est-à-dire que tout se passe comme si un code `tant que la garde n'est pas vraie faire wait` se trouvait au début de la primitive ;

© Dunod – La photocopie non autorisée est un délit.

– les fonctions (`function`) sont des primitives en lecture seule sur l'objet protégé : elles ne peuvent pas modifier les variables internes de l'objet. En revanche, les fonctions sont réentrantes les unes par rapport aux autres (une fonction peut s'exécuter même si une fonction du même objet protégé est en cours d'exécution). Ces primitives permettent d'affiner les accès aux objets protégés en leur donnant des primitives de type lecteur/écrivain.

Par exemple, le code ci-après définit un sémaphore à l'aide d'un objet protégé.

```
protected type Semaphore(Valeur_Initiale: Positive:=1) is
   -- Utilisation d'un discriminant donnant la valeur initiale
   -- Noter l'emploi du mot clé "type" permettant de définir un type
   -- d'objet protégé
   entry Prendre; -- Procédure gardée (il n'est possible de prendre
                  -- le sémaphore que lorsque sa valeur est > 0)
   procedure Vendre;
   function Valeur return Natural;  -- Renvoie la valeur courante du
compte du sémaphore
private -- Variables internes
   Valeur: Natural:=Valeur_Initiale; -- Compte du sémaphore
end Semaphore;
protected body Semaphore is
   entry Prendre when Valeur > 0 is
      -- Tant que le sémaphore est <=0, les tâches appelant cette
      -- entrée sont bloquées et mises en attente dans la file
      -- d'attente de l'entrée
   begin
      Valeur:=Valeur-1;
   end Prendre;
   procedure Vendre is
   begin
      Valeur:=Valeur+1;
   end Vendre;
   function Valeur return Natural is
   begin
      return Valeur;
   end;
end Semaphore;
```

La norme Ada propose le protocole à priorité plafond immédiat pour éviter l'inversion de priorité lors d'accès concurrents à un objet protégé. Par conséquent, un objet protégé peut être muni d'une priorité : sa priorité plafond, c'est-à-dire la priorité maximale des tâches susceptibles de l'utiliser. Ada étant un langage sûr, une exception est levée lorsqu'une tâche essaie d'accéder à un objet protégé ayant une priorité plafond inférieure à sa priorité. Par conséquent, par défaut, la priorité plafond d'un objet protégé est `priority'last` avec tout ce que cela implique (si l'on ne donne pas soi-même une priorité plafond à un objet protégé, et que l'on utilise le protocole à priorité plafond immédiat, toute tâche accédant à un objet protégé se voit munie de la priorité maximale, hors priorité d'interruption).

■ Ordonnancement des tâches

L'ordonnancement de tâches est par défaut défini par l'implémentation. Cependant, la norme Ada prévoit la possibilité de le modifier. Le type d'ordonnancement choisi pour une application est le même pour toute l'application. La façon de gérer les files

d'attentes au niveau des objets protégés est elle aussi configurable. Typiquement, une application est configurée de sorte que :

– les tâches sont ordonnancées en FIFO suivant leur priorité (ce qui correspond à l'algorithme POSIX SCHED_FIFO), c'est-à-dire qu'il y a une file d'attente gérée en FIFO pour chaque niveau de priorité, ceci est fait en insérant dans le programme la directive `pragma Task_Dispatching_Policy(FIFO_Within_Priorities)` ;
– le protocole à priorité plafond immédiat est utilisé pour éviter les inversions de priorité lors de l'utilisation d'objets protégés, ceci nécessite l'ajout de la directive `pragma Locking_Policy(Ceiling_Locking)` ;
– les files de tâches en attente d'accès à un objet protégé sont gérées dans des files FIFO par niveaux de priorités grâce à la directive `pragma Queuing_Policy (Priority_Queuing)`.

Par conséquent, toute application multitâche Ada devrait utiliser ces trois pragmas.

Le profil Ravenscar

Étant données les limitations des modèles utilisés pour la validation temporelle de systèmes, un concepteur doit, de préférence suivre différentes règles s'il souhaite pouvoir valider temporellement son application.
Pour le langage Ada, une norme de développement a été proposée, sous le nom de profil ***Ravenscar***, normalisé par l'ISO en 1999. La norme préconise les principes suivants :

– pas de déclaration de tâches dynamiques : tout est déclaré au début de l'application ;
– les tâches ne se terminent pas ou toutes ensemble ;
– toutes les tâches sont au même niveau (pas de hiérarchie mère/fille entre les tâches) ;
– tous les outils de communication et de synchronisation sont déclarés initialement, et pas de façon dynamique ;
– utilisation de dates et pas de durées pour les réveils de tâches périodiques, afin d'éviter une dérive des horloges (voir § 2.4) ;
– utilisation d'un protocole de gestion de ressources à priorité plafond ;
– pas d'utilisation du rendez-vous mais uniquement des objets protégés, avec une seule primitive de type `entry` par objet protégé, les autres pouvant être des procédures ou bien des fonctions ;
– un et un seul mécanisme de déclenchement pour une tâche (attente de synchronisation, de message ou d'interruption, ou bien attente périodique).

6.3.2 Implémentation des éléments DARTS

Notons qu'afin d'en améliorer la lisibilité, les exemples d'implémentation d'éléments DARTS sont donnés sous la forme d'un seul programme principal déclarant tous les éléments. Il est évident que pour les implémentations concrètes, les tâches et objets protégés seront déclarés dans des paquetages : généralement, les spécifications

© Dunod – La photocopie non autorisée est un délit.

des tâches et objets protégés sont placées dans les spécifications des paquetages, alors que leurs corps sont placés dans les corps des paquetages.

Modules de données

Les modules de données DARTS peuvent être implémentés à l'aide de l'encapsulation du module de données dans un objet protégé. L'exclusion mutuelle est alors garantie par la non réentrance des primitives d'accès à un objet protégé. Ainsi, une première implémentation de la figure 6.24 est donnée ci-après.

```
with Procede; use Procede;
-- Module contenant des fonctions d'accès aux capteurs/actionneurs
procedure Test_Mdd is
   -- Procédure montrant un exemple de module de données
   type T_Temperature_Pression is record
      -- Type de données contenu dans le module de données
      Pression:Float;
      Temperature: Float;
   end record;

   protected Temperature_Pression is
      -- Module de données
      procedure Ecrire(Tp: T_Temperature_Pression);
      -- Modifie la valeur contenue
      function Lire return T_Temperature_Pression;
      -- Renvoie la valeur contenue
      private
      -- Variables internes
      Valeur: T_Temperature_Pression := (Lire_Capteur_Temperature,
Lire_Capteur_Pression);
      -- Initialisation de la température et de la pression
   end;
   protected body Temperature_Pression is
      -- Implémentation du module de données
      procedure Ecrire(Tp: T_Temperature_Pression) is
      begin
         Valeur:=Tp;
      end;

      function Lire return T_Temperature_Pression is
      begin
         return Valeur;
      end;
   end;
   task Acquisition; -- Spécification de tâche
   task body Acquisition is
      -- Corps de tâche
      TP: T_Temperature_Pression;
   begin
      -- Code d'initialisation des capteurs
      -- etc.
      loop
         -- Lecture des capteurs
         Tp.Temperature:=Lire_Capteur_Temperature;
         Tp.Pression:=Lire_Capteur_Pression;
         Temperature_Pression.Ecrire(Tp); -- Ecriture dans le module
                                          -- de données
         delay(0.5); --Attendre au moins 500 ms
```

```
      end loop;
   end;
   task Traitement; -- Spécification de tâche
   task body Traitement is
      -- Corps de tâche
      TP: T_Temperature_Pression;
   begin
      loop
         Tp:=Temperature_Pression.Lire; -- Lecture dans le module de
                                        -- données
         -- etc.
      end loop;
   end;
begin -- Les tâches sont lancées
   null; -- Le programme principal ne fait rien
end; -- Attente de la terminaison des tâches
```

Cet exemple est correct d'un point de vue respect de l'exclusion mutuelle, cependant, il ne peut pas fonctionner dans la réalité ; la cause réside dans l'initialisation du module de données. Dans la réalité, il est nécessaire d'initialiser le matériel (alimentation, état interne, etc.) avant de pouvoir l'utiliser. Dans cette implémentation, l'initialisation du module de données par lecture des capteurs étant faite en même temps que le démarrage du programme principal, il n'est pas possible d'effectuer une initialisation des capteurs avant de les lire. Il est donc nécessaire de permettre l'initialisation du module après initialisation des capteurs, et donc d'empêcher toute lecture avant la première écriture dans le module. Par conséquent, le code généralement utilisé pour implémenter un module de données est potentiellement bloquant en lecture : la lecture est une procédure gardée (`entry`) possible uniquement après une première écriture.
Le code est donc modifié comme ci-après (noter le changement de la primitive `Lire` en procédure gardée) :

```
with Procede; use Procede;
-- Module contenant des fonctions d'accès aux capteurs/actionneurs
procedure Test_Mdd is
   -- Procédure montrant un exemple de module de données
   type T_Temperature_Pression is record
      -- Type de données contenu dans le module de données
      Pression:Float;
      Temperature: Float;
   end record;

   protected Temperature_Pression is
      -- Module de données
      procedure Ecrire(Tp: T_Temperature_Pression);
      -- Modifie la valeur contenue
      entry Lire(Tp: out T_Temperature_Pression);
      -- TP : paramètre de sortie renvoyant la valeur contenue
      -- Possible seulement après une première écriture
      private
      -- Variables internes
      Valeur: T_Temperature_Pression;
      -- L'initialisation est inutile, puisque la lecture ne peut
avoir lieu qu'après une première écriture
      Initialise : Boolean := False;
   end;
```

© Dunod – La photocopie non autorisée est un délit.

```
   protected body Temperature_Pression is
      -- Implémentation du module de données
      procedure Ecrire(Tp: T_Temperature_Pression) is
      begin
         Valeur:=Tp;
         Initialise := True; -- Ouverture de la garde de la lecture
      end;

      entry Lire (Tp: out T_Temperature_Pression) when Initialise is
         -- La garde est ouverte seulement après une première écriture
      begin
         TP:=Valeur;
      end;
   end;
   task Acquisition;
   task body Acquisition is
      -- etc. identique au code précédent
   end;
   task Traitement;
   task body Traitement is
      -- Corps de tâche
      TP: T_Temperature_Pression;
   begin
      loop
         Temperature_Pression.Lire(TP); -- Lecture dans le module de
                                        -- données
         -- etc.
      end loop;
   end;
begin -- Les tâches sont lancées
   null; -- Le programme principal ne fait rien
end; -- Attente de la terminaison des tâches
```

Afin de faciliter la réutilisation de code, il peut être intéressant d'utiliser la généricité présente dans le langage Ada. Ainsi, on pourrait créer un paquetage générique de module de données, cependant, cela n'est pas conforme à la norme *Ravenscar* (§ 6.3.1, p. 319).

■ Boîtes aux lettres

L'implémentation des boîtes aux lettres par objet protégé est assez similaire à l'implémentation proposée pour POSIX.

□ Boîtes aux lettres de taille 1 avec écrasement

Une boîte aux lettres de taille 1 avec écrasement est simple à implémenter à partir d'un objet protégé. Il suffit de constater que l'envoi de message est non bloquant (emploi d'une procédure) et que l'attente de message est bloquante (emploi d'une procédure gardée). Ainsi, l'implémentation de la figure 6.25 est donnée ci-après.

```
procedure Test_Bal_1_Ecr is
   protected type T_Bal_1_Ecr is
      -- Boîte aux lettres d'entiers de taille 1 avec écrasement
      procedure Envoyer(V: Integer); -- Envoi d'un message dans la
                                     -- boîte aux lettres
      entry Recevoir(V: out Integer); -- Attente d'un message en
                                      -- provenance de la boîte aux
                                      -- lettres
```

```
      private
         Vide : Boolean := True; -- Etat de la boîte, Faux si il y a
                                 -- un message non lu
         Contenu: Integer; -- Message contenu dans la boîte si
                           -- Vide=False
   end;
   protected body T_Bal_1_Ecr is
      procedure Envoyer(V: Integer) is
      begin
         Contenu:=V;
         -- Le contenu reçoit V, même si la boîte était non vide (dans
         -- ce cas, il y a écrasement du message précédent par V)
         Vide:=False; -- Il y a un message non lu
      end;

      entry Recevoir(V: out Integer) when not Vide is
         -- Il n'est possible de recevoir un message que s'il existe
         -- un message non lu
      begin
         -- Lorsque l'on arrive là, c'est qu'il y a un message non lu
         -- (garde vraie)
         V:=Contenu;
         Vide:=True; -- La boîte est maintenant vide
      end;
   end;
   B1:T_Bal_1_Ecr; -- Instanciation d'une boîte aux lettres
   task T1; -- Tâche envoyant un message
   task body T1 is
      msg: Integer;
   begin
      loop
         -- Calculs divers permettant de donner une valeur à msg
         B1.Envoyer(Msg); -- Envoi dans la boîte aux lettres
         -- etc.
      end loop;
   end;
   task T2; -- Tâche attendant un message
   task body T2 is
      Msg_Recu: Integer;
   begin
      loop
         B1.Recevoir(Msg_Recu); -- Attente d'un message,la tâche est
         -- bloquée jusqu'à ce qu'il y ait un message
         -- Ici, Msg_Recu contient le message reçu
         -- etc.
      end loop;
   end;
begin -- Lancement des tâches
   null; -- le programme principal ne fait rien
end;
```

☐ Boîte aux lettres de taille 1 sans écrasement

Une boîte aux lettres de taille 1 sans écrasement diffère d'une boîte aux lettres avec écrasement dans le fait que l'envoi de message dans la boîte aux lettres est potentiellement bloquant. La différence au niveau implémentation réside uniquement dans le fait que l'envoi est une procédure gardée (`entry`). Ainsi, par rapport à l'implé-

© Dunod – La photocopie non autorisée est un délit.

mentation de la boîte aux lettres à écrasement, seule l'implémentation de la primitive recevoir change. Le code modifié de l'objet protégé est donné ci-après.

```
protected type T_Bal_1 is
   -- Boîte aux lettres d'entiers de taille 1 avec écrasement
   entry Envoyer(V: Integer); -- Envoi d'un message dans la boîte aux
                              -- lettres
   entry Recevoir(V: out Integer); -- Attente d'un message en
                                   -- provenance de la boîte aux
                                   -- lettres
   private
      Vide : Boolean := True; -- Etat de la boîte, Faux si il y a un
                              -- message non lu
      Contenu: Integer; -- Message contenu dans la boîte si Vide=False
end;
protected body T_Bal_1 is
   entry Envoyer(V: Integer) when Vide is
      -- Il n'est possible d'envoyer un message que si la boîte est
      -- vide
   begin
      -- Lorsque l'on arrive là, c'est que la boîte est vide, il n'y a
      -- donc pas écrasement
      Contenu:=V;  -- Le contenu reçoit V
      Vide:=False; -- Il y a un message non lu
   end;

   entry Recevoir(V: out Integer) when not Vide is
      -- Il n'est possible de recevoir un message que s'il existe un
      -- message non lu
   begin
      -- Lorsque l'on arrive là, c'est qu'il y a un message non lu
      -- (garde vraie)
      V:=Contenu;
      Vide:=True; -- La boîte est maintenant vide
   end;
end;
```

□ Autres boîtes aux lettres

Le lecteur trouvera en annexe D l'extension des boîtes aux lettres de taille 1 avec et sans écrasement au cas des boîtes aux lettres de taille *n* (figure 6.27). La file de messages est implémentée par un tableau circulaire. La prise en compte d'une priorité de messages peut s'effectuer de deux façons : soit le nombre de priorités est élevé, dans ce cas la gestion de la file d'attente consiste en un tri par insertion des messages, ce qui est coûteux, soit il n'y a que deux niveaux de priorité (normal et urgent) et on peut créer deux files de messages.

■ Synchronisation

L'implémentation de la synchronisation est identique à l'implémentation du sémaphore à compte présentée au paragraphe 6.3.1, p. 317. Le code, placé dans un paquetage, est donné ci-après.

```
package Synchronisation is
   protected type T_Synchronisation is
      -- Implémentation d'une synchronisation à compte
      entry Wait; -- Attente de synchronisation
```

```
      procedure Signal; -- Déclenchement de synchronisation
   private
      Valeur: Natural:=0; -- Nombre de synchronisation non prises en
                          -- compte
   end T_Synchronisation;
end Synchronisation;

package body Synchronisation is
   protected body T_Synchronisation is
      entry Wait when Valeur > 0 is
      -- Tant qu'il n'y a pas de déclenchement non pris en compte,
      -- les tâches appelant cette entrée sont bloquées et mises en
      -- attente dans la file d'attente de l'entrée
      begin
         Valeur:=Valeur-1;
      end Wait;
      procedure Signal is
      begin
         Valeur:=Valeur+1;
      end Signal;
   end T_Synchronisation;
end Synchronisation;
```

L'utilisation d'une synchronisation sur l'implémentation de la figure 6.28 est de la forme suivante :

```
with Synchronisation; use Synchronisation;
procedure Test_Synchronisation is
   S1: T_Synchronisation; -- Instanciation d'une synchronisation
   task T1;
   task body T1 is
      -- Tâche déclenchant T2 par synchronisation
   begin
      -- etc.
      loop
         -- etc.
         S1.Signal;
         -- etc.
      end loop;
   end;
   task T2;
   task body T2 is
      -- Tâche en attente de synchronisation
   begin
      -- etc.
      loop
         S1.Wait;
         -- etc.
      end loop;
   end;
begin -- Toutes les tâches sont lancées
   null; -- Le programme principal ne fait rien
end; -- Attente de la terminaison des tâches
```

■ Tâches périodiques

Ada propose deux primitives d'attente : une attente relative (`delay`) présente dès Ada 83, et une attente de date (`delay until`) présente depuis Ada 95. Afin d'éviter

© Dunod – La photocopie non autorisée est un délit.

une dérive des horloges (§ 5.3.1, p. 216), il convient d'utiliser l'attente de date. Ainsi, l'implémentation des tâches de la figure 6.29 est :

```
with Ada.Real_Time;use Ada.Real_Time;
with Ada.Text_IO;use Ada.Text_IO;
procedure Test_Periodique is
   task type T_Periodique(Numero: Integer; Periode_Ms: Integer);
   -- Type de tâche périodique affichant son numéro à chaque itération
   -- Numéro : numéro de la tâche
   -- Periode_Ms: Période en millisecondes
   task body T_Periodique is
      Periode : constant Time_Span := Milliseconds(Periode_Ms);
      -- Le type Time_Span, défini dans Ada.Real_Time, permet de
      -- représenter une durée
      Prochaine_Activation: Time := Clock+Periode;
      -- Le type Time, défini dans Ada.Real_Time, représente une date
      -- Date de prochaine activation, initialisée à l'heure courante
      -- (retour de la fonction clock) + la période
   begin
      loop
         Put_Line(integer'image(Numero));  -- Affichage du numéro
         delay until Prochaine_Activation; -- Attente de la prochaine
                                           -- date d'activation
        Prochaine_Activation:=Prochaine_Activation+Periode; -- Calcul
                                    -- de la prochaine date de réveil
      end loop;
   end;
   Periode1 : T_Periodique(1, 100); -- Instanciation d'une tâche de
                                    -- période 100 ms
   Periode2 : T_Periodique(2, 150); -- Instanciation d'une tâche de
                                    -- période 150 ms
begin -- lancement de toutes les tâches
   null; -- le programme principal ne fait rien
end; -- Attente de terminaison des tâches
```

■ Tâches réveillées par interruption

Ada prévoit directement une gestion d'interruption de type ISR déclenchant une DSR (§ 5.2.3, p. 206). Une ISR doit forcément se trouver dans un objet protégé situé dans un paquetage. Le protocole à priorité plafond immédiat doit être utilisé, et la priorité plafond de l'objet protégé contenant l'ISR doit être une priorité d'interruption.

Ainsi, la tâche matérielle présentée sur la figure 6.30 s'implémente comme suit. L'interruption déclenchée par la combinaison de touches Contrôle-C, nommée SIGINT d'après une terminologie Unix/POSIX est traitée.

```
-- Fichier Isr.ads
with System; use System;
package Isr is
   protected Isr2 is
      -- Objet protégé gérant l'interruption 2 (SIGINT)
      entry Wait; -- Procédure gardée sur laquelle la tâche de
                  -- traitement (DSR) se met en attente de
                  -- l'interruption
      procedure Signal; -- Procédure effectivement appelée lors de
                        -- l'interruption traitée
   private
```

```
      pragma Interrupt_Priority(Interrupt_Priority'Last); -- Doit
s'exécuter avec la priorité maximale
      pragma Attach_Handler(Signal,2); -- Ce pragma fait que la
             -- procédure Signal est appelée sur l'interruption SIGINT
      Pending: Natural:=0; -- Nombre d'interruptions non traitées
   end;
end Isr;

-- Fichier Isr.adb
package body Isr is
      protected body Isr2 is
      entry Wait when Pending>0 is -- La DSR se bloque ici tant qu'il
                                -- n'y a pas d'interruption à traiter
      begin
         Pending:=Pending - 1; -- Une interruption est traitée
      end;
      procedure Signal is
      begin
         Pending := Pending + 1; -- Une interruption supplémentaire est
                                -- à traiter
      end;
   end;
end Isr;

-- Fichier Test_Interruption.adb (programme principal)
-- pragmas nécessaires dès lors que l'ordonnancement a une importance
pragma Task_Dispatching_Policy(Fifo_Within_Priorities);
-- Ordonnancement FIFO par niveaux de priorité
pragma Locking_Policy(Ceiling_Locking); -- Protocole à priorité
                                        -- plafond immédiat
pragma Queuing_Policy(Priority_Queuing); -- Gestion des attentes
                           -- d'objet protégé par priorité des tâches
with Text_Io;use Text_Io;
with Isr;use Isr;
procedure Test_Interruption is
   task Dsr is
      -- Tâche de traitement de l'interruption
      pragma Priority(25);
      -- Tâche de priorité 25
   end;
   task body Dsr is
      -- Tâche de traitement de l'interruption
   begin
      put_line("Appuyer sur Ctrl-C");
      loop
         Isr2.Wait; -- Attente d'une interruption
         -- etc.
         Put_Line("Ctrl-C");
      end loop;
   end;
begin -- lancement de la tâche
   null; -- ne fait rien
end; -- Attente de terminaison
```

6.3.3 Exemple

L'exemple de la « gestion de la sécurité d'une mine » dont le diagramme DARTS est donné sur la figure 3.27 est traité en langage Ada.

© Dunod – La photocopie non autorisée est un délit.

Commençons par le plus simple, à savoir le programme principal, qui est de la forme :

```
-- pragmas nécessaires dès lors que l'ordonnancement a une importance
pragma Task_Dispatching_Policy(Fifo_Within_Priorities);
-- Ordonnancement FIFO par niveaux de priorité
pragma Locking_Policy(Ceiling_Locking); -- Protocole à priorité
                                        -- plafond immédiat
pragma Queuing_Policy(Priority_Queuing); -- Gestion des attentes
                           -- d'objet protégé par priorité des tâches
with Controle;  -- Paquetage où les tâches sont déclarées
procedure Controle_Mine is
begin -- lancement des tâches (déclarées dans le paquetage Controle)
   null; -- ne fait rien
end; -- Attente de terminaison
```

Ce programme principal « vide » est assez caractéristique des programmes multitâches en langage Ada.
Nous supposons qu'il existe un paquetage générique `Communication` donné en annexe D. Ce paquetage générique doit être instancié sur les types de données manipulées. Il est donc instancié sur les nombres flottants, sous la forme d'un paquetage nommé `com_float` :

```
with Communications; -- Paquetage générique de communication
package Com_Float is new Communications(Element=>Float);
```

Le paquetage contenant les tâches et les instanciations des éléments de communication et de synchronisation est donné ci-après.

```
-- Fichier Controle.ads
with Com_Float; -- Définit les outils de communication
-- Noter l'absence de la clause use, obligeant à préfixer
-- l'utilisation des éléments de ce paquetage par le nom du paquetage
-- Cette écriture lève toute ambiguïté lorsque plusieurs types du même
-- nom sont définis dans des paquetages différents
package Controle is
   -- Spécifie et instancie les tâches du contrôle de mine
   -- Instancie les boîtes aux lettres, synchronisations et modules de
données nécessaires

   -- Déclaration et instanciation des tâches
   task Acquerir_Niveau_Methane is
      pragma Priority(12);
   end;
   task Acquerir_Capteur_Eau is
      pragma Priority(11);
   end;
   task Afficher_Alarme is
      pragma Priority(28);
   end;
   task Commander_Pompe is
      pragma Priority(29);
   end;
   task Controler_Mine is
      pragma Priority(30);
   end;

   MDD_Niveau_Eau : Com_Float.MDD; -- Instanciation d'un module de
données contenant un float
```

```
   Bal_Niveau_Methane : Com_Float.Bal_1_Ecrasement; -- Instanciation
                                 -- d'une boîte aux lettres de float
   Bal_Vitesse : Com_Float.Bal_1_Ecrasement; -- Instanciation d'une
                                      -- boîte aux lettres de float
   Sync_Alarme : Com_Float.Synchro_C;
end;
```

Le corps du paquetage, contenant l'implémentation des tâches, est fourni ci-après.

```
-- Fichier Controle.adb
with Procede; use Procede; -- Définit les fonctions d'accès aux
                           -- capteurs/actionneurs
with Ada.Real_Time;use Ada.Real_Time;
package body Controle is
   -- Définitions des constantes
   LLS : constant float:= 5.3; -- Valeur sous laquelle le niveau est
                               -- considéré trop bas
   HLS : constant float := 17.8; -- Valeur au-dessus de laquelle le
                                 -- niveau est considéré trop haut

   -- Seuils D'Alerte De Méthane
   MS_L1 : constant float:= 128.0;
   MS_L2 : constant float:=180.0;

   task body Acquerir_Capteur_Eau is
      --Tâche périodique d'acquisition
      Periode: constant Time_Span:=Milliseconds(5000); -- Période de 5s
      Prochaine: Time:=Clock+Periode; -- Prochaine date de réveil
   begin
      loop
         MDD_Niveau_Eau.Ecrire(Lire_Capteur_Eau); -- Ecrit la valeur
                                          -- du capteur dans le MDD
         delay until Prochaine; -- Attend la prochaine date de réveil
         Prochaine:=Prochaine+Periode; -- Calcul de la prochaine date
                                       -- de réveil
      end loop;
   end Acquerir_Capteur_Eau;

   task body Acquerir_Niveau_Methane is
      --Tâche périodique d'acquisition
      Periode: constant Time_Span:=Milliseconds(500); -- Période
                                                      -- de 500 ms
      Prochaine: Time:=Clock+Periode; -- Prochaine date de réveil
   begin
      loop
         Bal_Niveau_Methane.Envoyer(Lire_Capteur_Methane); -- Envoi
                                         -- de la valeur du capteur
         delay until Prochaine; -- Attend la prochaine date de réveil
         Prochaine:=Prochaine+Periode; -- Calcul de la prochaine date
                                       -- de réveil
      end loop;
   end Acquerir_Niveau_Methane;

   task body Afficher_Alarme is
      -- Tâche en attente de synchronisation
    Etat_Alarme: Boolean := False; -- Etat courant de l'alarme (vrai
                                   -- ssi allumée)
   begin
      loop
         Sync_Alarme.Wait; -- Attente de synchronisation
```

© Dunod – La photocopie non autorisée est un délit.

```
         Etat_Alarme:= not Etat_Alarme;
         Piloter_Alarme(Etat_Alarme); -- Actionne ou coupe l'alarme
      end loop;
   end Afficher_Alarme;

   task body Commander_Pompe is
      -- Tâche en attente de message
      Vitesse : float;
   begin
      loop
         Bal_Vitesse.Recevoir(Vitesse);
         Piloter_Pompe(Vitesse); --Pilotage effectif de la pompe
      end loop;
   end Commander_Pompe;

   task body Controler_Mine is
      Niveau_Methane, Niveau_Eau: Float;
      type T_Etat is (Nominal, Pompe, Alerte_Et_Pompe,  Alerte);
                    -- Etats possibles du diagramme Etats/Transitions
      Etat: T_Etat:=Nominal; -- Etat courant dans le diagramme états/
                             -- transitions
   begin
      loop
         Bal_Niveau_Methane.Recevoir(Niveau_Methane); -- Attente sur
                                                      -- boîte aux lettres
         MDD_Niveau_Eau.Lire(Niveau_Eau);
        -- Implémentation du diagramme Etats/Transitions donné sur la
        -- figure 2.33
         case Etat is
         when Nominal =>
            if Niveau_Methane >= MS_L1 then
            -- Seuil d'alerte
                Sync_Alarme.Signal; -- Alarme
                Etat := Alerte;
            elsif Niveau_Eau >= HLS then
             -- Allumage de la pompe, pour simplifier, nous supposons
             -- que nous la pilotons de façon proportionnelle
               Bal_Vitesse.Envoyer(Niveau_Eau-LLS);
               Etat := Pompe;
            end if;
         when Pompe =>
            if Niveau_Methane >= MS_L1 then
            -- Seuil d'alerte
                Sync_Alarme.Signal; -- Alarme
                Etat := Alerte_Et_Pompe;
            elsif Niveau_Eau <= LLS then
                Bal_Vitesse.Envoyer(0.0);
                Etat := Nominal;
else Bal_Vitesse.Envoyer(Niveau_Eau-LLS) ; -- Pilotage proportionnel
                                           -- de la pompe
            end if;
         when Alerte =>
            if Niveau_Methane < MS_L1 then
            -- Seuil d'alerte
                Sync_Alarme.Signal; -- Extinction de l'alarme
                Etat := Nominal;
            elsif Niveau_Methane < MS_L2 and Niveau_Eau > HLS then
            -- Allumage de la pompe, pour simplifier, nous supposons
            -- que nous la pilotons de façon proportionnelle
```

```
               Bal_Vitesse.Envoyer(Niveau_Eau-LLS);
               Etat := Alerte_Et_Pompe;
            end if;
         when Alerte_Et_Pompe =>
            if Niveau_Methane >= MS_L2 or niveau_eau <= LLS then
            -- Alerte MS_L2 ou niveau bas => extinction de la pompe
                Bal_Vitesse.Envoyer(0.0);
                Etat := Alerte;
            elsif Niveau_Methane < MS_L1 then
            -- Niveau de méthane sous le seuil d'alerte
                Sync_Alarme.Signal; -- Extinction de l'alarme
                Etat := Pompe;
else Bal_Vitesse.Envoyer(Niveau_Eau-LLS); -- pilotage proportionnel
            end if;
         end case;
      end loop;
   end Controler_Mine;
end Controle;
```

6.4 Programmation multitâche en langage LabVIEW

6.4.1 Implémentation multitâche LabVIEW

La nature flots de données du langage LabVIEW induit naturellement du parallélisme, ainsi, deux boucles non reliées par un fil de données peuvent être exécutées parallèlement. Le parallélisme est donc totalement naturel et transparent pour le programmeur.
En LabVIEW, une tâche peut donc être implémentée simplement à l'aide d'une boucle, et le fait de placer plusieurs « tâches » en parallèle implique naturellement une exécution parallèle. Ainsi, la figure 6.31 montre deux boucles sans fin LabVIEW s'exécutant en parallèle.

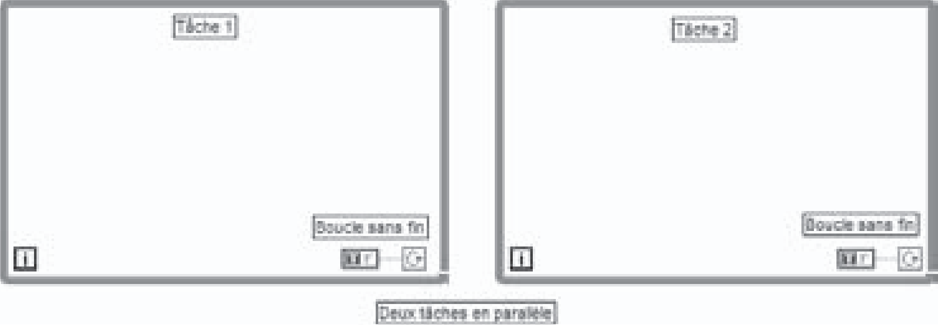
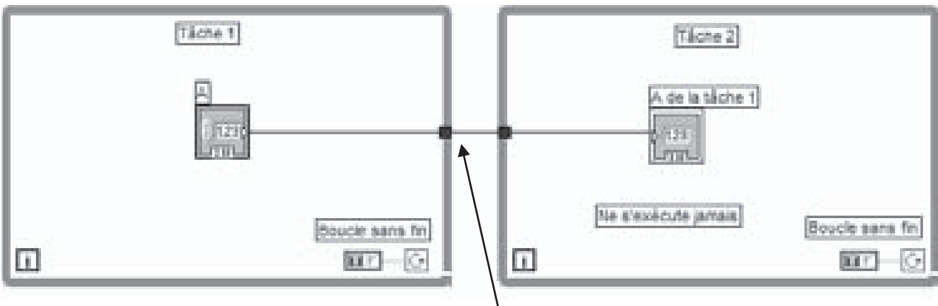


Figure 6.31 – Deux tâches sans fin en langage LabVIEW.

La difficulté vient du fait que si l'on désire faire communiquer les tâches, on ne peut pas utiliser de flot de données (figure 6.32), sinon, une boucle dépend de l'autre, et ne pourra commencer son exécution que lorsque la boucle précédente sera terminée : la tâche 2 n'est jamais exécutée, car elle doit attendre la terminaison de la tâche 1.

© Dunod – La photocopie non autorisée est un délit.



Le flot ne sort qu'à la terminaison de la tâche 1, c'est-à-dire jamais

Figure 6.32 – Programmation incorrecte de la communication entre tâches en langage LabVIEW.

Afin de régler ce type de problème, LabVIEW propose différents outils de communication et synchronisation (boîtes aux lettres, sémaphore, non réentrance des *vi*, rendez-vous sans données, événements) qui peuvent être utilisés pour implémenter les diagrammes DARTS.

Par exemple, la figure 6.33 montre comment on peut utiliser une boîte aux lettres pour réaliser le passage d'un entier entre deux tâches :

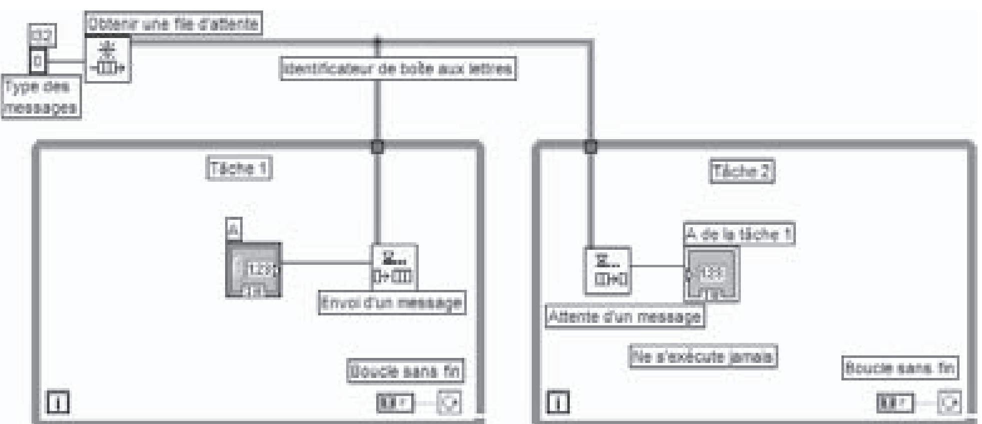


Figure 6.33 – Passage de données entre deux tâches en langage LabVIEW.

Afin d'augmenter la lisibilité du programme, et de profiter du support multitâche du système d'exploitation sous-jacent (§ 6.4.2), on préférera encapsuler chaque tâche dans un sous-*vi* (figure 6.34). Chaque *vi* contenant une tâche peut être configuré afin de modifier sa priorité (il existe 5 niveaux de priorité) et son système d'exécution. Un 6e niveau de priorité appelé « sous-programme » rend un *vi* de cette priorité non préemptible dans son *thread* (§ 6.4.2). Par conséquent, les opérations bloquantes ou suspensives sont interdites dans les *vi* de priorité « sous-programme », de plus, il ne peut appeler que des sous-*vi* de la même priorité que lui.

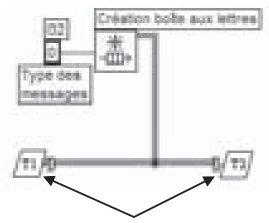


Figure 6.34 – Encapsulation de tâches en langage LabVIEW.

6.4.2 Gestion interne du multitâche par LabVIEW

Nous allons décrire la façon dont LabVIEW gère le multitâche de façon interne.

LabVIEW s'appuie si possible sur le système d'exploitation sous-jacent pour gérer les tâches : si celui-ci est multitâche (c'est le cas de la plupart des systèmes d'exploitation modernes), alors LabVIEW exécute le contenu des *vi* en les intégrant à un certain nombre de tâches du système (nommées *threads* dans la suite de ce paragraphe). Si le système d'exploitation sous-jacent n'est pas multitâche, ou bien **multitâche coopératif** (les tâches ne sont pas préemptées mais rendent elles-mêmes la main au système), alors LabVIEW s'occupe d'exécuter parallèlement les *vi* en multitâche coopératif (cas de Windows 3.1, MacOS avant Mac OS X, etc.).

Afin d'expliquer la façon dont LabVIEW gère le multitâche, il nous faut décomposer les tâches en deux niveaux : les tâches du système d'exploitation ou *threads*, et les tâches internes. Sur un système d'exploitation préemptif, les *threads* sont exécutés de façon préemptive (leur ordonnancement est effectué par le système d'exploitation lui-même), et gèrent une file d'attente de tâches internes, gérées de façon coopérative par le moteur d'exécution LabVIEW. Lorsqu'une tâche interne fait appel à du code externe (librairie dynamique partagée, code dynamique chargé *CIN*, etc.), elle devient non préemptible par une tâche interne du même *thread* jusqu'à la fin de l'appel.

Les *threads* sont regroupés en systèmes d'exécution. Il en existe 6, nommés interface utilisateur, standard, E/S d'instruments, acquisition de données, autre 1 et autre 2. Le premier système d'exécution possède au moins un *thread* qui est dédié à la gestion de l'interface graphique (affichage, positionnement/dimensionnement de fenêtre, gestion des événements utilisateur, etc.).

Les 5 autres systèmes d'exécutions contiennent de 0 à 40 *threads* chacun. En effet, chaque système d'exécution peut être configuré (avec le *vi threadconfig.vi* pouvant se trouver dans la bibliothèque *sysinfo.llb*) pour utiliser de 0 à 8 *threads* par niveau de priorité (il y a 5 niveaux de priorité). Par défaut, LabVIEW 7, sur le système d'exploitation MS Windows XP® utilise 4 *threads* par niveau de priorité (sauf pour le niveau le plus bas) pour chaque système d'exécution, soit en tout 16 par système d'exécution, ce qui donne par défaut un total de 81 *threads* (en incluant le *thread* d'interface graphique), pouvant être porté à 201.

Les *threads* sont ordonnancés par le système d'exploitation en fonction de leur niveau de priorité. À l'intérieur d'un *thread*, les tâches internes (qui correspondent au surplus

© Dunod – La photocopie non autorisée est un délit.

de parallélisme du diagramme n'ayant pas pu être mis seul dans un *thread*) sont ordonnancées en tourniquets classés par priorité. Notons que le *thread* en charge de l'interface graphique gère les interfaces graphiques des *vi* en fonction de leur niveau de priorité.

Étant donné que la priorité est définie au niveau de chaque *vi* (par défaut, c'est la priorité du *vi* appelant), lorsqu'un *vi* très prioritaire appelle un sous-*vi* moins prioritaire, celui-ci hérite de la priorité de l'appelant.

Notons que le nom donné aux systèmes d'exécution autres que celui de l'interface graphique est purement indicatif. Par défaut, un *vi* s'exécute dans le système d'exécution standard. Ceci peut être configuré différemment pour chaque *vi* (menu Fichier, Propriétés du *vi*, exécution).

Le nombre de *threads* contenus dans le seul système d'exécution standard exempte généralement les programmeurs d'utiliser des systèmes d'exécution autres que standard.

6.4.3 Implémentation des éléments DARTS

■ Tâches périodiques

LabVIEW possède deux *vi* d'attente de la granularité d'une milliseconde : une attente relative (`attendre`) et une attente de multiple d'horloge (`attendre un multiple de`). Afin d'éviter une dérive des horloges (§ 5.2.4), c'est le second type d'attente qui est utilisé pour assurer la périodicité des réveils d'une tâche : ce *vi* attend que l'horloge soit un multiple de son paramètre. Ainsi, sur la figure 6.35, soit t_1 la date à laquelle la première itération de la boucle a lieu. Le *vi* `attendre un multiple de` 50 ms attend que la valeur d'horloge ait atteint un multiple de 50 ms. La seconde itération, ne pouvant commencer que lorsque l'itération précédente est terminée, doit attendre la fin de tous les *vi* et donc la fin de `attendre un multiple de`, et donc le prochain multiple de 50 ms (notons que la seconde itération peut donc commencer moins de 50 ms après la première en fonction de la valeur de l'horloge lors de la première itération). Soit *50xn* ms cette date. Alors la troisième itération ne peut avoir lieu qu'à partir de la date $50(n + 1)$, la $i^{\text{ème}}$ itération ne peut pas débuter avant la date $50(n + i - 1)$, etc. Notons que si une itération « déborde » de sa période, l'itération

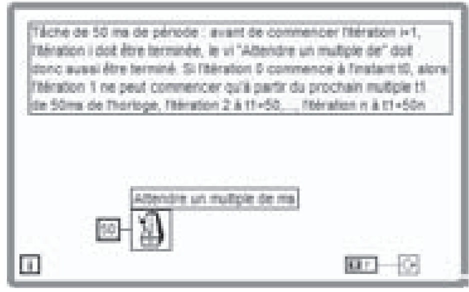


Figure 6.35 – Tâche périodique en langage LabVIEW.

suivant est repoussée : si l'itération i se termine après la date $50(n + i)$, alors l'itération $i + 1$ ne peut commencer qu'à la date $50(n + i - 1)$ au lieu de $50(n + i)$.

Modules de données

Les modules de données DARTS peuvent être implémentés en se basant sur la non réentrance des *vi* : par défaut, un *vi* est non réentrant. Afin d'illustrer plus aisément son fonctionnement, un parallèle est fait entre *vi* non réentrant et moniteur (par exemple objet protégé Ada).

Lorsque deux tâches tentent d'accéder simultanément à une primitive d'un même moniteur, deux accès simultanés au même *vi* non réentrant ne peuvent pas avoir lieu. Dans un moniteur, il y a des variables internes (partie `private` en langage Ada) représentant l'état du moniteur. En LabVIEW, le registre à décalage (figure 6.11) sert à conserver des données d'une itération de boucle pour la suivante. Lorsque la boucle se termine, les dernières données inscrites restent mémorisées dans le registre à décalage. Par conséquent, lors de la prochaine lecture du registre (lors d'une exécution suivante de la boucle), si le registre à décalage n'est pas initialisé, il contient les données inscrites dans le registre à décalage. L'idée consiste donc à créer une boucle n'ayant qu'une seule itération afin de la munir d'un registre à décalage qui mémorise les données inscrites d'une exécution sur l'autre. Cette méthode permet de conserver de façon rémanente des données dans un *vi*.

Ainsi, la figure 6.36 montre un module de données contenant un enregistrement (`cluster`) pouvant être lu et modifié par des tâches concurrentes tout en respectant l'exclusion mutuelle. Généralement, trois primitives de base sont définies : initialiser, lire et écrire. La primitive est choisie à l'aide d'un type énuméré passé en paramètre au *vi*. La figure 6.37 montre l'utilisation de ce module de données pour la lecture par un *vi* périodique.

Notons qu'il est assez courant de diversifier les primitives : ainsi, si par exemple on souhaite pouvoir modifier uniquement la température ou uniquement la pression, il suffit d'ajouter les primitives permettant de le faire.

Malheureusement, la comparaison entre moniteur Ada et *vi* non réentrant s'arrête là : il n'y a aucun moyen simple d'implémenter une procédure gardée (`entry` Ada) avec cette technique.

Boîtes aux lettres

Depuis la version 5, LabVIEW propose des boîtes aux lettres sans écrasement bornées ou non ; dans ce cas, le nombre de messages est limité par la mémoire, permettant une lecture bloquante, ou bien non bloquante (en utilisant un *timeout* nul ou borné en temps). Depuis la version 7, les boîtes aux lettres sont polymorphes, c'est-à-dire que le type des messages est donné à la création, alors qu'auparavant, les messages étaient de type chaîne de caractère (ce qui correspond au fonctionnement des boîtes aux lettres en langage C, obligeant à effectuer des coercions vers `char *` pour l'envoi et depuis `char *` pour la réception de messages).

© Dunod – La photocopie non autorisée est un délit.

Figure 6.36 – Module de données en langage LabVIEW.

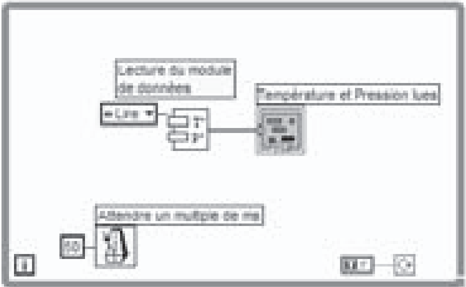


Figure 6.37 – Utilisation d'un module de données en langage LabVIEW.

☐ Boîtes aux lettres sans écrasement

Un exemple de boîte aux lettres non bornée sans écrasement a été présenté sur les figures 6.32 et 6.33. La figure 6.38 montre un exemple de boîte aux lettres de taille 1 sans écrasement. Par rapport à la figure 6.34, sur laquelle la boîte était non bornée, seul un paramètre change à la création de la boîte aux lettres.

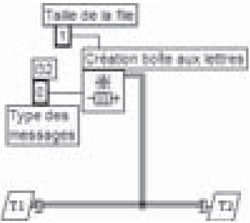


Figure 6.38 – Boîte aux lettres de taille 1 sans écrasement en langage LabVIEW.

☐ Boîtes aux lettres à écrasement

Si cela est nécessaire, il est possible de créer des boîtes aux lettres à écrasement en utilisant les sémaphores et une variable globale en utilisant un algorithme de type producteur à écrasement/consommateur. Cependant, l'implémentation de ce type de boîte aux lettres reste laborieuse en langage LabVIEW.

■ Synchronisation

LabVIEW propose l'outil sémaphore à compte, ce qui devrait rendre triviale l'implémentation des synchronisations. C'était le cas avant la version 7 de LabVIEW Ainsi, la figure 6.39 propose un exemple de synchronisation fonctionnant sur LabVIEW

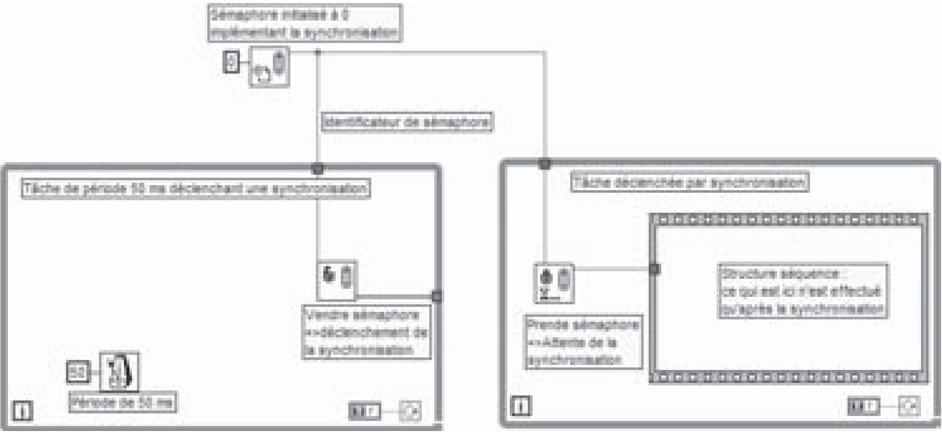


Figure 6.39 – Synchronisation en langage LabVIEW en version 5 et 6.

© Dunod – La photocopie non autorisée est un délit.

5 et 6. Afin de simplifier sa présentation, les tâches ne sont pas encapsulées dans des sous-*vis*.
Malheureusement, LabVIEW 7 a légèrement modifié la sémantique des sémaphores pour n'en faire que des sémaphores d'exclusion mutuelle. Par conséquent, la synchronisation pourra être implémentée en LabVIEW 7 par des boîtes aux lettres non bornées avec des messages d'un octet (par exemple booléens) n'ayant de signification que par leur présence, et non par leur valeur.

6.4.4 Exemple

La mise en œuvre des différents outils présentés pour l'implémentation d'éléments DARTS en langage LabVIEW est faite sur l'exemple de la « gestion de la sécurité d'une mine » dont le diagramme DARTS est donné sur la figure 3.27. Dans cette implémentation, la synchronisation entre la tâche de contrôle et la tâche d'alarme est remplacée par une communication par boîte aux lettres.
Nous pouvons constater que LabVIEW dispose d'une moindre puissance d'expression que POSIX ou Ada en terme de synchronisation et de communication, mais, en contrepartie, que l'implémentation d'une conception DARTS est extrêmement simple à réaliser. Le module de données « Niveau Eau » est implémenté comme le module de données de la figure 6.36, excepté qu'il contient une donnée interne de type flottant.

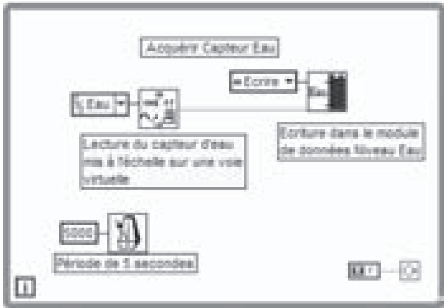


Figure 6.40 – Implémentation de la tâche « Acquérir Capteur Eau » en langage LabVIEW.

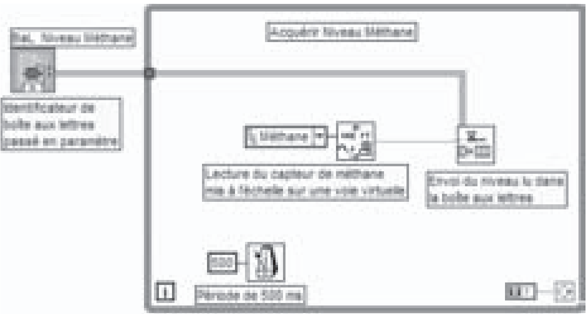


Figure 6.41 – Implémentation de la tâche « Acquérir Niveau Méthane » en langage LabVIEW.

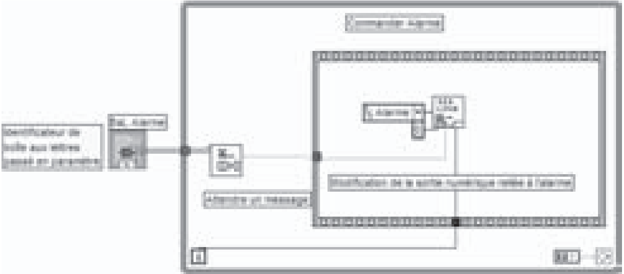


Figure 6.42 – Implémentation de la tâche « Commander Alarme » en langage LabVIEW.

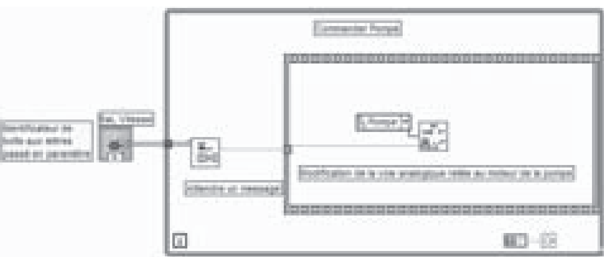


Figure 6.43 – Implémentation de la tâche « Commander Pompe » en langage LabVIEW.

© Dunod – La photocopie non autorisée est un délit.

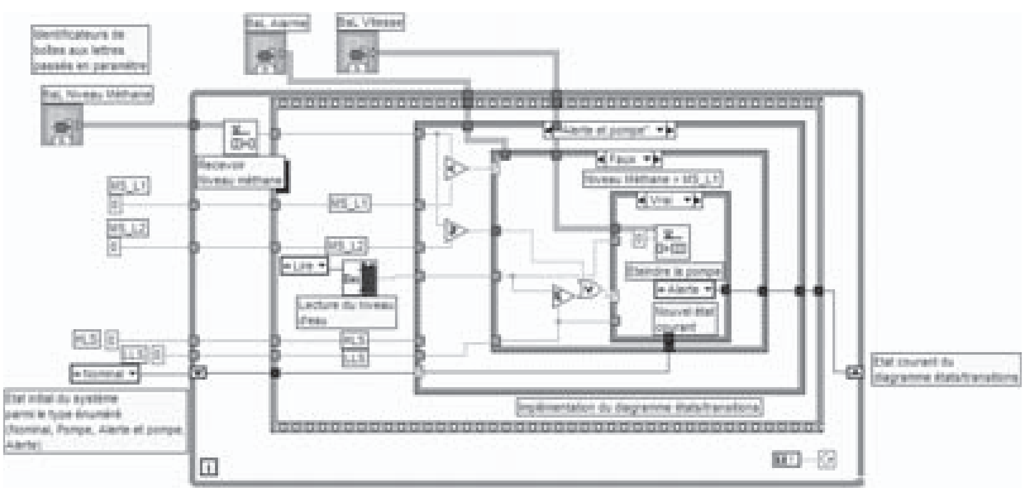


Figure 6.44 – Vue partielle de l'implémentation de la tâche « Contrôler Mine » en langage LabVIEW.

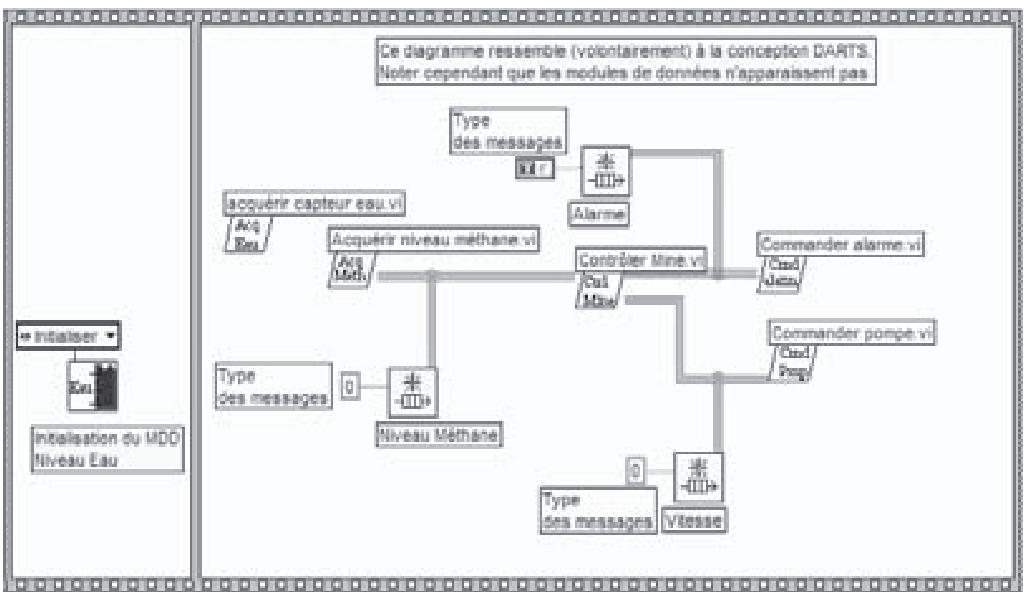


Figure 6.45 – Implémentation du contrôle d'une mine en langage LabVIEW.

7 • TRAITEMENT COMPLET D'UNE APPLICATION INDUSTRIELLE

Ce chapitre illustre par un exemple de taille réaliste la méthodologie présentée dans cet ouvrage.

Un échangeur de chaleur, procédé encombrant, pourra être contrôlé à partir d'un micro-ordinateur standard ou d'un PC industriel. Les contraintes de temps étant relativement molles, on peut utiliser un système d'exploitation généraliste (Linux, MS Windows®…) ou bien un système d'exploitation temps réel d'encombrement mémoire important, qui pourra être conforme au type 54 de la norme (par exemple RTLinux et la famille des systèmes temps réel basés sur Linux), ou encore un système temps réel propriétaire comme VxWorks®. Pour illustrer tout cela, les langages cibles abordés lors de cet exemple sont LabVIEW pour une implémentation sur système d'exploitation généraliste, et C (POSIX) et Ada pour une implémentation sur système d'exploitation généraliste ou bien temps réel.

7.1 Cahier des charges

Un échangeur de chaleur (figure 7.1), est un dispositif très utilisé dans le domaine agroalimentaire. L'une de ses utilisations consiste à refroidir un procédé évacuant de l'énergie sous forme de chaleur. L'échangeur peut être constitué de deux circuits de flux indépendants (dans le sens où il n'y a pas d'échange liquide) mais séparés par une paroi métallique favorisant les transferts thermiques.

Une pompe entraîne un flux continu d'eau distillée à l'intérieur d'un circuit fermé. Le procédé à refroidir (qui, lui, n'est pas piloté par le système à concevoir) est immergé dans une cuve irriguée par l'eau distillée. L'eau distillée est elle-même refroidie par échange thermique avec un flux d'eau industrielle en circuit ouvert. Ainsi, de l'eau « fraîche » (aux alentours de 15 à 21 °C en fonction de la saison) permet de refroidir l'eau distillée, qui elle-même maintient le procédé autour d'une température de consigne.

Le but du système de contrôle est de réguler le débit d'eau industrielle en fonction des températures mesurées, tout en assurant la sécurité du système.

Une console opérateur sert à afficher les relevés des températures (entrée et sortie d'eau distillée, entrée et sortie d'eau froide) et des débits, et une alarme est déclenchée si la température relevée en sortie de cuve est hors domaine (la température référence *T°ed2* en sortie de cuve doit se situer dans une fourchette donnée). De même, si

© Dunod – La photocopie non autorisée est un délit.

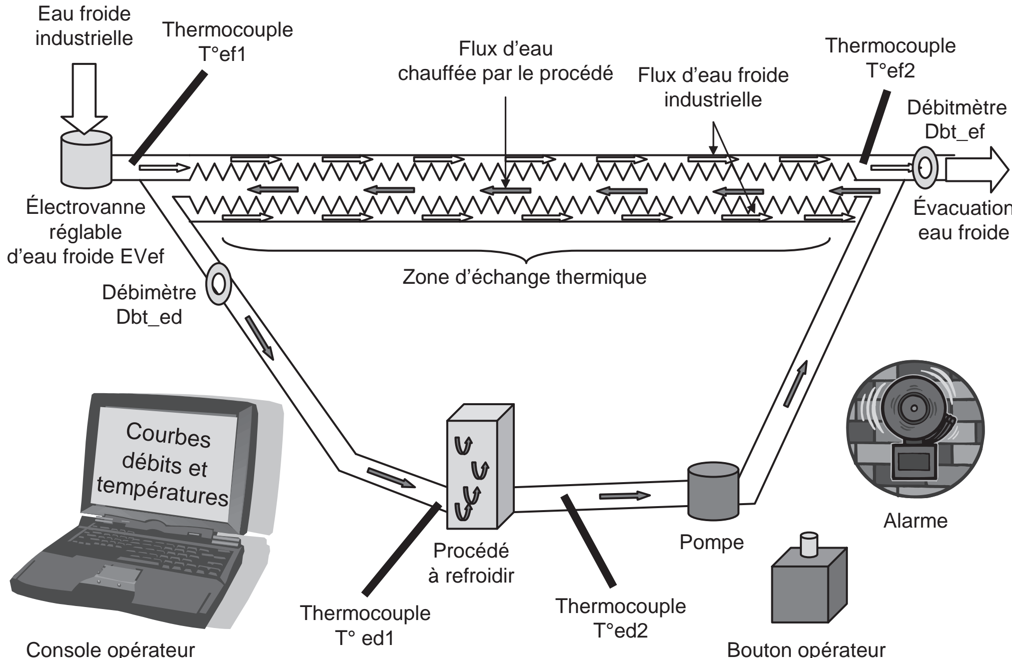


Figure 7.1 – Échangeur de chaleur.

l'on s'aperçoit de la présence d'une fuite dans l'un des circuits, ou de l'occurrence d'une coupure d'eau industrielle, ou encore d'une panne de pompe, alors l'alarme doit être déclenchée. Dans ce cas, un opérateur humain doit stopper le procédé refroidi, puis avertir le système de cet arrêt en appuyant sur le « bouton opérateur ». Entre le déclenchement de l'alarme et l'appui sur le bouton, le système doit ouvrir l'eau froide au maximum afin, si possible, de préserver l'intégrité du procédé refroidi. Notons qu'il n'est pas possible de distinguer si l'arrêt du flux d'eau industrielle est dû à une coupure d'eau ou à une fuite, et qu'il est impossible de distinguer si l'arrêt du flux d'eau distillée est dû à une panne de la pompe ou à une fuite.

7.2 Spécification

Nous avons présenté dans le chapitre 1 le cycle en W du développement d'un système de contrôle-commande. La première phase consiste donc à spécifier, concevoir et implémenter un système pilotant un simulateur purement logiciel. En effet, typiquement, l'échangeur est piloté manuellement dans une entreprise, et un arrêt de ses fonctionnalités pendant les phases de test du système de contrôle serait pénalisant. De plus, des commandes erronées pourraient mettre en péril son intégrité. Enfin, il arrive fréquemment que le matériel de commande et les dispositifs électroniques nécessaires à son interfaçage avec le système de contrôle soient en développement parallèlement au développement du système de contrôle.

Dans le second V, lors de la spécification adaptée, nous verrons qu'il est nécessaire de prendre en compte les spécificités matérielles (relais électriques, nécessité d'alimenter un débitmètre avant de l'utiliser…).

Dans la spécification, nous considérons qu'il est possible de lire directement les débits en litre/heure, qu'il est possible de piloter directement une électrovanne, etc. La figure 7.2 représente le diagramme de contexte obtenu.

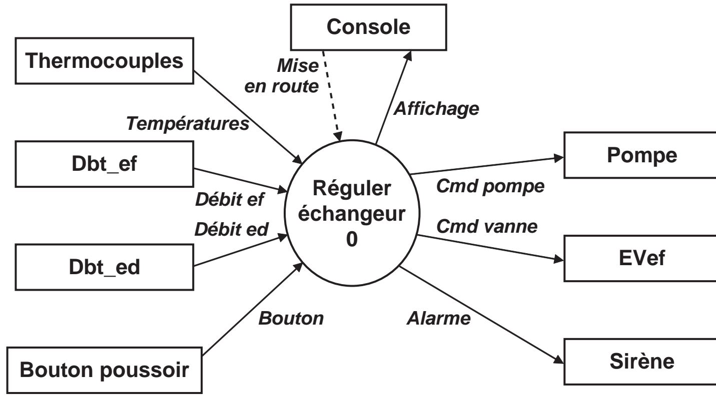


Figure 7.2 – Diagramme de contexte du système de contrôle de l'échangeur de chaleur.

Nous pouvons alors dériver le diagramme préliminaire (figure 7.3). Notons que les alarmes ne sont pas dissociées (*Acquérir T°*, *Réguler ef*, et *Vérifier débit ed* envoient le même événement *Alarme* en cas d'incohérence avérée). Notons que les processus 1, 2 et 3 sont des processus d'acquisition. L'événement *Trigger* déclenchant le processus 5 a une sémantique particulière : en effet, *Lire Bouton*, lorsqu'il est déclenché, doit scruter l'état du bouton jusqu'à ce qu'il soit en l'état appuyé, dans ce cas, il renvoie un événement *ACK* (pour *acknowledge*). Ce type d'utilisation du *Trigger* ressemble à un appel de sous-programme : il est déclenché, et lorsqu'il se termine, envoie un événement de retour.

La sémantique de l'événement *E/D* envoyé sur le processus 4 est une sémantique de type activation d'un processus fonctionnel synchronisé par flot de données : à partir de l'événement *Enable*, le processus traite chaque donnée *Commande débit* reçue. Le processus 8 est un processus cyclique d'affichage. Le rôle d'*Afficher alarme* est de modifier l'état de l'alarme, qui ne sera alors éteinte qu'à la terminaison du système. Enfin, l'événement *E/D* commandant le processus *Commander Pompe* a une sémantique de type *Allumer/Éteindre*.

Le rôle du processus de contrôle est donné sous la forme d'un diagramme état/transition sur la figure 7.4. Initialement les processus d'acquisition (1, 2, 3) sont déclenchés, le processus 4 se tient prêt à traiter les données, le processus 8 est lancé afin d'afficher les courbes de températures et débits, et la pompe est mise en route par le processus 6.

© Dunod – La photocopie non autorisée est un délit.

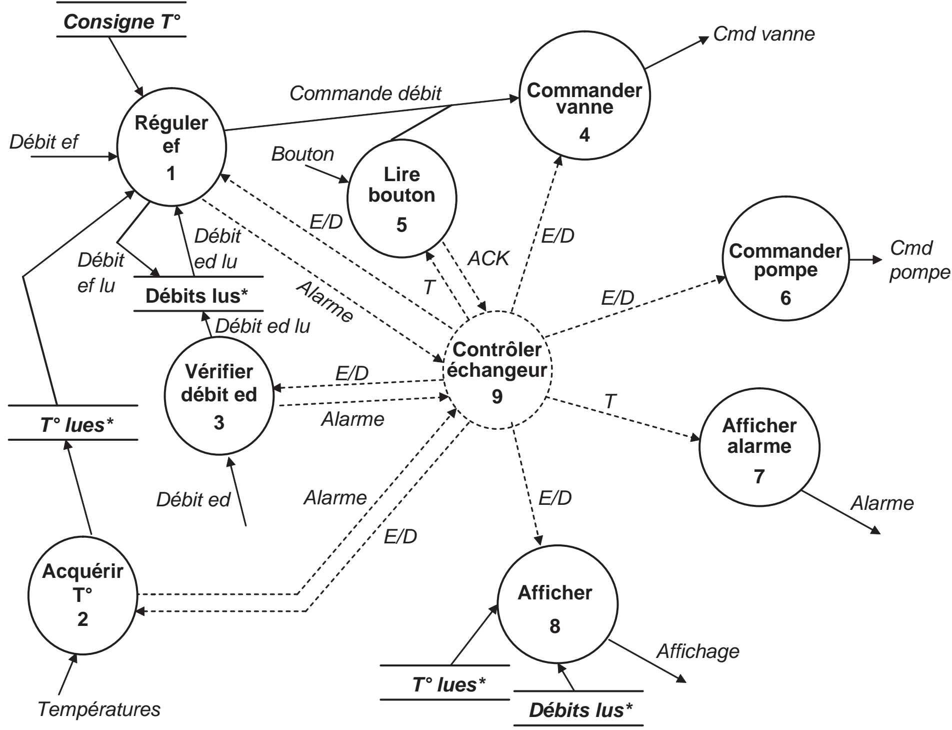


Figure 7.3 – Diagramme préliminaire
du système de contrôle de l'échangeur de chaleur.

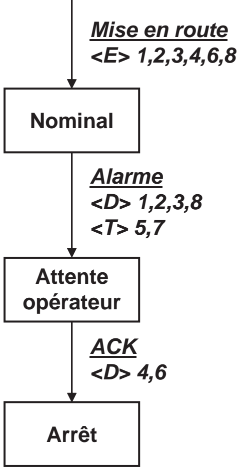


Figure 7.4 – Diagramme état/transition du processus de contrôle.

En cas d'alarme, les processus d'acquisition et d'affichage sont arrêtés, l'alarme est déclenchée, et on attend la prise en compte de l'alarme par l'opérateur (déclenchement du processus 5, dont la commande « ouvrir la vanne d'eau industrielle en grand » va supplanter les commandes envoyées habituellement par le processus 1). Lorsque l'opérateur a validé l'alarme (appui sur le bouton), le système s'arrête en coupant l'électrovanne et la pompe.

Avant d'aller plus en profondeur (décomposition des processus fonctionnels complexes en DFD), nous pouvons étudier le dictionnaire de données du système et la spécification textuelle des processus fonctionnels simples, qui deviendront de fait primitifs (tableau 7.1).

Tableau 7.1 – Dictionnaire de données associé aux diagrammes de contexte et préliminaire.

Données et événements externes	
Températures	$T°ef_1 + T°ef_2 + T°ed_1 + T°ed_2$
$T°ef_1$	Signal en entrée Rôle : Température d'entrée de l'eau industrielle en °C Type : Flottant sur 32 bits Domaine :]0..40 °C]
$T°ef_2$	Signal en entrée Rôle : Température de sortie d'eau industrielle en °C Type : Flottant sur 32 bits Domaine :]0..60 °C]
$T°ed_1$	Signal en entrée Rôle : Température d'entrée de cuve en °C Type : Flottant sur 32 bits Domaine :]0..70 °C]
$T°ed_2$	Signal en entrée Rôle : Température de sortie de cuve en °C Type : Flottant sur 32 bits Domaine :]0..85 °C]
Débits lus	Débit ef + Débit ed
Débit ef	Signal en entrée Rôle : Débit d'eau industriel relevé en entrée en litre/heure (l/h) Type : Flottant sur 32 bits Domaine : [0..200 l/h]
Débit ed	Signal en entrée Rôle : Débit d'eau distillée pris en entrée de cuve en l/h Type : Flottant sur 32 bits Domaine : [0..320 l/h]

© Dunod – La photocopie non autorisée est un délit.

Tableau 7.1 (suite) – Dictionnaire de données associé aux diagrammes de contexte et préliminaire.

Données et événements externes	
Bouton	Signal en entrée Rôle : État du bouton poussoir Type : booléen (vrai : appuyé et faux : relâché)
Mise en route	Événement en entrée Rôle : Représente la mise sous tension du système de contrôle
Affichage	Températures + Débits
Débits	Débit ef + Débit ed
Cmd pompe	Signal en sortie Rôle : Commande d'alimentation de la pompe d'eau distillée Type : booléen (vrai : allumée et faux : éteinte)
Cmd vanne	Signal en sortie Rôle : Commande de l'électrovanne d'entrée d'eau industrielle Type : flottant sur 32 bits Domaine : [0..200 l/h]
Alarme	Signal en sortie Rôle : Commande du système d'alarme Type : booléen (vrai : allumée et faux : éteinte)
Zones de stockage du diagramme préliminaire	
Débit ed lu	Débit ed
T° lues	Températures
Flots de données du diagramme préliminaire	
Commande débit	Débit ed

Le tableau 7.2 donne la spécification des processus primitifs de la figure 7.3.
Nous pouvons remarquer que, mis à part le processus *Réguler ef* qu'il est nécessaire de réellement arrêter (il existe une action *<D>Réguler ef* dans le diagramme état/transition du processus de contrôle), les processus déclenchés par *E/D* sont des processus de type « faire toujours » (un événement *Disable* ne leur est envoyé qu'à la terminaison du programme).
Les processus fonctionnels 2, 3, 4, 5, 6, 7 et 8 sont suffisamment simples pour être primitifs. Le diagramme de décomposition du processus *Réguler ef* est donné sur la figure 7.5. Le processus de contrôle 1.1 n'est qu'un relais du processus de contrôle 9.

Tableau 7.2 – Processus primitifs du diagramme préliminaire.

Acquérir T°	**E/ données** : Températures **E/ événements** : E/D **S/ données** : T°lues **S/ événements** : Alarme **Entraîne** : **Faire toujours** Acquérir Températures Placer Températures dans T°lues **Si** l'une des Températures est hors domaine **Alors** Émettre Alarme vers Processus de contrôle 9 **Finsi** **Fin faire**
Vérifier débit ed	**E/ données** : Débit ed **E/ événements** : E/D **S/ données** : Débit ed lu **S/ événements** : Alarme **Entraîne** : **Faire toujours** Acquérir Débit ed Placer Débit ed dans Débit ed lu **Si** Débit ed est hors domaine **Alors** Émettre Alarme vers Processus de contrôle 9 **Finsi** **Fin faire**
Réguler ef	**E/ données** : Consigne T° **E/ données** : Débit ed lu **E/ données** : T°lues **E/ données** : Débit ef **E/ événements** : E/D **S/ données** : Commande débit **S/ événements** : Alarme **Entraîne** : **Tant que** non(Disable) **faire** Ce processus doit s'arrêter en pratique sur l'événement Disable afin d'éviter un conflit sur le flot Commande débit Acquérir Débit ef **Si** Débit ef hors domaine ou incohérent avec la commande **Alors** Émettre Alarme vers Processus de contrôle 9 **Finsi** Placer Débit ef dans Débits lus Élaborer Commande débit=f(T°lues, Débit ed lu, Débit ef, Consigne T°) Émettre Commande débit vers Processus 4 **Fin faire** **Si** Disable **Alors** Émettre Commande débit au maximum **Finsi**

© Dunod – La photocopie non autorisée est un délit.

Tableau 7.2 (suite) – Processus primitifs du diagramme préliminaire.

<table>
<tr><td>Lire bouton</td><td>E/ données : Bouton
E/ événements : T
S/ événements : ACK
Entraîne :
    Faire
        Acquérir Bouton
    Jusqu'à Bouton = vrai
    Émettre ACK vers Processus de contrôle 9</td></tr>
<tr><td>Afficher alarme</td><td>E/ événements : T
S/ données : Alarme
Entraîne :
    Déclencher l'alarme</td></tr>
<tr><td>Commander pompe</td><td>E/ événements : E/D
S/ données : Cmd pompe
Entraîne :
    Faire toujours
        Attendre E/D
        Si E
            Alors Allumer pompe
        Sinon Éteindre pompe
        Finsi
    Fin faire</td></tr>
<tr><td>Commander vanne</td><td>E/ données : Commande débit
E/ événements : E/D
S/ données : Cmd vanne
Nécessite : Commande débit dans son domaine
Entraîne :
    Faire toujours
        Attendre Commande débit
        Appliquer Commande débit à Cmd vanne
    Fin faire</td></tr>
<tr><td>Afficher</td><td>E/ données : T°lues
E/ données : Débits lus
E/ événements : E/D
S/ données : Affichage
Entraîne :
    Faire toujours
        Afficher T°lues et Débits lus
    Fin faire</td></tr>
</table>

Le tableau 7.3 représente la spécification des nouveaux processus créés (la spécification du processus 1 – *Réguler ef* peut alors être retirée de la spécification des processus primitifs). Notons que la prise en compte de l'événement *Disable* n'est placée que dans le processus fonctionnel *Calculer consigne*, ceci afin d'éviter un conflit d'émission de commande de débit vers *Commander vanne*.

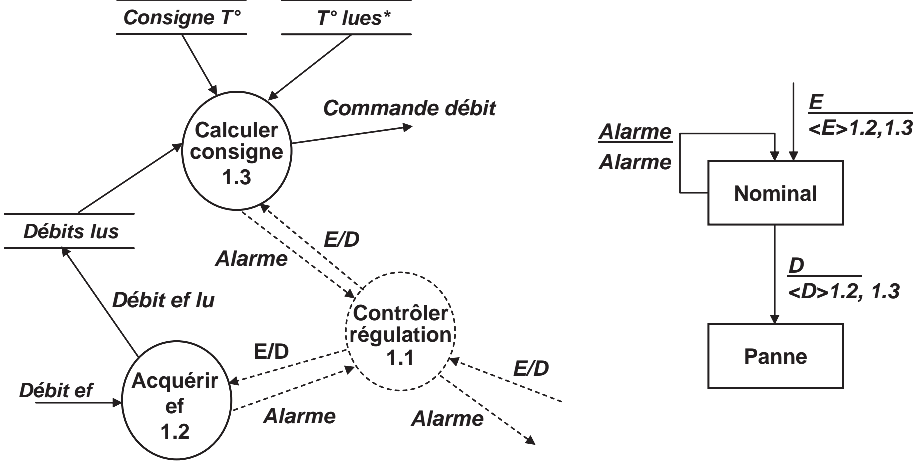


Figure 7.5 – DFD de *Réguler ef* et diagramme état/transition du contrôle.

Tableau 7.3 – Spécification des processus primitifs du DFD de « Réguler ef ».

Calculer consigne	**E/ données** : Consigne T°
	E/ données : Débits lus
	E/ données : T°lues
	E/ événements : E/D
	S/ données : Commande débit
	S/ événements : Alarme
	Entraîne :
	    Tant que non(Disable) **faire**
	Ce processus doit s'arrêter en pratique sur l'événement Disable afin d'éviter un conflit sur le flot Commande débit
	        Si Débits lus incohérents avec dernière Commande débit
	            Alors Émettre Alarme
	        Finsi
	        Élaborer Commande débit = f(T°lues, Débit ed lu, Débit ef, Consigne T°)
	        Émettre Commande débit vers Processus 4
	    Fin faire
	    Si Disable
	        Alors Émettre Commande débit au maximum
	    Finsi

© Dunod – La photocopie non autorisée est un délit.

Tableau 7.3 (suite) – Spécification des processus primitifs du DFD de « Réguler ef ».

Acquérir ef	**E/ données** : Débit ef
	E/ événements : E/D
	S/ données : Débit ef lu
	S/ événements : Alarme
	Entraîne :
	    Faire toujours
	        Acquérir Débit ef
	        Si Débit ef hors domaine ou incohérent avec la commande
	            Alors Émettre Alarme vers Processus de contrôle 9
	        Finsi
	        Placer Débit ef dans Débits lus
	    Fait

7.3 Conception

Il est maintenant possible de passer à la conception DARTS du système. Il est toujours intéressant de regrouper les processus fonctionnels en un minimum de tâches : la validation temporelle du système est simplifiée, la cohérence fonctionnelle est plus simple à obtenir (dans le cas étudié, il faut s'assurer que *Lire Bouton* envoie bien une donnée à *Commander vanne* après que *Calculer consigne* se soit arrêté). De plus, avec moins de tâches, le fonctionnement est plus efficace (diminution du nombre de préemptions, des communications, de la taille de la file des processus à ordonnancer, etc.).

Initialement, chaque processus est vu comme une tâche, les flots de données et d'événements sont implémentés par une boîte aux lettres, une synchronisation, ou rien (cas de l'événement *E/D* lorsque seul l'événement *E* est envoyé initialement par exemple), enfin, les zones de stockage partagées par plusieurs processus sont généralement implémentées par des modules de données. La figure 7.5 représente un diagramme DARTS pour le système de contrôle de l'échangeur.

Notons que seuls les événements réellement utilisés par le processus de contrôle sont reportés dans la conception. Ainsi, par exemple, l'événement *E/D* envoyé sur le processus fonctionnel *Commander_Vanne* ne sera pas implémenté : la tâche *Commander_Vanne*, implémentant les fonctionnalités du processus fonctionnel du même nom, est prête au lancement du système, et ne sera arrêtée qu'à l'extinction du système.

La tâche *Réguler débit* intègre les processus fonctionnels 1, 3, 5, 9 : les processus 1 et 3 ont une cohérence fonctionnelle forte, de plus, la régulation est d'autant plus fine qu'elle se base sur des valeurs de débit juste acquises (le débit peut en effet varier relativement vite, contrairement aux températures). Le fonctionnement du processus 5 correspond à un appel de sous-programme par le processus 9 : ils sont donc regroupés en une tâche. Cependant, étant donné que la tâche obtenue serait assez simple, et fonctionnerait au plus à la même vitesse que la régulation de débit, les 4 processus fonctionnels sont finalement intégrés dans une même tâche.

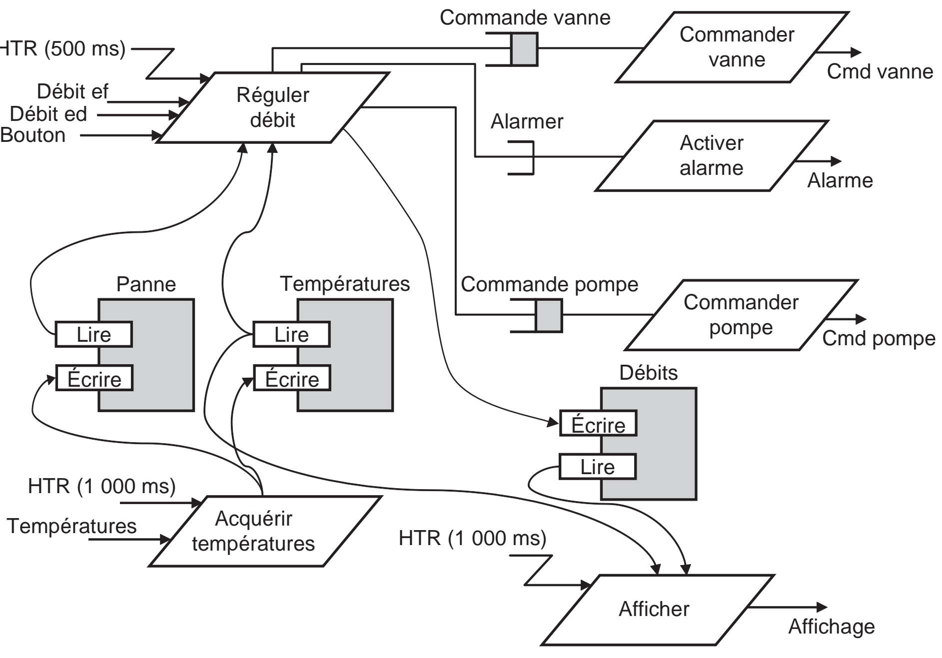


Figure 7.6 – Diagramme DARTS de l'échangeur.

Il est intéressant d'isoler les tâches de commande des tâches de décision : en effet, il est fréquent que la durée nécessaire à une entrée/sortie sur une carte d'acquisition soit très grande comparée à la durée d'un calcul, même complexe. Par conséquent, dans la mesure du possible, il est bon d'éviter de surcharger une tâche de calcul et/ou de décision en lui faisant effectuer des entrées/sorties (sauf si celles-ci sont de toute façon nécessaires à l'élaboration d'un calcul, comme c'est le cas ici pour les débits). Notons cependant que nous sommes face à un système très particulier : l'alarme n'est déclenchée, et la pompe éteinte, qu'en cas de panne, ce qui est exceptionnel. Dans ce cas précis, étant donné que cela précède l'arrêt du système, il serait plus intéressant de regrouper les tâches *Alarme* et *Commander pompe* avec *Réguler débit*. Par conséquent, nous donnons un schéma DARTS simplifié sur la figure 7.7.

Notons qu'il est possible de passer d'une communication synchrone (flot de données ou événement) SA-RT à une communication asynchrone DARTS, ceci afin de garantir pour chaque tâche DARTS un seul événement déclencheur (conformité au profil Ravenscar). Ainsi, l'événement *Alarme* envoyé par *Acquérir températures* au processus de contrôle implémenté par *Réguler débit* se retrouve implémenté par le module de données *Panne* (communication asynchrone). Cela permet de conserver un seul mode d'activation pour réguler débit, ce qui est obligatoire dans le cas où une validation temporelle serait nécessaire.

Afin de pouvoir tester le système sans mettre en péril le procédé, et de tester sa réaction face aux pannes, il convient de créer un simulateur du procédé, conformément au premier V du cycle de développement en W. Celui-ci est intégré au diagramme de la figure 7.8.

© Dunod – La photocopie non autorisée est un délit.

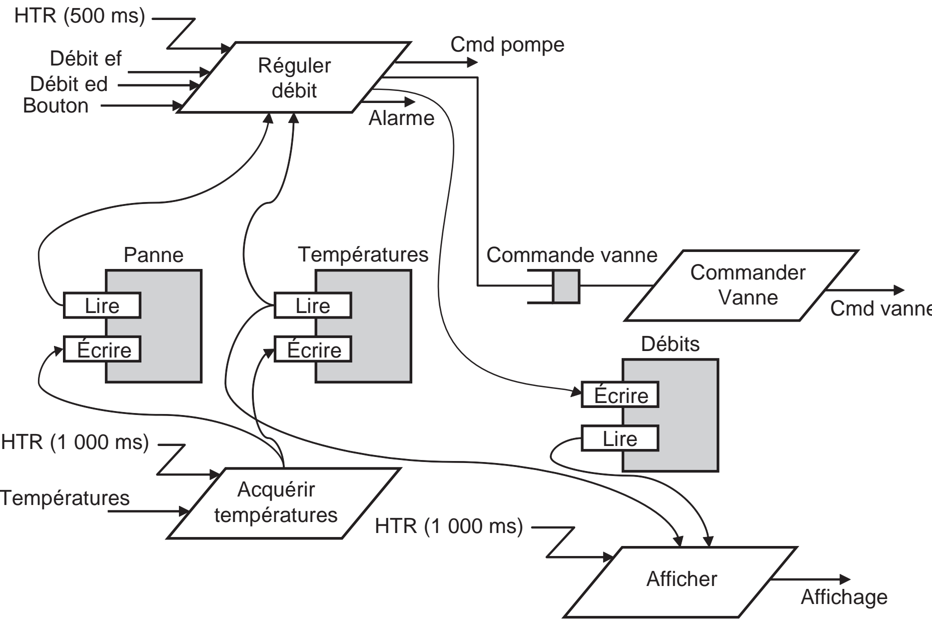


Figure 7.7 – Diagramme DARTS simplifié de l'échangeur.

Il reste alors à identifier les éléments DARTS aux éléments déjà définis dans SA-RT et à définir les nouveaux éléments. Afin de ne pas noyer les informations importantes, les éléments DARTS portant le même nom que les éléments SA-RT correspondants ne sont pas redéfinis dans le tableau 7.4.

HTR (500 ms)
Débit ef
Débit ed
Bouton
Réguler débit
Alarme
Cmd pompe
Panne
Lire
Écrire
Températures
Lire
Écrire
HTR (1 000 ms)
Températures
Acquérir températures
Commande vanne
Commander vanne
Débits
Écrire
Lire
HTR (1 000 ms)
Afficher
Affichage
In
Écrire
Lire
HTR (500 ms)
Simulateur
Simulateur de pannes
Affichage simulateur
out
Écrire
Lire

Figure 7.8 – Diagramme DARTS de l'échangeur intégrant un simulateur.

© Dunod – La photocopie non autorisée est un délit.

Tableau 7.4 – Correspondance entre éléments DARTS et SA-RT.

Panne	Module de donnée Rôle : état de panne du système Type : booléen (vrai : panne et faux : nominal)
Commande Vanne	Boîte aux lettres (si possible avec écrasement) Rôle : état désiré de la pompe d'eau industrielle Type : booléen (vrai : ouverte et faux : fermée)
Afficher	Tâche Correspond à : Processus fonctionnel 8
Acquérir températures	Tâche Correspond à : Processus fonctionnel 2
Réguler débits	Tâche Correspond à : Processus fonctionnels 1, 3, 5 et processus de contrôle 9
Simulateur	Tâche de simulation du procédé
In	Module de données Rôle : simule les capteurs Type : enregistrement (Débit ef, Débit ed, Températures)
Out	Module de données Rôle : simule les actionneurs Type : enregistrement (Cmd vanne, Cmd pompe)

7.4 Implémentation sur simulateur

La première implémentation d'un système de contrôle-commande se base généralement sur un simulateur logiciel du procédé.

7.4.1 Le simulateur

Le simulateur peut s'exécuter en tant que tâche parmi les autres tâches du système, ou bien tout simplement dans un autre processus, écrit dans un langage de programmation facilitant la simulation (LabVIEW, Matlab/Simulink, etc.). L'une des techniques les plus simples permettant de faire communiquer deux processus différents pouvant s'exécuter sur des ordinateurs différents consiste à utiliser le protocole TCP/IP. Cela reste vrai lorsque les processus s'exécutent sur un même ordinateur : le protocole TCP/IP est alors utilisé en boucle locale. Le simulateur utilisé dans ce chapitre sera donc un programme lançant un serveur TCP, et exécutant en parallèle un modèle numérique (plus ou moins réaliste) du procédé.

Le protocole applicatif défini entre le client (le système de contrôle) et le serveur (le simulateur) est en général relativement simple : après l'établissement d'une

connexion, le simulateur est en attente de commandes, envoyées par le système de contrôle sous forme de texte. En effet, lorsqu'un protocole de communication peut impliquer des systèmes d'exploitation/langages de programmation hétérogènes, il est conseillé d'utiliser un protocole purement textuel afin de ne pas avoir de problèmes liés à la représentation binaire des données (ordre des octets *big-endian* ou *little-endian*, taille des représentations). Ainsi, pour le cas étudié, les commandes possibles sont :

– *pompe 1* : le simulateur met la pompe en marche ;
– *pompe 0* : le simulateur éteint la pompe ;
– *alarme 1* : le simulateur allume l'alarme ;
– *alarme 0* : le simulateur éteint l'alarme ;
– *vanne val* : *val* est un flottant donné sous forme de chaîne de caractères. Le simulateur ouvre la vanne au niveau de *val* ;
– *ef* : le simulateur renvoie la valeur de débit d'eau industrielle ;
– *ed* : le simulateur renvoie la valeur de débit d'eau distillée ;
– *bouton* : le simulateur renvoie 1 si le bouton est appuyé, 0 sinon ;
– *temp* : le simulateur renvoie les quatre températures dans l'ordre *T°ed1*, *T°ed2*, *T°ef1*, *T°ef2*.

7.4.2 Implémentation en Ada du système sur simulateur

La figure 7.9 montre une architecture logicielle typique pour la partie multitâche d'un programme de contrôle-commande en Ada, facilitant le passage du simulateur à la commande réelle. Le paquetage générique *Communications* (fourni en entier en annexe D) regroupe différents éléments DARTS très utilisés (voir § 6.3). Ceux-ci sont instanciés par le paquetage *Contrôle* regroupant les tâches définies à la conception. Ce qui est intéressant est que deux paquetages ont une spécification commune : en effet, *Simulateur* et *Procédé* partagent les mêmes signatures de primitives, qui sont des primitives d'accès de haut niveau au matériel (dans le cas présent, Lire_Températures en °C, Commander_Vanne en litres/heure, etc.). Ainsi, le passage de la simulation au contrôle réel sera simplifié, puisque seules les clauses *with* et *use* devront changer et passer du simulateur au procédé réel.

■ Le module de communication

Commençons par observer le paquetage *Communications.* Celui-ci regroupe différents éléments de communication.

```
-- Paquetage Communications
-- Définit des outils génériques de communication DARTS
-- Modules de données, synchronisations, boîtes aux lettres
-- avec ou sans écrasement, de taille unitaire ou bornée

with System; use System;
with File_Bornee;
generic
   type element is private;
package Communications is
```

© Dunod – La photocopie non autorisée est un délit.

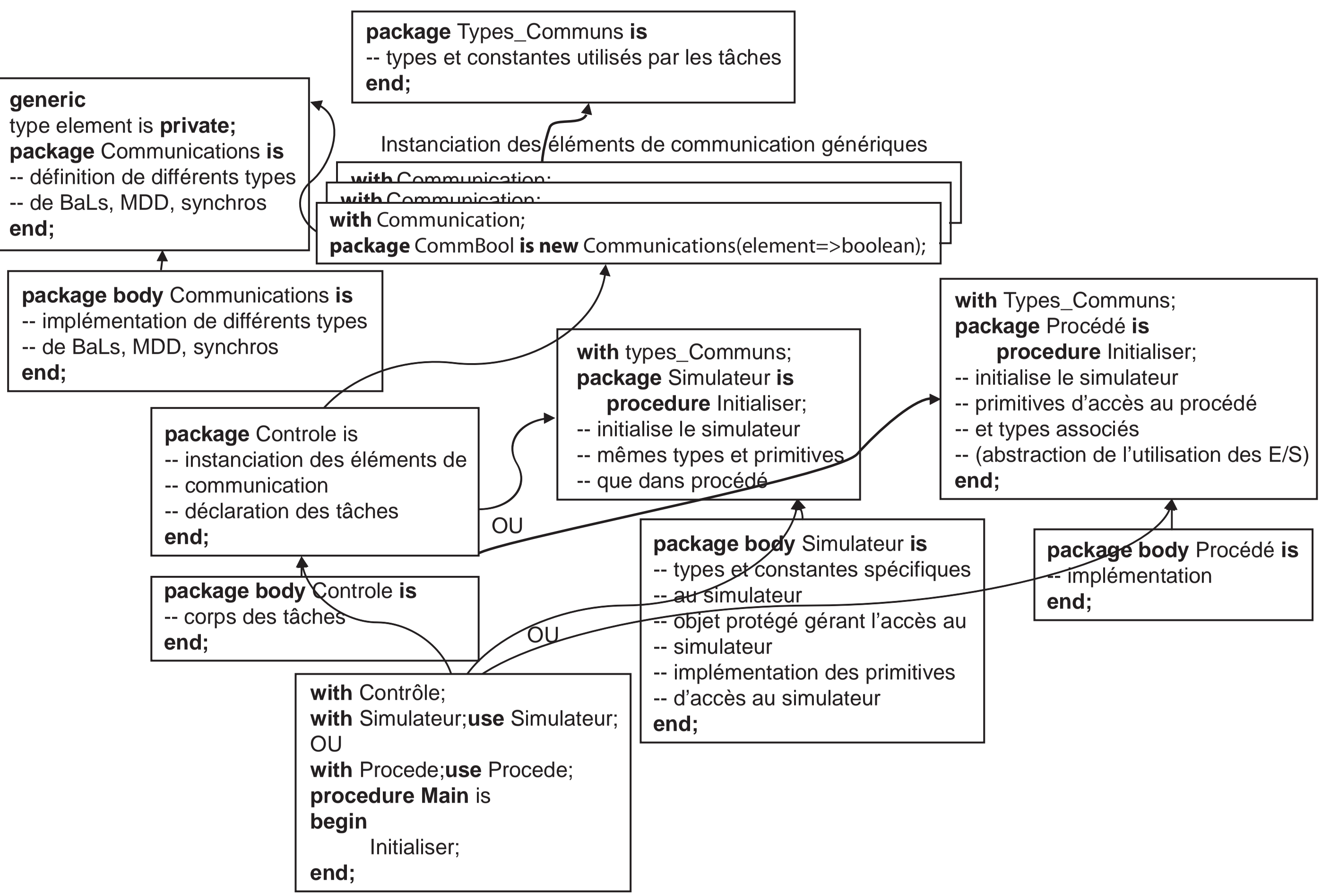


Figure 7.9 – Architecture logicielle typique d'un programme de contrôle-commande simple en Ada.

```
   package File_Bornee_N_Elements is new
File_Bornee(element=>element);
   use File_Bornee_N_Elements;
   type Pt_Element is access all element;
------------------------
-- Modules de données
------------------------
   protected type MDDi(initial: pt_element := null; priorité:
natural := priority'last) is
   -- module de données avec valeur initiale
   -- le passage par pointeur est dû au fait qu'un discriminant
   -- doit être de tye discret ou pointeur
      pragma Priority(priorité);
      procedure Ecrire(e: element);
      -- modifie le contenu
      function Lire return element;
      -- retourne le contenu
   private
      val: element := initial.all;
   end;

   protected type MDD(priorité: natural := priority'last) is
   -- module de données sans valeur initiale
      pragma Priority(priorité);
      procedure Ecrire(e: element);
      -- modifie le contenu
      function Lire return element;
      -- retourne le contenu
   private
      val: element;
   end;
```

Les modules de données (MDD) sont avec ou sans valeur initiale. Notons qu'un discriminant ne peut être qu'une valeur de type discret ou bien un pointeur. Un inconvénient visible ici est le fait d'être obligé de passer par pointeur une valeur initiale au module de donnée de type `MDDi`.

```
------------------------
-- Synchronisation
------------------------
   protected type Synchro_C(priorité: natural := priority'last) is
   -- Synchronisation à compte (les déclenchements successifs
   -- non pris en compte sont accumulés)
      pragma Priority(priorité);
      procedure Signal;
      entry Wait;
   private
      Nombre_Signalés: natural := 0;
   end;

   protected type Synchro_B(priorité: natural := priority'last) is
   -- Synchronisation binaire (les déclenchements successifs
   -- non pris en compte ne sont pas accumulés)
      pragma Priority(priorité);
      procedure Signal;
      entry Wait;
   private
      Signalé: boolean := false;
   end;
```

© Dunod – La photocopie non autorisée est un délit.

Les synchronisations peuvent être de type binaire ou à compte.

```
------------------------
-- Boîtes aux lettres
------------------------
  protected type BaL_1_Ecrasement(priorité: natural := priority'last)
is
   -- Boîte aux lettres de taille 1 avec écrasement
      pragma Priority(priorité);
      procedure Envoyer(e: element);
      entry Recevoir(e: out element);
   private
      Pleine: boolean := false;
      val: element;
   end;

   protected type BaL_1(priorité: natural := priority'last) is
   -- Boîte aux lettres de taille 1 sans écrasement
      pragma Priority(priorité);
      entry Envoyer(e: element);
      -- l'envoi peut être bloquant
      entry Recevoir(e: out element);
   private
      Pleine: boolean := false;
      val: element;
   end;

  protected type BaL_n_Ecrasement(n:positive:=1; priorité: natural :=
priority'last) is
   -- Boîte aux lettres de taille n avec écrasement
      pragma Priority(priorité);
      procedure Envoyer(e: element);
      entry Recevoir(e: out element);
   private
      F: File_Bornee_N_Elements.File_Bornee(n);
   end;

   protected type BaL_n(n:positive:=1; priorité: natural :=
priority'last) is
   -- Boîte aux lettres de taille n sans écrasement
      pragma Priority(priorité);
      entry Envoyer(e: element);
      -- l'envoi peut être bloquant
      entry Recevoir(e: out element);
   private
      F: File_Bornee_N_Elements.File_Bornee(n);
   end;
end; -- Fin de la spécification de paquetage
```

Les boîtes aux lettres peuvent être de taille 1 ou *n*, et être à écrasement ou sans écrasement. Les boîtes aux lettres de taille *n* avec écrasement sont assez proche de ce qui est qualifié de RT FIFO (file temps réel) permettant une communication entre tâche émettrice à contraintes de temps et tâche réceptrice sans contrainte de temps : la tâche à contraintes de temps ne se bloque pas si la file est pleine, mais dans ce cas le plus ancien message est écrasé. Le paquetage générique de gestion de liste circulaire est donné en annexe D : il s'agit tout simplement d'un tableau, dans lequel un élément enfilé écrase le plus ancien élément si la file est pleine.

Rappelons enfin que la priorité associée à un objet protégé est la priorité plafond de celui-ci, et qu'une tâche hérite de la priorité plafond d'un objet (protocole à priorité plafond immédiat, *pragma Locking_Policy(Ceiling_Locking)*) pendant son utilisation de celui-ci. La priorité plafond d'un objet protégé doit donc être la priorité maximale d'une tâche pouvant accéder à cet objet. Dans le cas où une tâche de priorité supérieure à l'objet protégé y accéderait, l'exception *Program_Error* serait levée.
Le corps de ce paquetage est donné en annexe D.

■ Le module de contrôle

Le paquetage *Types_Communs* est très court, puisque seuls deux types spécifiques sont définis pour le système :

```
-- Paquetage Types_Communs
package Types_Communs is
   type Temperatures is record
      Ted1, Ted2, Tef1, Tef2 : float;
   end record;
   type Debits is record
      ed, ef: float;
   end record;
end;
```

Le paquetage *Contrôle* qui s'appelle dans le cas présent *Contrôler_Echangeur* découle directement du diagramme DARTS :

```
-- Paquetage Controler_Echangeur
with Communications;
with System; use System;
with Types_Communs; use Types_Communs;
package Controler_Echangeur is
   ------------------------------
   -- Elements de communication
   ------------------------------
   package Com_Températures is new
Communications(Element=>Temperatures);
   -- Instanciation du paquetage générique de communication sur le
   -- type Temperatures
   use Com_Températures;
   Mdd_Temperatures: Com_Températures.MDD;
   -- Instanciation du MDD Temperatures

   package Com_Debits is new Communications(Element=>Debits);
   -- Instanciation du paquetage générique de communication sur le
   -- type Debits
   use Com_Debits;
   Mdd_Debits: Com_Debits.MDD;
   -- Instanciation du MDD Debits

   package Com_Boolean is new Communications(Element=>Boolean);
   -- Instanciation du paquetage générique de communication sur le
   -- type booléen
   use Com_Boolean;
   Faux: aliased Boolean:=False;
   -- Remarquer la technique employée pour passer un paramètre par
   -- adresse
```

© Dunod – La photocopie non autorisée est un délit.

```
   Mdd_Panne: Com_Boolean.Mddi(Faux'access, Priority'Last);
   -- Instanciation du MDD Panne

   package Com_Float is new Communications(Element => Float);
   -- Instanciation du paquetage générique de communication sur le
   -- type flottant
   use Com_Float;
   Bal_Commande_Vanne: Com_Float.Bal_1_Ecrasement;
   -- Instanciation de la BaL Commande Vanne, de taille 1 à écrasement
   -- (seule la dernière consigne est prise en compte)

   --------------------------------
   -- Taches
   --------------------------------
   -- Dans la partie simulation, le choix des priorités est
   -- généralement arbitraire
   -- puisqu'on emploie fréquemment une station de développement
   -- tournant sous un
   -- système d'exploitation non temps réel (=>indéterminisme
   -- temporel)
   task Reguler_Debit is
      pragma Priority(Default_Priority);
   end;
   task Commander_Pompe is
      pragma Priority(Default_Priority+1);
   end;
   task Acquerir_Temperatures is
      pragma Priority(Default_Priority-1);
   end;
   task Afficher is
      pragma Priority(Default_Priority-2);
   end;
end;
```

L'implémentation suit le schéma défini dans le chapitre 6. Trois tâches sont périodiques, la quatrième est déclenchée par boîte aux lettres.

```
with Ada.Real_Time; use Ada.Real_Time;
with Affichage;use Affichage;
with Simulateur; use Simulateur;
package body Controler_Echangeur is
   T0: constant Time := Clock;
   -- Base des temps

   function Elaborer_Consigne(D: Debits ; T: Temperatures) return
float is
      csg: float;
   begin
      if (T.Ted2<=45.0) then
         if D.Ef-1.0>=0.0 then
            return D.Ef-1.0;
         else
            return D.Ef;
         end if;
      end if;
      Csg:=(T.Ted2-45.0)*3.0;
      if Csg>Max_Ouverture_Vanne then
         return Max_Ouverture_Vanne;
      else
```

```
      return Csg;
   end if;
end;

function Verification(T: Temperatures) return Boolean is
   -- Renvoie: vrai ssi les températures sont dans leur domaine
begin
   if T.Tef1<0.0 or T.Tef1>40.0 then return false;
   elsif T.Tef2<0.0 or T.Tef2>60.0 then return false;
   elsif T.Ted1<0.0 or T.Ted1>70.0 then return false;
   elsif T.Ted2<0.0 or T.Ted2>85.0 then return false;
   end if;
   return true;
end;

task body Acquerir_Temperatures is
   Periode: constant Time_Span:= To_Time_Span(1.0);
   -- Tâche périodique de période 1 seconde
   Prochaine: Time := T0+Periode;
   -- Prochaine date d'activation
   T: Temperatures;
begin
   loop
      T:=Lire_Temperatures;
      -- Acquisition des températures
      if not(Verification(T)) then
         Mdd_Panne.Ecrire(True);
      end if;
      Mdd_Temperatures.Ecrire(T);
      delay until Prochaine;
      Prochaine := Prochaine + Periode;
   end loop;
end Acquerir_Temperatures;

task body Afficher is
   Periode: constant Time_Span:= To_Time_Span(1.0);
   -- Tâche périodique de période 1 seconde
   Prochaine: Time := T0+Periode;
   -- Prochaine date d'activation
   T: Temperatures;
   D: Debits;
begin
   loop
      T:=Mdd_Temperatures.Lire;
      D:=Mdd_Debits.Lire;
      -- Lecture des températures et débits
      Afficher_Courbes(T,D);
      -- Affichage
      delay until Prochaine;
      Prochaine := Prochaine + Periode;
   end loop;
end Afficher;

task body Piloter_Vanne is
   Cmd:Float;
begin
   loop
      BaL_Commande_Vanne.Recevoir(Cmd);
      -- Tâche déclenchée à chaque message
```

© Dunod – La photocopie non autorisée est un délit.

```
         Commander_Vanne(Cmd);
      end loop;
   end Piloter_Vanne;

   task body Reguler_Debit is
      Periode: constant Time_Span:= To_Time_Span(0.5);
      -- Tâche périodique de période 500 millisecondes
      Prochaine: Time := T0+Periode;
      -- Prochaine date d'activation
      D: Debits;
      Consigne_Ed: Float;
      type T_Etat is (Normal, Alarme);
      Etat: T_Etat:=Normal;
   begin
      Commander_Pompe(True);
      -- Allumage de la pompe d'eau distillée
      loop
         case Etat is
            when Normal=>
               if Mdd_Panne.Lire then
                  Etat:=Alarme;
                  Commander_Alarme(True);
                  --Mise en route de l'alarme
                  Bal_Commande_Vanne.Envoyer(Max_Ouverture_Vanne);
                 --Ouverture de la vanne d'eau industrielle au maximum
               else
                  D.Ed:=Lire_Debit_Ed;
                  D.Ef:=Lire_Debit_Ef;
                  -- Acquisition des débits
                  Mdd_Debits.Ecrire(D);
                  -- Stockage dans le MDD

Consigne_Ed:=Elaborer_Consigne(D,Mdd_Temperatures.Lire);
                  -- Elaboration de la consigne
                  Bal_Commande_Vanne.Envoyer(Consigne_Ed);
                  -- Envoie à la tâche de commande
               end if;
            when Alarme =>
               if Lire_Bouton then
                  Commander_Alarme(False);
                  --Extinction de l'alarme
                  Bal_Commande_Vanne.Envoyer(0.0);
                  --Fermeture de la vanne d'eau industrielle
                  Commander_Pompe(False);
                  -- Extinction de la pompe d'eau distillée
                  exit; --Terminaison de la tâche
               end if;
         end case;
         delay until Prochaine;
         Prochaine := Prochaine + Periode;
      end loop;
   end Reguler_Debit;
end Controler_Echangeur;
```

Le module de communication avec le simulateur

Voyons maintenant comment le programme Ada communique par TCP avec le simulateur. Pour cet exemple, le paquetage *Gnat.Sockets* est utilisé : c'est un paquetage spécifique fourni avec le compilateur gratuit GNAT. Il implémente les communications TCP/IP et UDP/IP et est d'une utilisation très simple. Cependant, il souffre d'un petit inconvénient dont il faut être averti lorsque l'on doit permettre une communication entre deux programmes écrits dans un langage différent : *Gnat.Sockets* utilise un protocole spécifique, ainsi, lors de l'envoi d'une chaîne de caractère (procédure *String'Output(s)*), la chaîne de caractères *s* est transmise sous la forme : un entier valant 1 donné sur 4 octets au format binaire, suivi d'un entier sur 4 octets au format binaire valant la longueur de la chaîne, et enfin les caractères de *s*. La fonction *String'Input* attend le même format. Ainsi, la bibliothèque *Ada.Sockets* ajoute une fine couche protocolaire au-dessus de la pile TCP dont il faut tenir compte dans le simulateur.

Il est important de constater que différentes tâches peuvent en parallèle tenter d'accéder au simulateur. Il est donc nécessaire de protéger les accès au simulateur afin de garantir leur exclusion mutuelle. C'est le rôle de l'objet protégé *Simulateur*, qui est le seul à accéder réellement au simulateur. Cependant, afin de permettre au paquetage *Simulateur* d'offrir les mêmes signatures de primitives que le paquetage *Procédé*, les appels à l'objet protégé sont encapsulés dans des fonctions et procédures classiques (ayant la même signature que celles qui seront dans le paquetage *Procédé*). Enfin, il faut noter que grâce à l'utilisation de l'objet protégé *Simulateur*, nous garantissons que la connexion TCP est initialisée avant tout autre accès : la variable *Initialisé* du simulateur, mise à vrai lors de l'initialisation de celui-ci, conditionne l'accès à toute autre primitive du simulateur.

```
with Types_Communs;use Types_Communs;
with Gnat.Sockets;use Gnat.Sockets;
package Simulateur is
-- Un simulateur est défini par un serveur TCP écoutant un port
-- (par défaut 34532).
-- Au début de l'application, une connexion TCP est ouverte avec le
-- simulateur
-- Le protocole employé est de type commande du système/réponse du
-- procédé
-- Par exemple, pour lire le débit ef, la fonction Lire envoie "ef",
-- le simulateur
-- répond alors en envoyant un flottant sous forme de chaîne de
-- caractères
-- Pour commander la pompe, le simulateur envoie "pompe 0" ou "pompe
-- 1"
-- en fonction de la valeur de la commande

-- !!! ATTENTION !!! Le format de communication TCP de Gnat.Sockets
-- est très spécifique :
-- Lors de l'envoi d'une chaîne de caractères, celle-ci est précédée
-- de 8 octets correspondant
-- chacun à un entier en format binaire : le premier vaut 1, le second
-- vaut la longueur de la chaîne
-- Les chaînes de caractères lues doivent être au même format
   Erreur_Procede : exception;
   -- Exception levée en cas d'erreur de connexion avec le simulateur
```

© Dunod – La photocopie non autorisée est un délit.

```
   Max_Ouverture_Vanne: constant Float:=200.0;
   -- Consigne maximale en litre/heure de l'ouverture de vanne
   procedure Initialiser;
   -- Connexion au simulateur
   -- Exception: erreur de connexion => Erreur_Procede
   function Lire_Debit_Ef return Float;
   -- Renvoie le débit d'eau industrielle en l/h
   -- Exception: erreur de connexion => Erreur_Procede
   function Lire_Debit_Ed return Float;
   -- Renvoie le débit d'eau distillée en l/h
   -- Exception: erreur de connexion => Erreur_Procede
   function Lire_Temperatures return Temperatures;
   -- Renvoie les températures en °C
   -- Exception: erreur de connexion => Erreur_Procede
   function Lire_Bouton return Boolean;
   -- Renvoie l'état du bouton poussoir
   -- Retourne: vrai si appuyé, faux sinon
   -- Exception: erreur de connexion => Erreur_Procede
   procedure Commander_Pompe (Com : Boolean );
   -- Entraine: Com = vrai => allume la pompe
   --           Com = faux => éteint la pompe
   -- Exception: erreur de connexion => Erreur_Procede
   procedure Commander_Vanne (Com : Float );
   -- Commande l'électrovanne en l/h
   -- Exception: erreur de connexion => Erreur_Procede
   procedure Commander_Alarme(On: Boolean);
   -- Allume ou éteint l'alarme
   -- Exception: erreur de connexion => Erreur_Procede
private
   protected Simulateur is
      -- Objet protégé garantissant l'exclusion mutuelle des accès au
      -- simulateur
      procedure Initialiser;
      -- Connexion au simulateur
      entry Lire_Debit_Ef (Val :    out Float );
      -- Interroge le simulateur sur la valeur de Debit Ef
      -- Ne peut avoir lieu qu'après initialisation
      entry Lire_Debit_Ed (Val :    out Float );
      -- Interroge le simulateur sur la valeur de Debit Ed
      -- Ne peut avoir lieu qu'après initialisation
      entry Lire_Temperatures (T :    out Temperatures );
      -- Interroge le simulateur sur la valeur des températures
      -- Ne peut avoir lieu qu'après initialisation
      entry Lire_Bouton(On: out Boolean);
      -- Interroge le simulateur sur l'état du bouton
      -- Entraine: On=vrai si le bouton est appuyé, faux sinon
      -- Ne peut avoir lieu qu'après initialisation
      entry Commander_Pompe (Val : Boolean );
      -- Allume ou éteint la pompe
      -- Ne peut avoir lieu qu'après initialisation
      entry Commander_Vanne (Val : Float );
      -- Commande la vanne
      -- Ne peut avoir lieu qu'après initialisation
      entry Commander_Alarme (Val : Boolean );
      -- Allume ou éteint l'alarme
      -- Ne peut avoir lieu qu'après initialisation
   private
      Initialisé: Boolean:=False;
      -- Mis à vrai par l'initialisation
```

```
      Socket: Socket_Type;
      -- Identifiant de connexion au simulateur
      Channel: Stream_Access;
      -- Flux de caractères échangés sur la connexion
   end;
end;
```

Le corps de ce paquetage est donné ci-après :

```
with Gnat.Sockets;use Gnat.Sockets;
with Types_Communs;use Types_Communs;
package body Simulateur is

   Adresse_Simulateur : constant String := "localhost";
   -- Adresse IP du simulateur
   Port_Simulateur : constant Integer := 34532;
   -- Port sur lequel le simulateur lance un serveur TCP

   -- Toutes les primitives appelées par les tâches se contentent
   -- d'appeler les primitives de l'objet protégé simulateur.
   -- Le but est de permettre aux tâches du système d'appeler les mêmes
   -- primitives que pour un contrôle réel sur le procédé.
   procedure Initialiser is
   begin
      Simulateur.Initialiser;
   exception when Socket_Error => raise Erreur_Procede;
   end Initialiser;

   procedure Commander_Pompe (Com : Boolean ) is
   begin
      Simulateur.Commander_Pompe(Com);
   exception when Socket_Error => raise Erreur_Procede;
   end Commander_Pompe;

   procedure Commander_Vanne (Com : Float ) is
   begin
      Simulateur.Commander_Vanne(Com);
   exception when Socket_Error => raise Erreur_Procede;
   end Commander_Vanne;

   procedure Commander_Alarme (On: Boolean) is
   begin
      Simulateur.Commander_Alarme(On);
   exception when Socket_Error => raise Erreur_Procede;
   end Commander_Alarme;

   function Lire_Debit_Ed return Float is
      Val : Float;
   begin
      Simulateur.Lire_Debit_Ed(Val);
      return Val;
   exception when Socket_Error => raise Erreur_Procede;
   end Lire_Debit_Ed;

   function Lire_Bouton return Boolean is
      Val : Boolean;
   begin
      Simulateur.Lire_Bouton(Val);
      return Val;
   exception when Socket_Error => raise Erreur_Procede;
   end Lire_Bouton;
```

© Dunod – La photocopie non autorisée est un délit.

```
function Lire_Debit_Ef return Float is
      Val : Float;
   begin
      Simulateur.Lire_Debit_Ed(Val);
      return Val;
   exception when Socket_Error => raise Erreur_Procede;
   end Lire_Debit_Ef;

   function Lire_Temperatures return Temperatures is
      T : Temperatures;
   begin
      Simulateur.Lire_Temperatures(T);
      return T;
   exception when Socket_Error => raise Erreur_Procede;
   end Lire_Temperatures ;

   protected body Simulateur is
      -- Objet protégé garantissant l'exclusion mutuelle des accès au
      -- simulateur
      procedure Initialiser is
         -- Connexion au simulateur
         Address : Sock_Addr_Type;
         -- Représentation d'une adresse Gnat.Sockets
      begin
         Initialisé:=True;
         -- Les autres entrées sur l'objet protégé sont dorénavant
         -- possibles
         Initialize;
         -- Initialise Gnat.Sockets

Address.Addr:=Addresses(Get_Host_By_Name(Adresse_Simulateur),1);
         Address.Port:=Port_Type(Port_Simulateur);
         -- Création de l'adresse dans le format nécessaire à
         -- Gnat.Sockets
         Create_Socket(Socket);
         Set_Socket_Option(Socket,Socket_Level,(Reuse_Address,True));
         -- Initialisation du Socket
         Connect_Socket(Socket,Address);
         -- Connexion au simulateur
         Channel:=Stream(Socket);
         -- Elaboration d'un flux de caractères (permettant des
         -- lectures et écritures -- comme dans un fichier ou à
         -- l'écran)
      end;
      entry Lire_Debit_Ef (Val :    out Float ) when Initialisé is
         -- Interroge le simulateur sur la valeur de Debit Ef
         -- Ne peut avoir lieu qu'après initialisation
      begin
         String'Output(Channel,"ef");
         Val:=Float'Value(String'Input(Channel));
      end;

      entry Lire_Debit_Ed (Val :    out Float ) when Initialisé is
         -- Interroge le simulateur sur la valeur de Debit Ed
         -- Ne peut avoir lieu qu'après initialisation
      begin
         String'Output(Channel,"ed");
         Val:=Float'Value(String'Input(Channel));
      end;
```

```
      entry Lire_Bouton (On:    out Boolean) when Initialisé is
         -- Interroge le simulateur sur l'état du bouton
         -- Ne peut avoir lieu qu'après initialisation
         V: integer;
      begin
         String'Output(Channel,"bouton");
         V:=Integer'Value(String'Input(Channel));
         if V=0 then On:=False;
         else On:=True;
         end if;
      end;
      entry Lire_Temperatures (T :    out Temperatures ) when
Initialisé is
         -- Interroge le simulateur sur la valeur des températures
         -- Ne peut avoir lieu qu'après initialisation
      begin
         String'Output(Channel,"temp");
         T.Ted1:=Float'Value(String'Input(Channel));
         T.Ted2:=Float'Value(String'Input(Channel));
         T.Tef1:=Float'Value(String'Input(Channel));
         T.Tef2:=Float'Value(String'Input(Channel));
      end;
      entry Commander_Alarme (Val : Boolean ) when Initialisé is
         -- Allume ou éteint l'alarme
         -- Ne peut avoir lieu qu'après initialisation
      begin
         if Val then
            String'Output(Channel,"alarme 1");
         else
            String'Output(Channel,"alarme 0");
         end if;
      end;
      entry Commander_Pompe (Val : Boolean ) when Initialisé is
         -- Allume ou éteint la pompe
         -- Ne peut avoir lieu qu'après initialisation
      begin
         if Val then
            String'Output(Channel,"pompe 1");
         else
            String'Output(Channel,"pompe 0");
         end if;
      end;
      entry Commander_Vanne (Val : Float ) when Initialisé is
         -- Commande la vanne
         -- Ne peut avoir lieu qu'après initialisation
      begin
         String'Output(Channel,"vanne "&Float'Image(Val));
      end;
   end;
end Simulateur;
```

© Dunod – La photocopie non autorisée est un délit.

Le programme principal

Il ne reste plus alors qu'à écrire le programme principal : celui-ci définit les trois *pragmas* usuels permettant de définir un ordonnancement à priorités fixes gérant les accès aux objets protégés avec le protocole à priorité plafond (élimine l'inversion de priorité). Rappelons qu'au moment du lancement du programme principal, toutes les tâches définies sont lancées en parallèle. Le programme principal ne se termine que lorsque toutes les tâches sont terminées (dans le cas présent : jamais).

```
pragma Queuing_Policy(Priority_Queuing);
-- L'accès aux objets protégés est géré par niveau de priorité
pragma Locking_Policy(Ceiling_Locking );
-- Utilisation du protocole à priorité plafond
pragma Task_Dispatching_Policy(Fifo_Within_Priorities);
-- Ordonnancement des tâches à priorités
with Controler_Echangeur;
procedure Echangeur is
begin
   Initialiser;
end;
```

7.4.3 Implémentation en C du système sur simulateur

Ce paragraphe présente l'implémentation en C obtenue à partir du diagramme DARTS. Comme nous l'avons vu précédemment, le langage C n'est pas nativement multitâche. Étant donné que l'application permet l'utilisation d'un PC industriel, nous prendrons comme extension temps réel la norme POSIX type 54 (voir § 5.3.1). La figure 7.10 montre une architecture logicielle typique pour la partie multitâche d'un programme de contrôle-commande en C, facilitant le passage du simulateur à la commande réelle. Ainsi, le passage de la simulation au contrôle réel sera simplifié puisque seules quelques directives *#include* devront être modifiées.

Le module de communication

Le module *BaLs* est l'équivalent en langage C du paquetage Ada *Communications*. Il n'est pas générique, puisque tout élément échangé entre deux tâches passe par le type chaîne de caractères (il subit une coercion en *char* *) : toute notion de type est perdue, et le programmeur doit s'assurer lui-même que les données envoyées sont du même type que les données reçues. Ce fonctionnement est assez typique du langage C.

Les modules *procede* et *simulateur* ont la même interface, ce qui permet, comme pour le cas traité en Ada, de ne changer que la directive *#include* présente dans *contrôle.c* afin de passer de la simulation à la commande réelle.

Commençons par observer le module *BaLs*. Les boîtes aux lettres peuvent être de taille 1 ou *n*, et être à écrasement ou sans écrasement. Comme nous l'avons vu dans le paragraphe 6.1, les éléments de communications s'appliquant aux tâches (*pthread*) proposent des sémaphores et de variables conditionnelles permettant de programmer un moniteur de Hoare. Les moniteurs permettent de programmer simplement des éléments de communication de type boîtes aux lettres : un *buffer* est alloué initialement, son rôle est de stocker le ou les messages sous forme de chaîne de caractères (tableau d'octets). On peut ensuite synchroniser les envois/réceptions suivant le

© Dunod – La photocopie non autorisée est un délit.

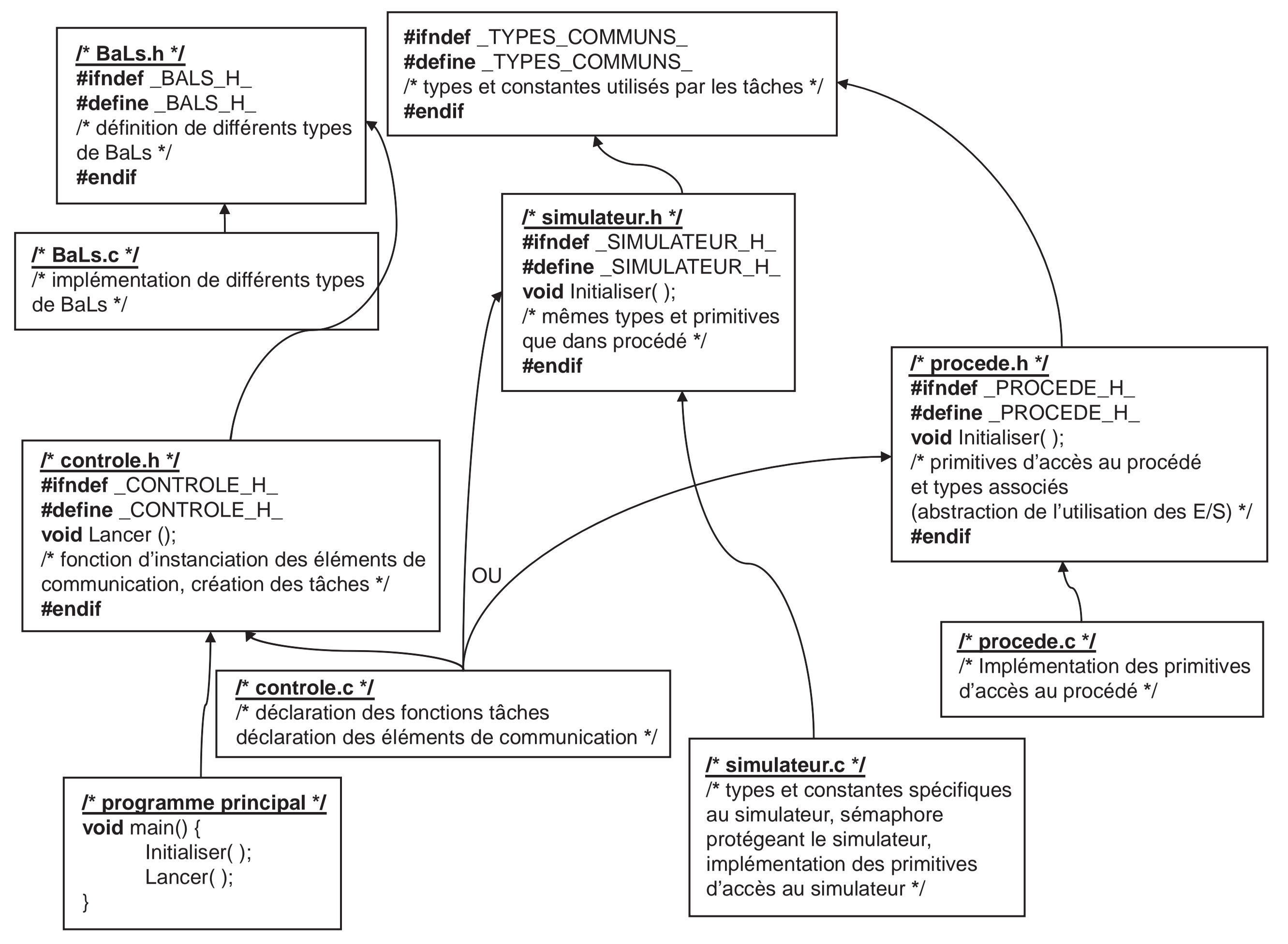


Figure 7.10 – Architecture logicielle typique d'une application de contrôle-commande simple en C.

schéma choisi. Cela revient à un problème de type producteur/consommateur lorsque les boîtes aux lettres sont sans écrasement, et à un problème de type consommateur dans le cas où les boîtes aux lettres sont avec écrasement.
Nous avons présenté au paragraphe 6.2 les différentes implémentations de boîtes aux lettres. Le code source du module est donné en annexe C.

■ Le module de contrôle

Poursuivons en présentant le module *types_communs*, contenant les définitions de type température et débit.

```
#ifndef _TYPES_COMMUNS_H_
#define _TYPES_COMMUNS_H_
typedef struct {
    float Ted1, Ted2, Tef1, Tef2;
} Temperatures_t;

typedef struct {
    float ef, ed;
} Debits_t;

#ifndef BYTE
#define BYTE unsigned char
#endif
#endif
```

L'interface du module de contrôle est telle qu'elle est décrite sur la figure 7.10 :

```
/* controle.h */
#ifndef _CONTROLER_ECHANGEUR_H_
#define _CONTROLER_ECHANGEUR_H_
void Lancer ();
/* Initialise les éléments de communication et
lance le système de tâches */
#endif
```

Et son implémentation suit les règles décrites au paragraphe 6.2.

```
#include <pthread.h>
#include "BaLs.h"
#include "simulateur.h"
#include "controler_echangeur.h"
#include "types_communs.h"
void ajouter_microsecondes(struct timespec *time, long us) {
/* Modifie la structure time afin d'y ajouter us microsecondes */
    (*time).tv_nsec+=(us%1000000)*1000; /* Ajout des microsecondes au
champ en nanosecondes */
    (*time).tv_sec+=(us/1000000)+((*time).tv_nsec/1000000000); /*
Ajout des secondes entières au champ en secondes */
    (*time).tv_nsec%=1000000000; /* Si il y avait débordement des
nanosecondes sur les secondes, il a été ajouté aux secondes dans
l'instruction précédente */
}

#define Max_Ouverture_Vanne 200.0
/* Valeur d'ouverture maximale de la vanne en l/h */

/* Module de données Températures */
pthread_mutex_t s_Temperatures;
```

```
Temperatures_t MDD_Temperatures={0.0,0.0,0.0,0.0};

/* Module de données Débits */
pthread_mutex_t s_Debits;
Debits_t MDD_Debits={0.0,0.0};

/* Module de données Panne */
pthread_mutex_t s_Panne;
BYTE MDD_Panne=0;

/* Boîte aux lettres Commande Vanne de taille 1 bloquante à écrasement
*/
bal_ecr Commande_Vanne;

void Acquerir_Temperatures() {
/* Acquiert périodiquement la température et la stocke dans le MDD
MDD_Temperatures */
    Temperatures_t T;
    struct timespec horloge ;
    pthread_cond_t Reveil; /* Variable conditionnelle utilisée par la
tâche afin de se réveiller périodiquement */
    /* Cette variable n'est jamais signalée */
    pthread_mutex_t sReveil; /* Mutex lié à la variable conditionnelle
*/
    pthread_mutex_init(&sReveil,NULL); /* Initialisation du mutex */
    pthread_cond_init(&Reveil,NULL); /* Initialisation de la variable
conditionnelle */
    clock_gettime(CLOCK_REALTIME, &horloge); /* heure courante de
l'horloge au démarrage de la tâche */
    while (1) {
        T=Lire_Temperatures();
        if (!Verifier_Temperatures(T)) {
        /* Une température est hors domaine */
                pthread_mutex_lock(&s_Panne);
                /* On prévient qu'il y a panne */
                MDD_Panne=1;
                pthread_mutex_unlock(&s_Panne);
        }
        pthread_mutex_lock(&s_Temperatures);
        /* On stocke la température dans le MDD */
        MDD_Temperatures=T;
        pthread_mutex_unlock(&s_Temperatures);
        ajouter_microsecondes(&horloge,1000000);/* Calcul de la date
du prochain reveil=date du dernier réveil+1 seconde */
        pthread_mutex_lock(&sReveil);
        pthread_cond_timedwait(&Reveil, &sReveil, &horloge);
        /* Cette variable n'étant pas signalée, c'est au timeout que
cette instruction se termine */
    }
}

void Afficher() {
/* Affiche périodiquement le débit et la température dans un graphe */
    Temperatures_t T;
    Debits_t D;

    /* Gestion de la périodicité */
    struct timespec horloge ;
    pthread_cond_t Reveil;
```

© Dunod – La photocopie non autorisée est un délit.

```
    pthread_mutex_t sReveil;
    pthread_mutex_init(&sReveil,NULL);
    pthread_cond_init(&Reveil,NULL);
    clock_gettime(CLOCK_REALTIME, &horloge);
    while (1) {
        pthread_mutex_lock(&s_Temperatures);
        /* Lecture des températures */
        T=MDD_Temperatures;
        pthread_mutex_unlock(&s_Temperatures);
        pthread_mutex_lock(&s_Debits);
        /* Lecture des débits */
        D=MDD_Debits;
        pthread_mutex_unlock(&s_Debits);
        /* Affichage */
        afficher_courbes(T,D);

        /* Attente de la prochaine période */
        ajouter_microsecondes(&horloge,1000000);
        pthread_mutex_lock(&sReveil);
        pthread_cond_timedwait(&Reveil, &sReveil, &horloge);
    }
}

void Piloter_Vanne() {
/* Applique une commande arrivant dans la BaL Commande_Vanne sur la
vanne */
    float com;
    while (1) {
        /* Attente d'un message */
        bal_ecr_recevoir(Commande_Vanne,(char*)&com);
        /* Commande de la vanne */
        Commander_Vanne(com);
    }
}

void Reguler_Debit () {
/* Acquiert périodiquement et vérifie les débits, élabore une consigne
de commande de vanne,
qu'elle envoie dans la BaL Commande_Vanne, gère les pannes */
    Debits_t D;
    Temperatures_t T;
    float Consigne_Ed;
    BYTE panne;
    /* Modes de fonctionnement */
#define MODE_NORMAL 0
#define MODE_ALARME 1
    /* Mode courant de fonctionnement */
    BYTE Etat=MODE_NORMAL;
    /* Gestion de la périodicité */
    struct timespec horloge ;
    pthread_cond_t Reveil;
    pthread_mutex_t sReveil;
    pthread_mutex_init(&sReveil,NULL);
    pthread_cond_init(&Reveil,NULL);
    clock_gettime(CLOCK_REALTIME, &horloge);
    while (1) {
       switch (Etat) {
       case MODE_NORMAL:
              pthread_mutex_lock(&s_Panne);
```

```
                /* Lecture de l'état de panne */
                panne=MDD_Panne;
                pthread_mutex_unlock(&s_Panne);
                if (panne) {
                /* Le système est en panne */
                  Etat=MODE_ALARME;
                  /* Mise en route de l'alarme */
                  Commander_Alarme(1);
                 /*Ouverture de la vanne d'eau industrielle au maximum*/
                  Consigne_Ed=Max_Ouverture_Vanne;
                  /* Envoie à la tâche de commande */
                  bal_ecr_envoyer(Commande_Vanne,(char *)&Consigne_Ed);
                } else {
                  /* Acquisition des débits */
                  D.ed=Lire_Debit_Ed();
                  D.ef=Lire_Debit_Ef();
                  pthread_mutex_lock(&s_Debits);
                  /* Stockage dans le MDD */
                  MDD_Debits=D;
                  pthread_mutex_unlock(&s_Debits);
                  pthread_mutex_lock(&s_Temperatures);
                  /* Lecture des températures */
                  T=MDD_Temperatures;
                  pthread_mutex_unlock(&s_Temperatures);
                  /* Elaboration de la consigne */
                  Consigne_Ed=Elaborer_Consigne(D,T);
                  /* Envoie à la tâche de commande */
                  bal_ecr_envoyer(Commande_Vanne,(char*)&Consigne_Ed);
               }
               break;
         case MODE_ALARME :
        /* L'alarme est déclenchée, on attend que l'opérateur la valide
*/
               if (Lire_Bouton()) {
                  /* L'opérateur a validé l'alarme */
                  /* Extinction de l'alarme */
                  Commander_Alarme(0);
                  /* Fermeture de la vanne d'eau industrielle */
                  Consigne_Ed=0.0;
                  bal_ecr_envoyer(Commande_Vanne,(char*)&Consigne_Ed);
                  /* Extinction de la pompe d'eau distillée */
                  Commander_Pompe(0);
                  return; /* Terminaison de la tâche */
               }
               break;
         default: ;
         }
         /* Attente de la prochaine période */
         ajouter_microsecondes(&horloge,1000000);
         pthread_mutex_lock(&sReveil);
         pthread_cond_timedwait(&Reveil, &sReveil, &horloge);
     }
}

void Lancer () {
    int i ;
    pthread_t taches[4]; /* Tableau stockant les identificateurs de
tâches */
```

© Dunod – La photocopie non autorisée est un délit.

```
    pthread_attr_t attributes; /* Attributs utilisés lors de la
création des tâches */
    /* Création des sémaphores d'exclusion mutuelle */
    pthread_mutex_init(&s_Temperatures,0);
    pthread_mutex_init(&s_Debits,0);
    pthread_mutex_init(&s_Panne,0);
    /* Création de la boîte aux lettres */
    Commande_Vanne=bal_ecr_init(sizeof(float));
    Commander_Pompe(1); /* Allumage de la pompe d'eau distillée */
    /* Lancement des tâches */
    pthread_attr_init(&attributes);
    /* Remarque: plusieurs systèmes POSIX non temps réel ne permettent
pas la manipulation d'attributs */
    pthread_create(&taches[0], &attributes,(void *)
Acquerir_Temperatures, (void *)0);
    pthread_create(&taches[1], &attributes,(void *) Afficher, (void
*)0);
    pthread_create(&taches[2], &attributes,(void *) Piloter_Vanne,
(void *)0);
    pthread_create(&taches[3], &attributes,(void *) Reguler_Debit,
(void *)0);
    /* Afin d'éviter que le programme principal ne se termine juste
après le lancement des tâches, toutes les tâches sont attachées */
    for (i=0 ;i<4 ;i++) {
        pthread_join(taches[i],NULL); /* Attente de terminaison */
    }
}
```

Le module de communication avec le simulateur

Voyons maintenant comment le programme C communique par TCP avec le simulateur. Nous utiliserons pour cela le même protocole que celui utilisé en Ada : une chaîne de caractères est toujours précédée de deux entiers au format binaire : le premier vaut 1, le second vaut la taille en octets de la chaîne transmise.
Il est important de constater que différentes tâches peuvent en parallèle tenter d'accéder au simulateur. Il est donc nécessaire de protéger les accès au simulateur afin de garantir leur exclusion mutuelle. C'est le rôle du mutex *s_Simulateur*, utilisé par chaque sous-programme accédant au simulateur. Il est primordial d'initialiser le simulateur avant tout autre accès à celui-ci (c'est-à-dire avant la création des tâches).
Voici l'en-tête du module :

```
/*
    simulateur.h
    Un simulateur est défini par un serveur TCP écoutant un port
(par défaut 34532). Au début de l'application, une connexion TCP est
ouverte avec le simulateur. Le protocole employé est de type commande
du système/réponse du procédé. Par exemple, pour lire le débit ef, la
fonction Lire envoie "ef", le simulateur répond alors en envoyant un
flottant sous forme de chaîne de caractères. Pour commander la pompe,
le simulateur envoie "pompe 0" ou "pompe 1" en fonction de la valeur
de la commande
*/
#ifndef _SIMULATEUR_H_
#define _SIMULATEUR_H_

#include "types_communs.h"
extern const float Max_Ouverture_Vanne;
```

```
/* Consigne maximale en litre/heure de l'ouverture de vanne */
int Initialiser();
/* Initialisation du simulateur
   Retour: 0 si connexion, 1 si erreur
*/
float Lire_Debit_Ef();
/* Renvoie le débit d'eau industrielle en l/h
   Nécessite: simulateur initialisé préalablement*/
float Lire_Debit_Ed();
/* Renvoie le débit d'eau distillée en l/h
   Nécessite: simulateur initialisé préalablement */
Temperatures_t Lire_Temperatures();
/* Renvoie les températures en °C
   Nécessite: simulateur initialisé préalablement */
BYTE Lire_Bouton();
/* Renvoie l'état du bouton poussoir
 Retourne: vrai si appuyé, faux sinon
   Nécessite: simulateur initialisé préalablement */
void Commander_Pompe (BYTE Com );
/* Allume ou éteint la pompe
   Entraine: Com = 1 => allume la pompe
             Com = 0 => éteint la pompe
   Nécessite: simulateur initialisé préalablement */
void Commander_Vanne (float Com );
/* Commande l'électrovanne en l/h
   Nécessite: simulateur initialisé préalablement */
void Commander_Alarme(BYTE On);
/* Allume ou éteint l'alarme
   Entraine: On = 1 => allume l'alarme
             On = 0 => éteint l'alarme
   Nécessite: simulateur initialisé préalablement */

#endif
```

Une partie du corps du module est donnée ci-après :

```
#include <pthread.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include "simulateur.h"

static const char * Adresse_Simulateur = "127.0.0.1";
/* Adresse IP du simulateur */

static unsigned short Port_Simulateur=34532;
/* Port sur lequel le simulateur lance un serveur TCP */

static int sock_sim;
/* Socket utilisé */

#define TAILLE_BUFFER 256
/* Taille du buffer de réception des messages */

const float Max_Ouverture_Vanne=200.0; /* l/h */
static pthread_mutex_t s_Simulateur;
/* Assure l'exclusion mutuelle des accès au simulateur */

int Initialiser() {
    struct sockaddr_in addr_sim;
    /* Adresse du simulateur */
```

© Dunod – La photocopie non autorisée est un délit.

```
    pthread_mutex_init(&s_Simulateur,0);
    /* Initialisation du sémaphore */
    if ((sock_sim = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        return 1;
    /* Initialisation de la connexion TCP */
    memset(&addr_sim, 0, sizeof(addr_sim));
    addr_sim.sin_family      = AF_INET;
    addr_sim.sin_addr.s_addr = inet_addr(Adresse_Simulateur);
    addr_sim.sin_port        = htons(Port_Simulateur);
    if (connect(sock_sim, (struct sockaddr *) &addr_sim,
sizeof(addr_sim)) < 0)
        return 1;
    /* Connexion TCP */
}

static int Simu_Read(char *buf) {
/* Lit une chaîne de caractères provenant du simulateur
   Nécessite: buf a au moins la taille de TAILLE_BUFFER
   Entraine: buf contient la chaîne lue (au plus TAILLE_BUFFER
caractères) terminée par '\0'
   Retourne: taille de la chaîne lue, ou -1 si erreur
*/
    int entier1,longueur;
    /* Le premier entier d'une chaîne reçue vaut toujours 1, et le
second vaut la longueur en octet de la chaîne */
    int nb_octets;
    if ((nb_octets = recv(sock_sim, &entier1, 4, 0)) <= 0) return -1;
    if ((nb_octets = recv(sock_sim, &longueur, 4, 0)) <= 0) return -1;
    if (longueur<0||longueur>TAILLE_BUFFER-1) return -1;
    if ((nb_octets = recv(sock_sim, buf, longueur, 0)) != longueur)
return -1;
    buf[longueur]=0;
    return longueur;
}

static int Simu_Read_Float(float *f) {
    static char buf[TAILLE_BUFFER];
    int result=Simu_Read(buf);
    sscanf(buf,"%f",f);
    return result;
}

static int Simu_Write(const char *buf) {
/* Envoie une chaîne de caractères au simulateur
   Nécessite: buf est une chaîne terminée par '\0', de taille maximale
TAILLE_BUFFER
   Entraine: buf est envoyé vers le simulateur
   Retourne: taille de la chaîne envoyée, ou -1 si erreur
*/
    static char tampon[TAILLE_BUFFER+8];
    /* Taille maximale de la chaîne + 2 entiers */
    int un=1;
    int longueur=strlen(buf);
    int nb_octets;
    memcpy(tampon,(char *)&un,4);
    memcpy(&tampon[4],(char *)&longueur,4);
    memcpy(&tampon[8],buf,longueur);
    /* Le premier entier d'une chaîne envoyée vaut toujours 1, et le
second vaut la longueur en octet de la chaîne */
```

```
    if ((nb_octets = send(sock_sim, tampon, longueur+8, 0)) !=
longueur+8) return -1;
    return longueur;
}

float Lire_Debit_Ef() {
    float lu;
    pthread_mutex_lock(&s_Simulateur);
    Simu_Write("ef");
    Simu_Read_Float(&lu);
    pthread_mutex_unlock(&s_Simulateur);
    return lu;
}
```

Le programme principal

Le programme principal a pour rôle d'initialiser le simulateur et d'appeler la fonction de lancement des tâches. Noter que contrairement à Ada, le programme principal se termine immédiatement (ici, une attente artificielle de caractère a été ajoutée afin d'empêcher à la fonction *main* de se terminer, et ainsi d'arrêter le système). Il est possible, lorsque les attributs de tâches sont implémentés, de donner le même comportement à la fonction *main* qu'au programme principal Ada.

```
#include "controler_echangeur.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    char c;
    Initialiser();/* Initialisation du procédé ou simulateur */
    Lancer();/* Lancement des tâches et attente de terminaison */
    return 0;
}
```

7.4.4 Implémentation en LabVIEW du système sur simulateur

L'implémentation LabVIEW obtenue à partir d'un diagramme DARTS peut être visuellement très proche de la conception. En effet, grâce à l'encapsulation, il est possible de créer des tâches ayant un aspect graphique évoquant les éléments DARTS. La figure 7.11 montre la hiérarchie de *vi* utilisée pour implémenter le système. Nous avons choisi d'utiliser la même technique de simulation (communication TCP/IP avec un processus de simulation) que dans les langages Ada et C, mais nous aurions pu très simplement implémenter le simulateur par une tâche LabVIEW, et le faire communiquer avec les tâches du système via des modules de données.

Il est à noter que tout type utilisé, en dehors de types de base, est déclaré en tant que *Type Strict* ; ainsi, dans le cas présent, ce sont les *clusters* utilisés pour les Températures et les Débits, ainsi que les types énumérés pouvant être utilisés par plusieurs *vi* (actions possibles sur un module de données, actions possibles sur le simulateur). Ainsi, la figure 7.12 présente la définition du type *Températures*.

© Dunod – La photocopie non autorisée est un délit.

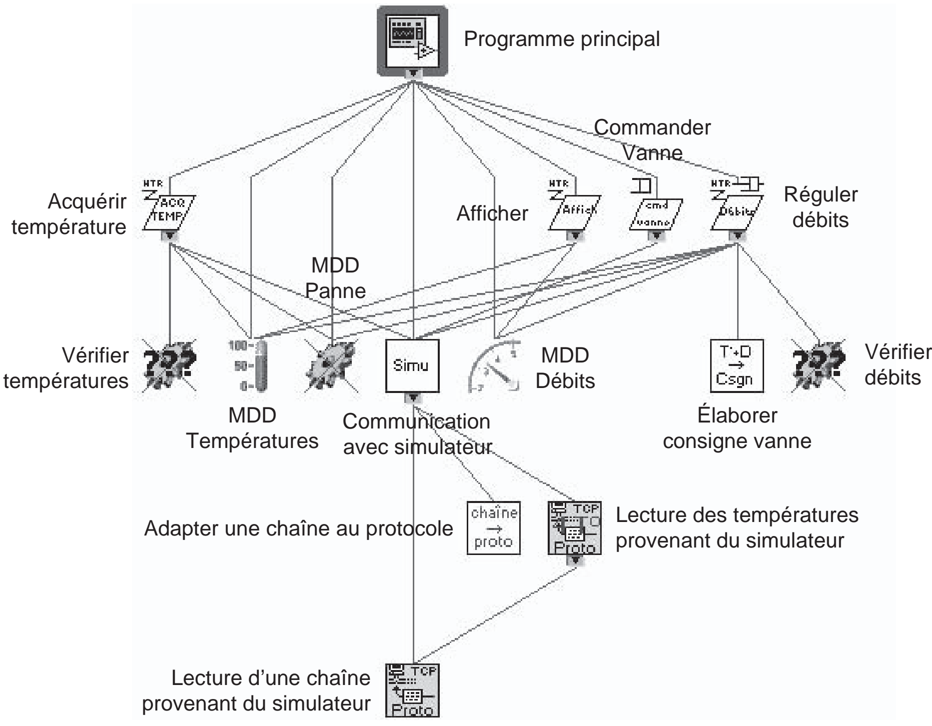


Figure 7.11 – Hiérarchie des *vi* du système de contrôle de l'échangeur en LabVIEW.

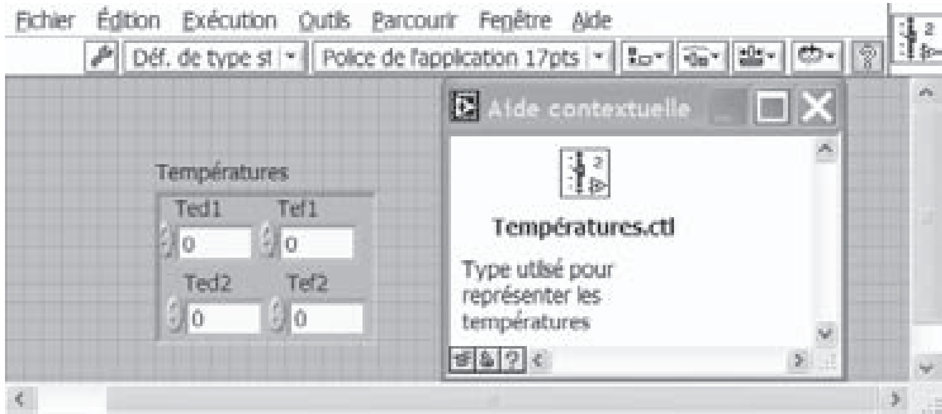


Figure 7.12 – L'utilisation de types stricts est fortement conseillée : exemple du type *Températures*.

Outils de communication

La boîte aux lettres peut être simplement implémentée par un élément *message queue* LabVIEW : cette boîte est créée au plus haut niveau, dans le programme principal, et son identifiant est passé comme paramètre aux deux tâches l'utilisant. Lors de la création d'une boîte aux lettres, à partir de la version 7 de LabVIEW, le type d'éléments transportés par la boîte aux lettres est donné à la création. Ainsi, sur la figure 7.13, la création de la boîte *BaL Commande Vanne* se voit fournir une constante réelle (double précision). L'identifiant de boîte aux lettres créée est alors passé aux tâches *Réguler Débit* et *Commander Vanne* qui pourront ainsi communiquer à l'aide de cette boîte aux lettres.

Les modules de données sont implémentés à l'aide de *vi* non réentrants, comme expliqué dans le paragraphe 6.4.3, p. 335. Généralement, on donne la possibilité d'initialiser, lire et modifier un module de données. Remarquons sur la figure 7.13 la façon dont on initialise les modules de données avant de lancer les tâches grâce à l'utilisation d'une structure séquence. En guise d'exemple de module de données, la figure 7.14 montre le diagramme de *MDD Débits*. Noter l'utilisation d'un type strict pour les débits.

Les communications avec le simulateur sont elles aussi accessibles à travers un *vi* non réentrant, ce qui garantit l'exclusion mutuelle des accès à celui-ci.

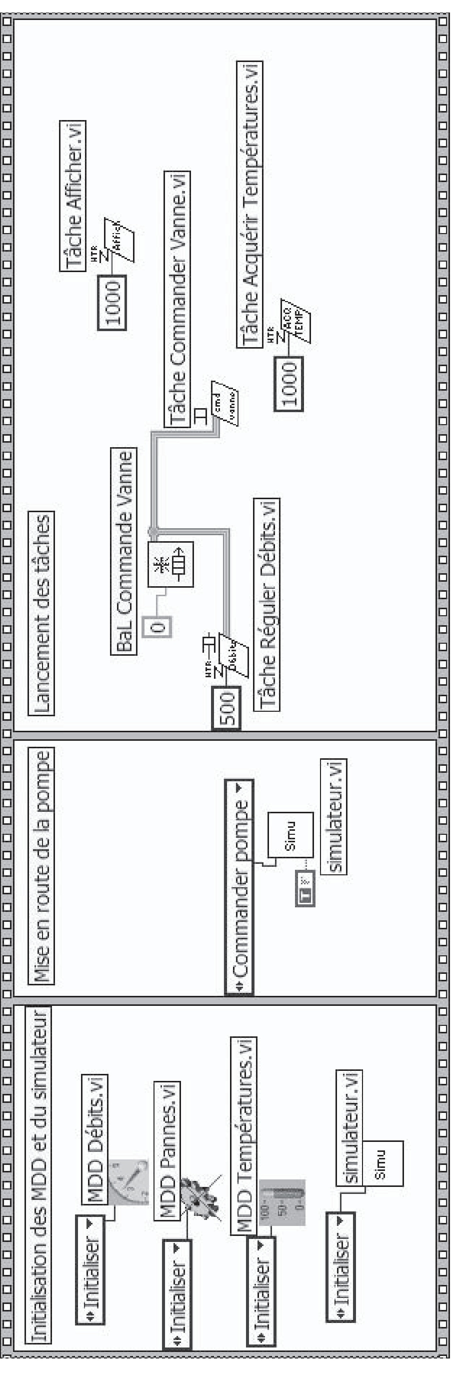


Figure 7.13 – Programme principal LabVIEW du contrôle de l'échangeur.

© Dunod – La photocopie non autorisée est un délit.

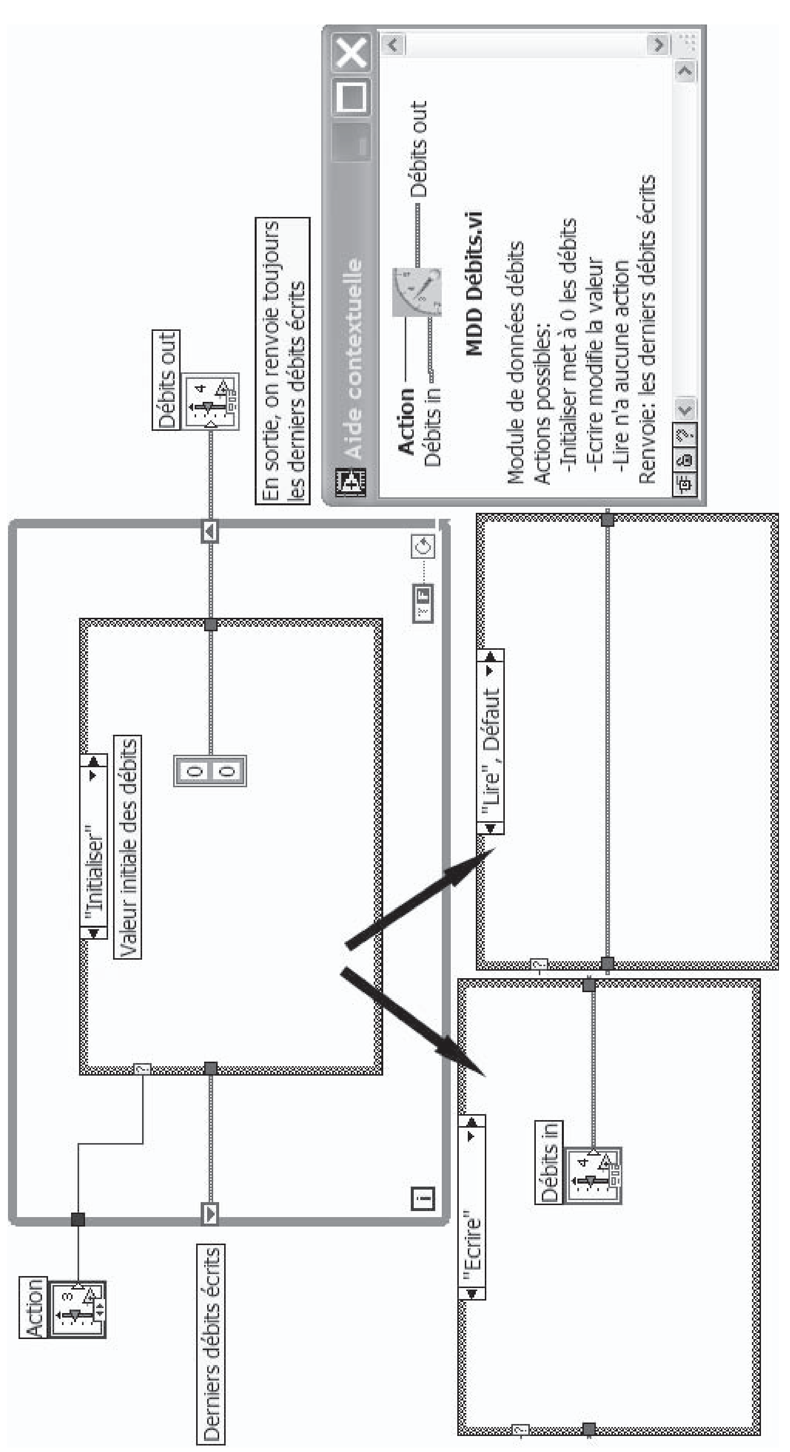


Figure 7.14 – Module de données *Débits* implémenté par *vi* non réentrant.

■ Implémentation des tâches

Dans le cas présent, il y a deux types de tâches : des tâches périodiques, et une tâche déclenchée par boîte aux lettres.

Chaque tâche est implémentée par une boucle sans fin. Dans le cas périodique, on utilise le *vi Attendre le prochain multiple de* (voir § 6.4.3, p. 334) qui permet d'assurer un régime périodique sans dérive des horloges (activation à chaque fois que l'horloge atteint un multiple de *x* millisecondes). La figure 7.15 présente la tâche *Acquérir Températures*. Noter qu'il est conseillé, pour faciliter la lecture du programme, de choisir des icônes exprimant la fonctionnalité des *vi*. Ainsi, celui de la tâche ressemble à une tâche DARTS périodique. Noter aussi que le paramètre *Période*, passé par le programme principal, est un paramètre que nous avons rendu obligatoire. Enfin, comme tout *vi* qui doit être correctement documenté, il convient pour les tâches de mettre en évidence dans leur documentation quels éléments de communication et éléments matériels sont utilisés.

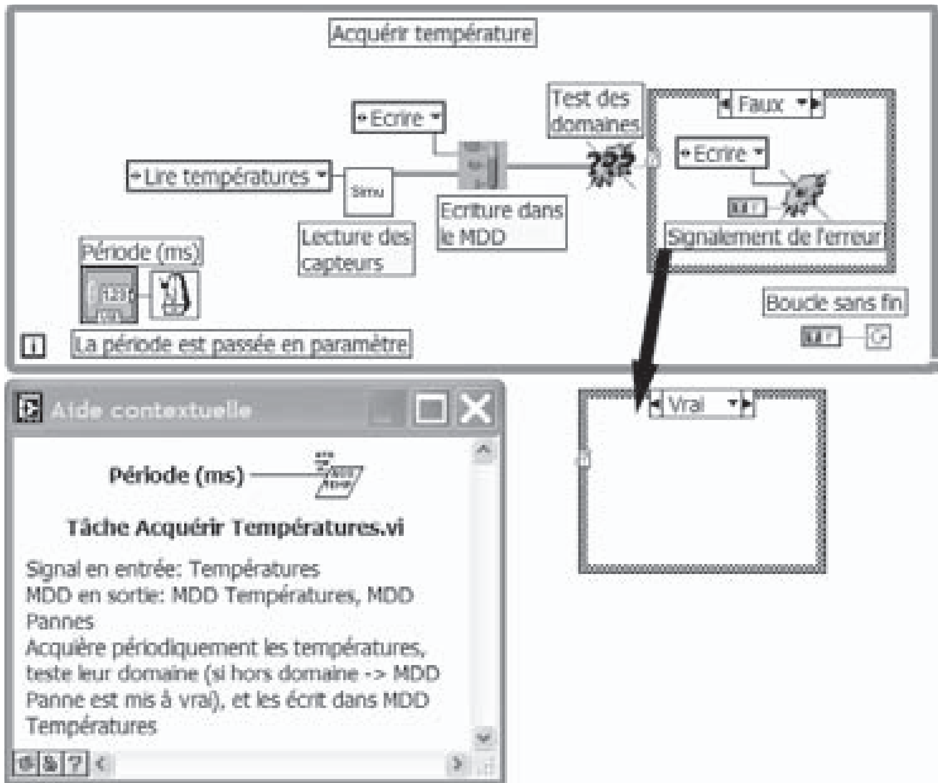


Figure 7.15 – Diagramme de la tâche Acquérir Débits.

La tâche *Afficher* est elle aussi très simple, et est présentée sur la figure 7.16. On peut remarquer la facilité avec laquelle LabVIEW permet de réaliser des affichages (voir la face avant d'*Afficher* sur la figure 7.17).

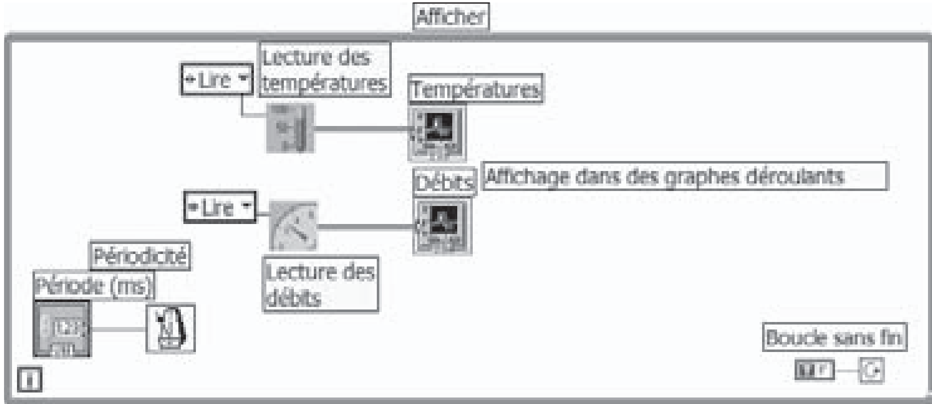


Figure 7.16 – Diagramme de la tâche *Afficher*.

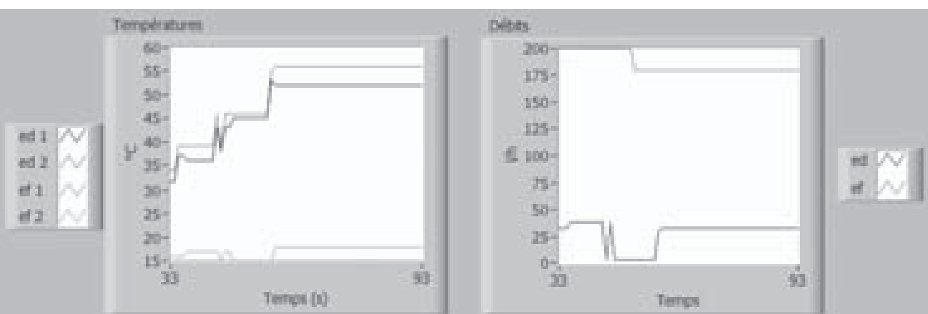


Figure 7.17 – Face avant du *vi Afficher*.

La tâche la plus complexe est *Réguler Débits* (figure 7.18). Elle intègre le diagramme états-transitions du processus de contrôle défini en SA-RT. Cet automate est implémenté en LabVIEW grâce à une structure de choix, dont la condition est un type énuméré dont les valeurs possibles sont les états de l'automate. Dans le cas présenté, les 3 états du processus de contrôle ont été implémentés par 2 états (*Normal* et *Attente bouton*), le 3e état, *Panne*, correspond à l'arrêt de la tâche et n'est donc pas représenté dans le type énuméré. L'identificateur de boîte aux lettres donné en paramètre (par le programme principal) permet d'envoyer des consignes vers la tâche de commande.

Enfin, la tâche *Commande Vanne* est une tâche déclenchée par boîte aux lettres. Elle prend donc en entrée un identificateur de boîte aux lettres, se contente d'attendre un message et d'appliquer la commande sur le procédé ou le simulateur (figure 7.19).

© Dunod – La photocopie non autorisée est un délit.

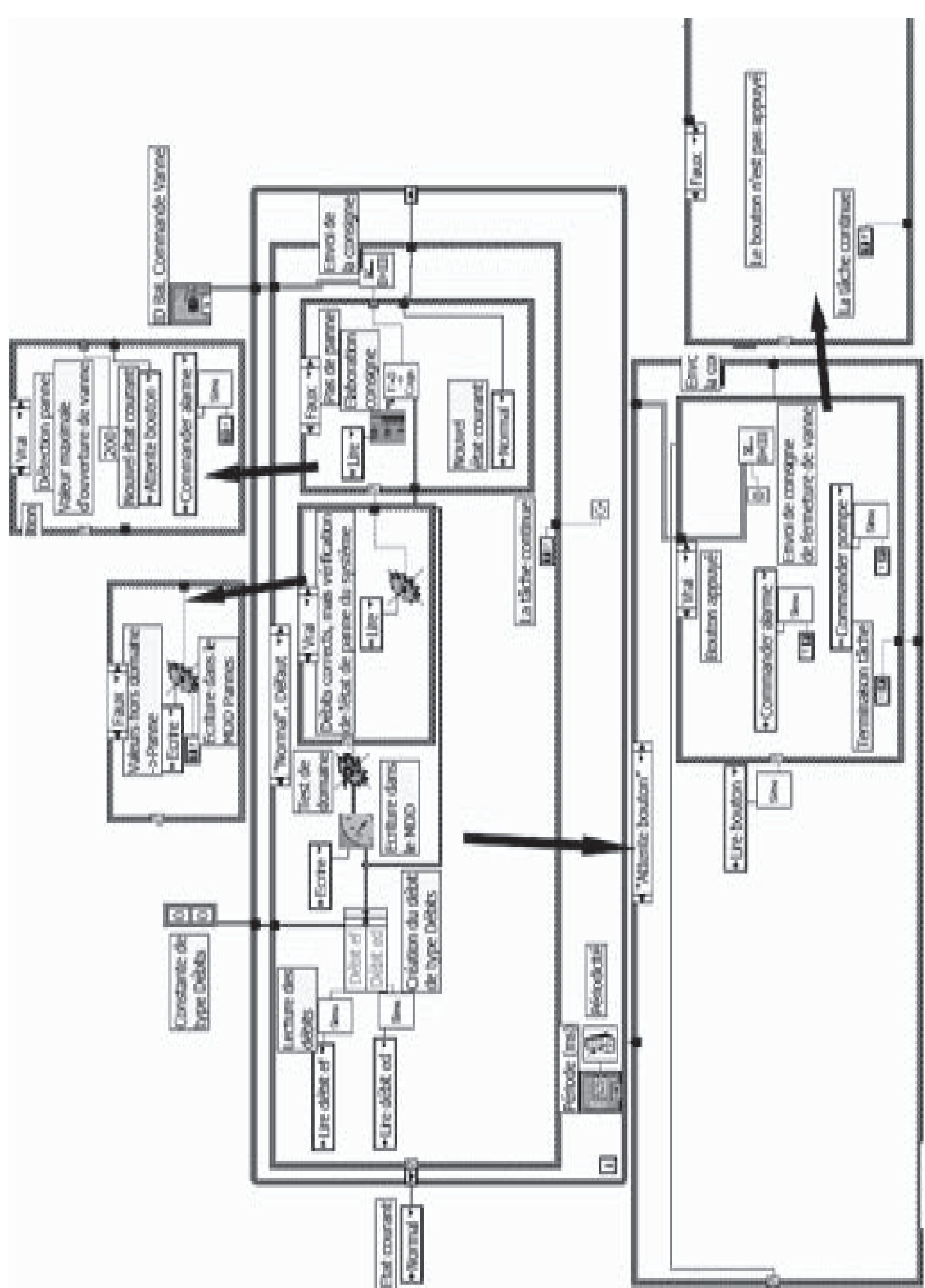


Figure 7.18 – Diagramme de la tâche *Réguler Débit*.

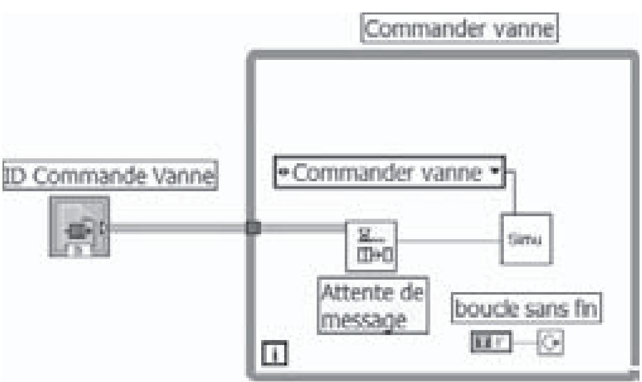


Figure 7.19 – Diagramme de la tâche *Commander Vanne*.

Le choix effectué dans cet exemple est d'utiliser l'appel au simulateur dans chaque tâche communiquant avec le procédé. C'est exactement comme si en Ada (voir § 7.4.2), nous avions directement utilisé l'objet protégé Simulateur dans les tâches. Nous aurions pu ici aussi définir des fonctions d'accès élémentaires au procédé, de façon à ne pas appeler le simulateur directement dans les tâches.

■ Le vi de communication avec le simulateur

Rappelons qu'en LabVIEW, il aurait été tout aussi simple, voire même préférable, d'implémenter le simulateur directement dans une tâche LabVIEW. Cependant, dans un souci d'homogénéité avec ce qui a été fait en C et Ada, nous avons choisi d'utiliser un simulateur sous forme de processus externe, avec lequel nous communiquons via TCP/IP (voir § 7.4.1).

Le *vi* de communication avec le simulateur doit garantir l'exclusion mutuelle des communications avec le simulateur : cela est fait naturellement par le comportement non réentrant du *vi*. Ce *vi* peut être vu comme un module de donnée de haut niveau. Notons cependant que contrairement au moniteur, l'aspect non réentrant des *vi* ne permet pas de représenter une primitive gardée (de type `entry` Ada). Par conséquent, on ne peut pas imposer simplement tout appel au *vi* d'être précédé par un appel d'initialisation (comparativement, observer comment on garantit en Ada qu'initialiser est appelé avant tout autre accès).

La figure 7.20 montre l'initialisation du *vi* de communication avec le simulateur : il s'agit simplement d'une connexion TCP avec un serveur.

Nous voyons sur la figure 7.20 l'inconvénient d'utiliser une telle structure pour gérer les accès au simulateur : de nombreuses entrées et sorties du *vi* ne servent que dans certains cas d'*Action*. Il aurait aussi été possible de définir une fonction par action (acquisition ou commande) et de les protéger par sémaphore.

Afin d'illustrer le fonctionnement des envois et réceptions TCP, la figure 7.21 présente le cas correspondant à la lecture du débit d'eau industrielle : la chaîne de caractère *ef* est émise, en respectant le protocole utilisé en C et en Ada. Le rôle du *vi chaîne en chaîne protocole* est, étant donnée une chaîne en entrée, de la faire précéder de deux fois quatre octets : un entier valant 1, l'autre valant la longueur de la chaîne

© Dunod – La photocopie non autorisée est un délit.

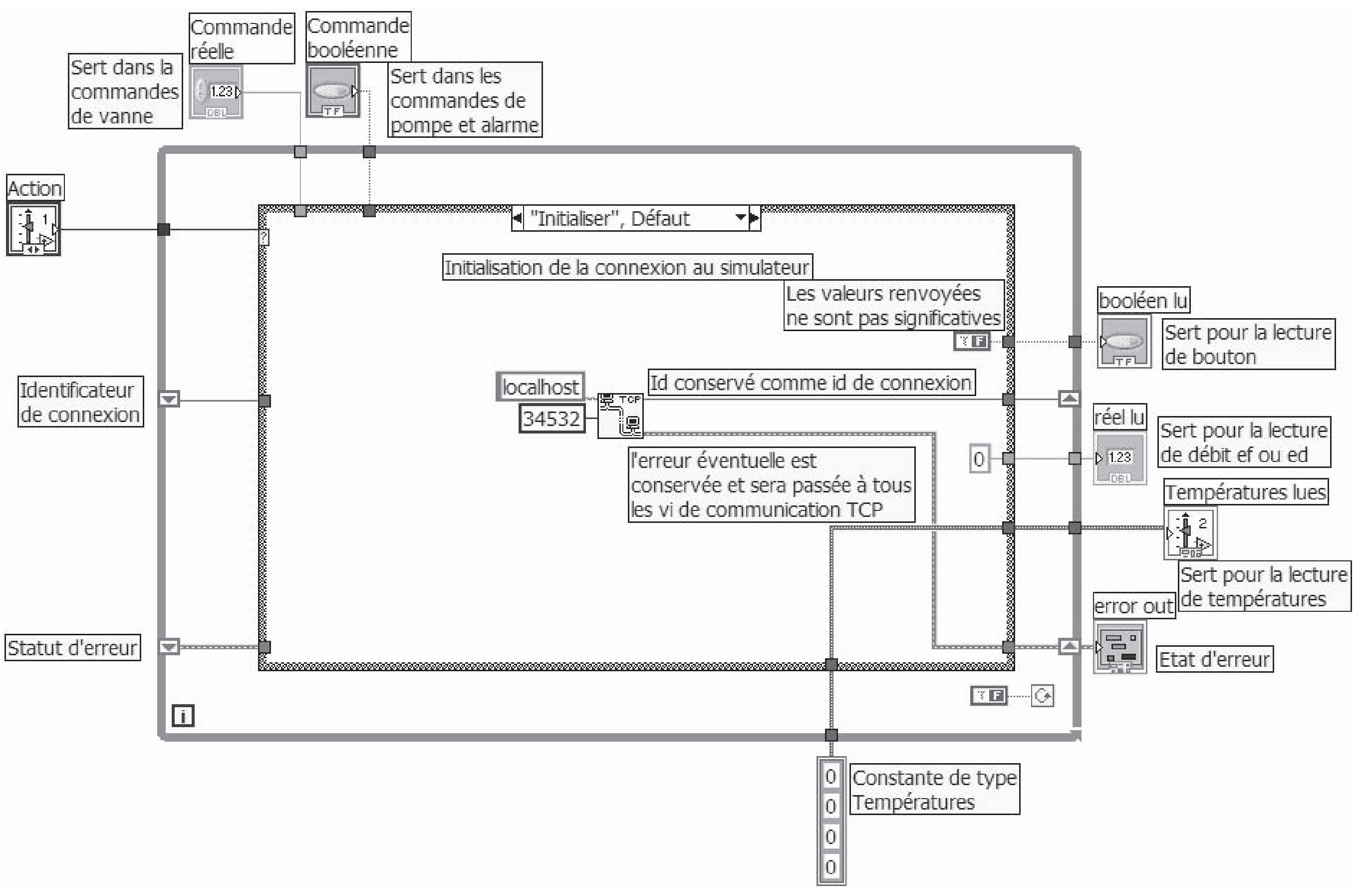


Figure 7.20 – Initialisation du *vi* de communication avec le simulateur.

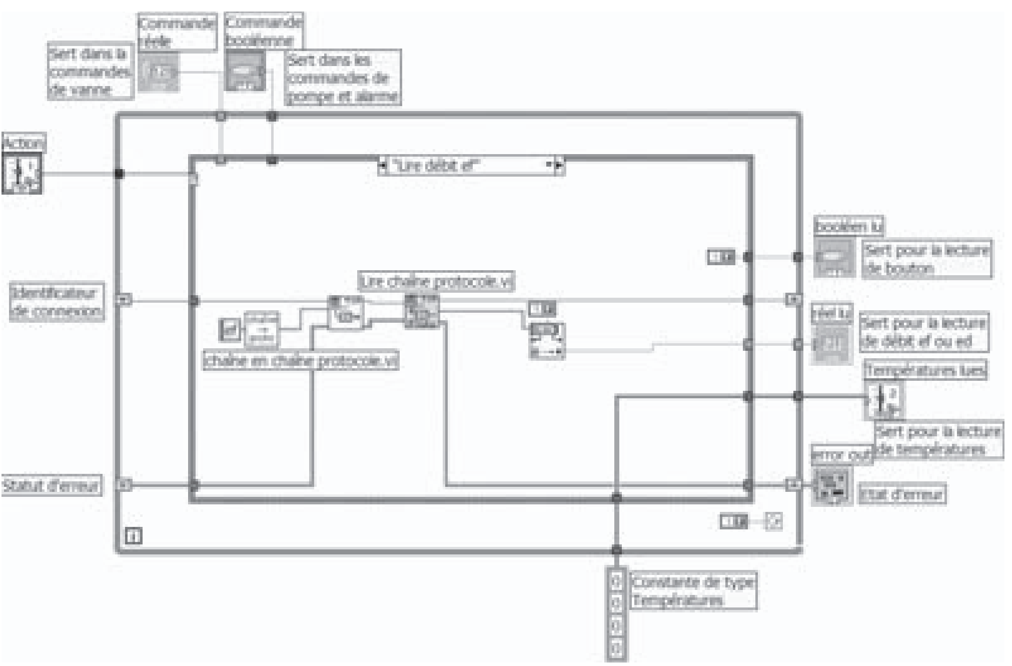


Figure 7.21 – Communication avec le simulateur : cas de la lecture de débit.

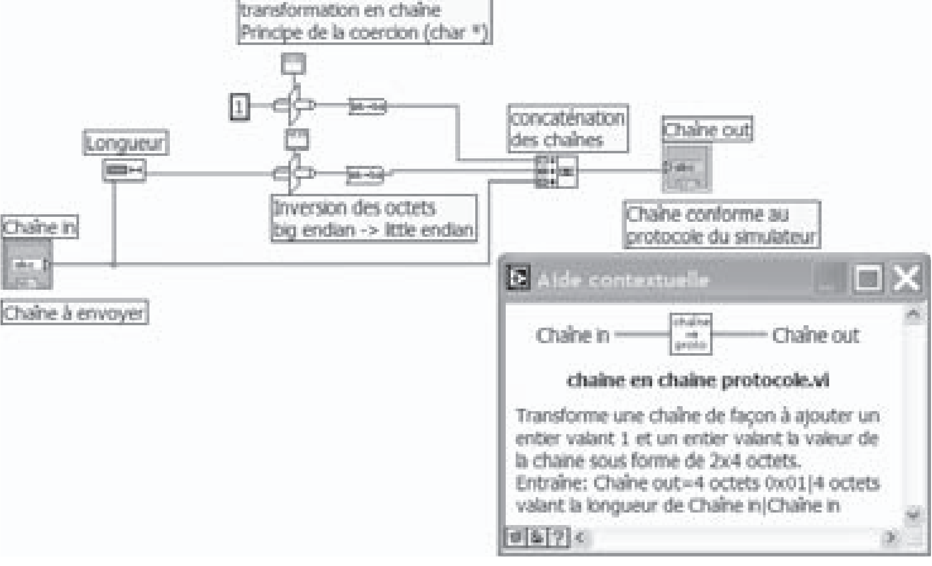


Figure 7.22 – Mise en forme d'une chaîne à envoyer suivant le protocole du simulateur.

© Dunod – La photocopie non autorisée est un délit.

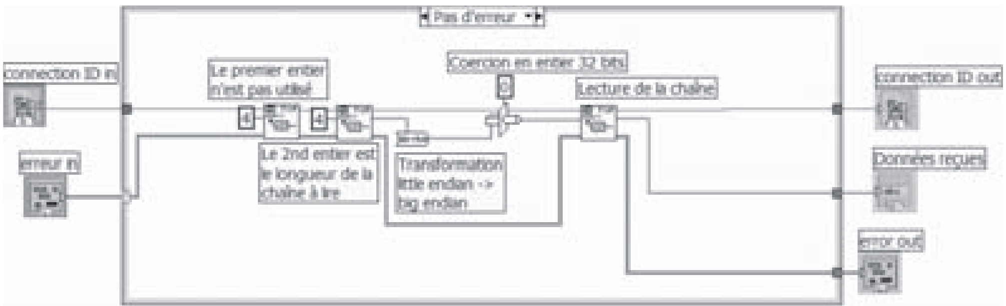


Figure 7.23 – Lecture d'une chaîne en provenance du simulateur.

(figure 7.22). De façon duale, la lecture d'une chaîne en provenance du simulateur, effectuée dans le *vi Lire chaîne protocole* utilise le deuxième entier reçu pour connaître le nombre de caractères à lire (figure 7.23).

Remarque

Afin d'homogénéiser les communications entre différentes applications LabVIEW pouvant tourner sur différents systèmes d'exploitation, National Instruments® a fait le choix de toujours utiliser une représentation de type « big endian » (l'octet de poids fort est mis en premier lors de la sérialisation d'un entier) alors que sur le système employé, le simulateur utilise une représentation de type « little endian » (octet de poids faible d'abord). Nous avons donc dû inverser les octets avant tout envoi ou après toute réception afin de ne dialoguer qu'en « little endian ». Cet exemple montre bien qu'il est en général préférable d'utiliser un protocole purement textuel : typiquement, chaque commande pourrait être terminée par les deux caractères *Carriage Return*/*Line Feed* (CR/LF), qui serviraient alors de marqueurs de fin de commande ou de réponse. Une lecture de données en provenance du réseau s'effectuerait donc jusqu'au CR/LF, et il ne serait pas nécessaire de préfixer chaque envoi par sa longueur.

7.5 Spécification et conception adaptées

Quel que soit le langage choisi, après une phase de test sur le simulateur, le concepteur peut passer à la commande réelle. Avant cela, il convient de revenir sur la spécification, et de préciser certains éléments inhérents au matériel choisi. En effet, il est probable que dans le premier V du développement en W du système, le matériel ne soit pas connu, ou connu partiellement, ou bien en cours de développement, etc.

Le tableau 7.5 montre que le système n'agit finalement pas directement sur les éléments mais par le biais d'une carte d'acquisition, d'un port série, et de différents relais électriques ainsi que d'une centrale d'acquisition de thermocouples.

En effet, les tensions (*Températures*) délivrées par les thermocouples sont trop faibles pour être lues par une carte d'acquisition standard. Contrairement à une carte d'acquisition de haute précision, une carte d'entrées/sorties standard peut lire des tensions analogiques avec une sensibilité d'environ 2,5 mV (pour un thermocouple de type K, une différence de 2,5 mV peut correspondre à un écart de 60 °C). Une centrale de conditionnement sera donc utilisée. Cette centrale, connectée à un port série (RS-232), est une centrale supportant un protocole propriétaire de type texte,

Tableau 7.5 – Description des éléments matériels choisis.

Éléments matériels	
Console	PC industriel.
Carte d'acquisition	Carte d'acquisition standard 8 entrées/2 sorties analogiques (0-10 V), 24 E/S numériques (0-5 V).
$T°ef_1$	Thermocouple de type K situé à 30 cm de l'arrivée d'eau industrielle.
$T°ef_2$	Thermocouple de type K situé à la sortie de l'échangeur.
$T°ed_1$	Thermocouple de type K situé à l'entrée de la cuve.
$T°ed_2$	Thermocouple de type K situé à la sortie de la cuve.
Thermocouple de type K	Domaine de valeur : – 270-1 372 °C. Signal fourni : – 6,458 mV-54,819 mV.
Centrale d'acquisition	Centrale reliée aux thermocouples accessible par connexion série. Programmable, fonctionne à 1 Hz (1 acquisition par seconde). **Reliée au port série COM1** et aux 4 thermocouples.
Bouton poussoir	Bouton (avec antirebond). Alimenté en 5 V cc, délivre 5 V si appuyé, 0 V sinon. **Relié à la ligne 0 du port numérique 0** de la carte d'acquisition.
Dbt_ef	Débitmètre de type vortex situé à la sortie de l'échangeur. Alimentation : 24 V cc reliée (via relais électrique 0-5 V cc → 0-24 V cc) **à la ligne 1 du port numérique 0** de la carte d'acquisition. Signal fourni : [0-10 V] **relié à l'entrée analogique 0** de la carte d'acquisition.
Dbt_ed	Débitmètre de type vortex situé à l'entrée de la cuve. Alimentation : 24 V cc reliée (via relais électrique 0-5 V cc → 0-24 V cc) **à la ligne 2 du port numérique 0** de la carte d'acquisition. Signal fourni : [0-10 V] **relié à l'entrée analogique 1** de la carte d'acquisition.
Débitmètre de type vortex	Domaine de valeur : [0,01-400 l/h] Marge d'erreur : 0,5 % Contraintes physiques : [– 40-85 °C] Correspondance volts – litres/heure donnée dans des fichiers d'étalonnage en fonction de la température
Pompe	Pompe de type tout ou rien fournissant une pression de 3 bars à 25 °C. Alimentation externe : 230 V~ **Reliée à la ligne 3 du port numérique 0** de la carte d'acquisition (via un relais électrique 0-5 V cc → 0-230 V~)
EVef	Électrovanne réglable située à l'entrée d'eau industrielle. Alimentation externe : 12 V~ **reliée à la ligne 4 du port numérique 0** de la carte d'acquisition via un relais électrique 0-5 V cc → 0-12 V~. Signal de commande : 0-10 V cc **relié à la sortie analogique 0** de la carte d'acquisition.

© Dunod – La photocopie non autorisée est un délit.

Tableau 7.5 (suite) – Description des éléments matériels choisis.

Éléments matériels	
	Correspondance litres/heure - volts donnée dans des fichiers d'étalonnage en fonction de la température
Alarme	Dispositif d'alarme sonore et lumineuse Alimentation externe : 24 V cc (éteint à 0 V) **reliée à la ligne 5 du port numérique 0** de la carte d'acquisition via un relais électrique 0-5 V cc → 0-24 V cc.

capable de scruter une température par seconde (pour les procédés thermiques, l'évolution de température est généralement lente) et de la fournir sur le port série sous forme d'une chaîne de caractères.

Le diagramme de contexte de la spécification adaptée est donné sur la figure 7.24.

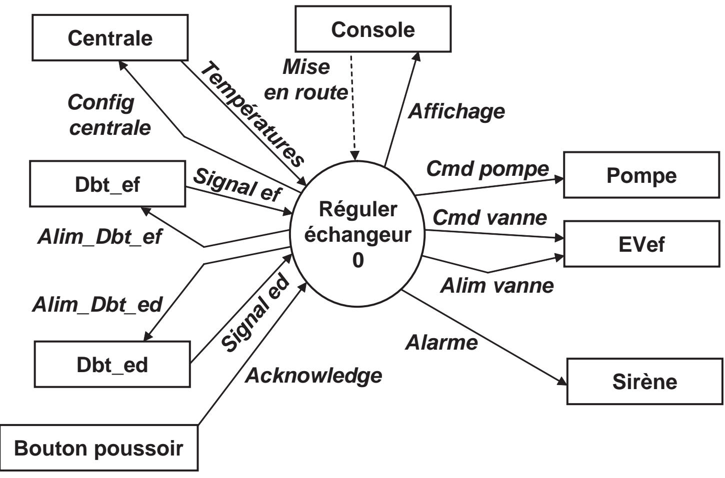


Figure 7.24 – Diagramme de contexte de la spécification adaptée.

Le diagramme préliminaire correspondant est donné sur la figure 7.25. Typiquement, les modifications portent sur la nécessité d'initialiser-configurer-alimenter-programmer certains éléments matériels. Généralement, des identificateurs d'élément configuré (port numérique, port série, etc.) doivent ensuite être utilisés. Dans la spécification, ces éléments sont regroupés dans un unique module de données nommé *Infos E/S*. Nous pouvons noter que le processus fonctionnel ajouté *Init E/S* est de type sous-programme, de même que *Lire bouton*.

Enfin, par rapport à la spécification donnée sur la figure 7.3, et ce afin d'éviter de trop charger le diagramme préliminaire, le processus *Vérifier débit ed* a été inséré dans le processus *Gérer Débits*, dont le diagramme flots de données est donné sur la

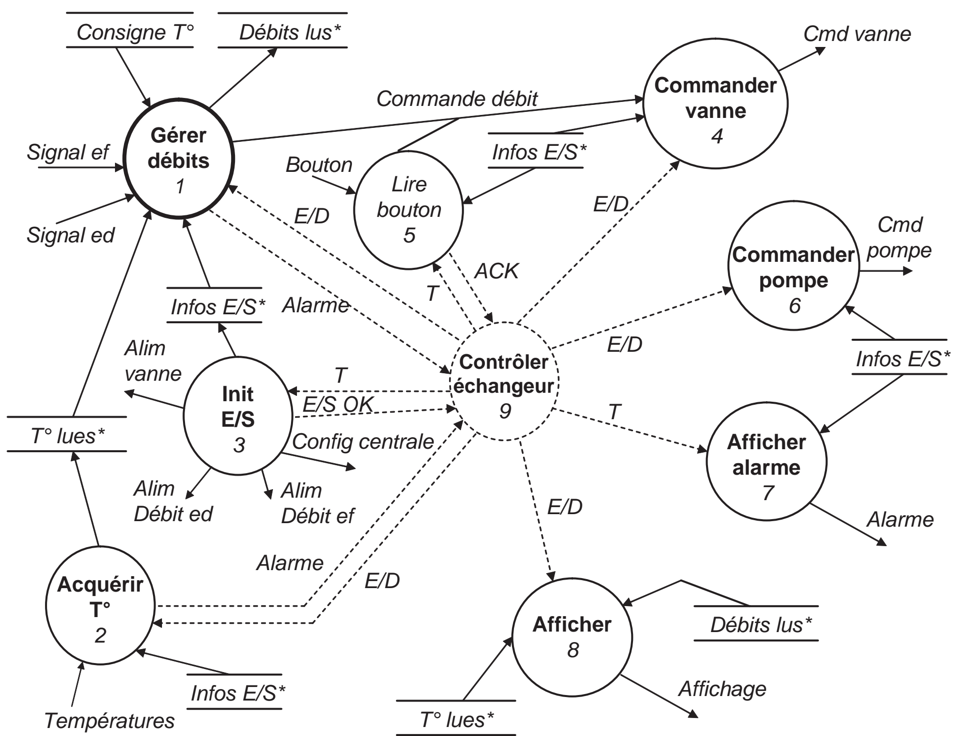


Figure 7.25 – Diagramme préliminaire de la spécification adaptée.

figure 7.26. Sur ce diagramme, nous pouvons noter l'apparition de deux nouveaux processus, chargés d'effectuer une conversion volts → litres/heure en fonction des tables d'étalonnage.

Observons maintenant le processus 4 : en pratique, il est impossible (sauf si la vanne intègre un débitmètre et qu'elle est capable de s'ajuster de façon autonome), d'obtenir à partir d'une commande seule un débit désiré. Il est nécessaire de mettre en place un algorithme d'asservissement, qui, étant donné un débit, modifie la consigne d'ouverture de l'électrovanne, la cible étant la consigne en litre/heures. C'est pour cette raison que le diagramme préliminaire donné sur la figure 7.25 doit être légèrement modifié, afin que *Commander Vanne* puisse asservir la vanne. De plus, lors d'un asservissement, des anomalies de fonctionnement peuvent être détectées (incohérence entre la commande et le débit lu, divergence). Il en résulte que *Commander Vanne* peut être à même de déclencher l'alarme. Enfin, il est fort possible que lors de l'initialisation du matériel, une erreur survienne, et nécessite l'arrêt du système. Le diagramme préliminaire finalement obtenu est donné sur la figure 7.27.

Le diagramme état/transition du processus de contrôle *Contrôler Echangeur* montre bien que l'utilisation du processus fonctionnel *Init E/S* est un appel de type sous-programme (figure 7.28).

Le dictionnaire de données, non reporté ici, contient quelques modifications triviales par rapport à celui de la spécification du système sur simulateur. Notons cependant

© Dunod – La photocopie non autorisée est un délit.

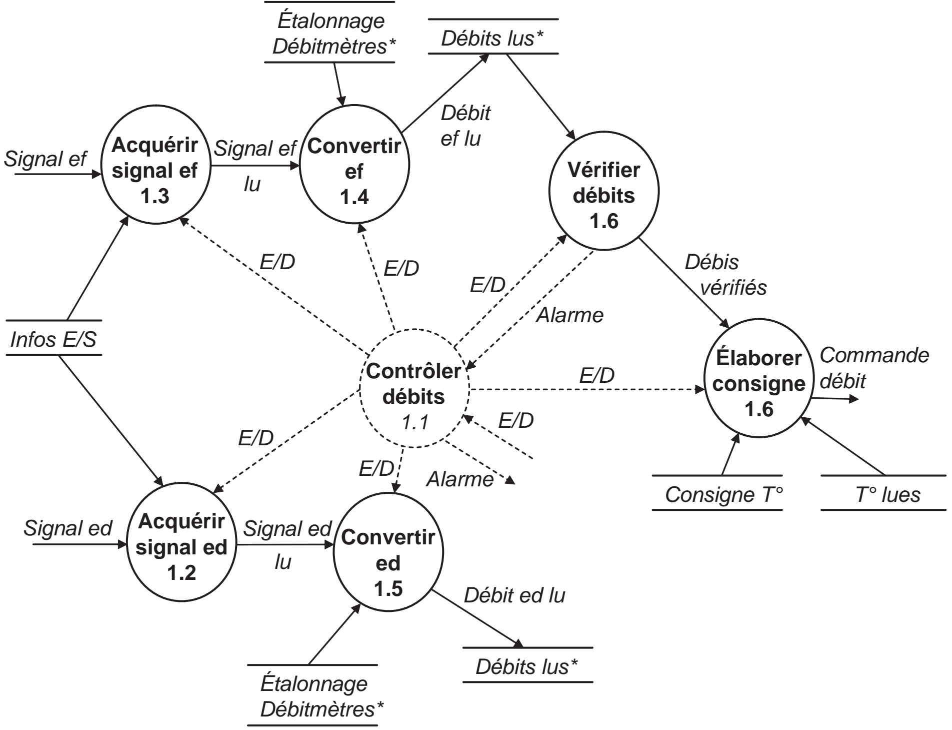


Figure 7.26 – DFD du processus Gérer Débits.

que les unités de mesure utilisées entre les processus fonctionnels du système sont des litres/heure pour les débits, et des degrés Celsius pour les températures : les conversions en volts sont réalisées lors de l'acquisition et de la commande.

Lors du passage à la conception, il existe deux possibilités en ce qui concerne le nouveau processus fonctionnel *Init E/S* :

– le processus est découpé, chaque partie d'initialisation étant alors placée dans la tâche qui utilise le matériel à initialiser ;

– le processus est implémenté dans le programme principal, avant le lancement des tâches.

Il faut tenir compte d'une contrainte technique liée à l'utilisation d'un port numérique : plusieurs tâches utilisent une ligne du même port, mais le port ne doit être initialisé qu'une seule fois avant tout autre accès. Pour le port numérique, seule la seconde solution peut être implémentée sans introduire d'éléments supplémentaires de synchronisation entre les tâches. En ce qui concerne le port série et les entrées ou sorties analogiques, n'importe laquelle des deux solutions peut être envisagée.

Afin de conserver une cohérence de l'initialisation, nous choisirons la seconde solution, qui présente l'avantage supplémentaire de demander peu ou pas de modification à l'intérieur des tâches par rapport à la programmation sur simulateur.

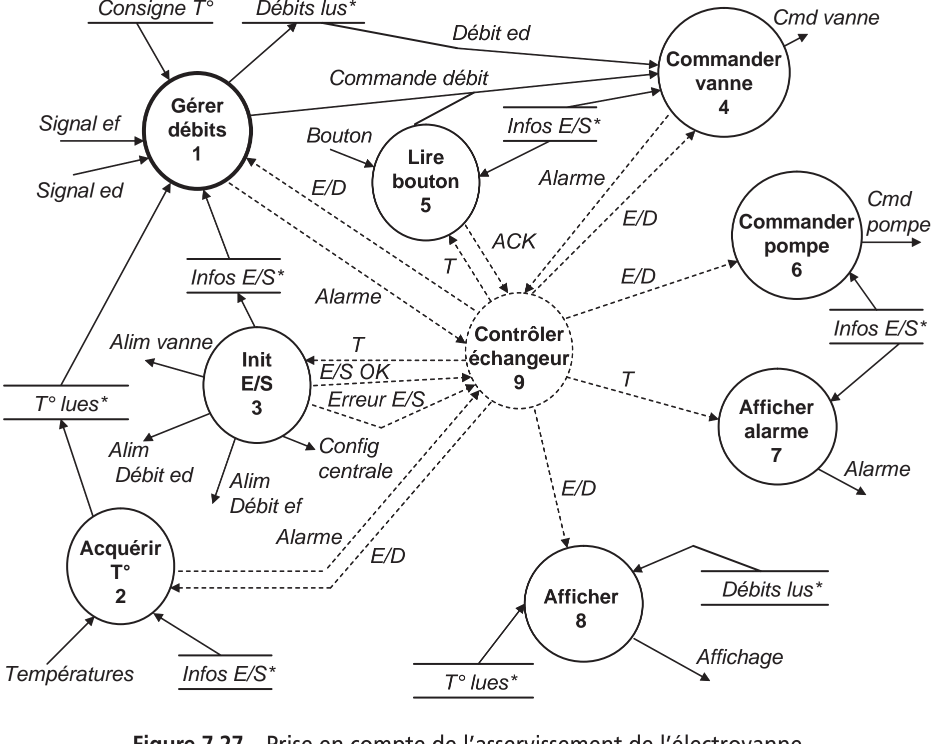


Figure 7.27 – Prise en compte de l'asservissement de l'électrovanne dans le diagramme préliminaire.

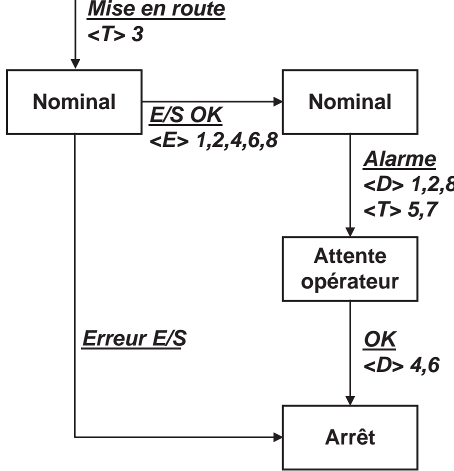


Figure 7.28 – Diagramme état/transition du processus de contrôle *Contrôler Echangeur.*

© Dunod – La photocopie non autorisée est un délit.

Le diagramme DARTS n'est donc presque pas modifié par rapport au diagramme DARTS présenté sur la figure 7.7. L'unique point est l'ajout d'une lecture du module de données *Débits* par la tâche *Commander Vanne*, qui pourrait s'appeler *Asservir Vanne*.

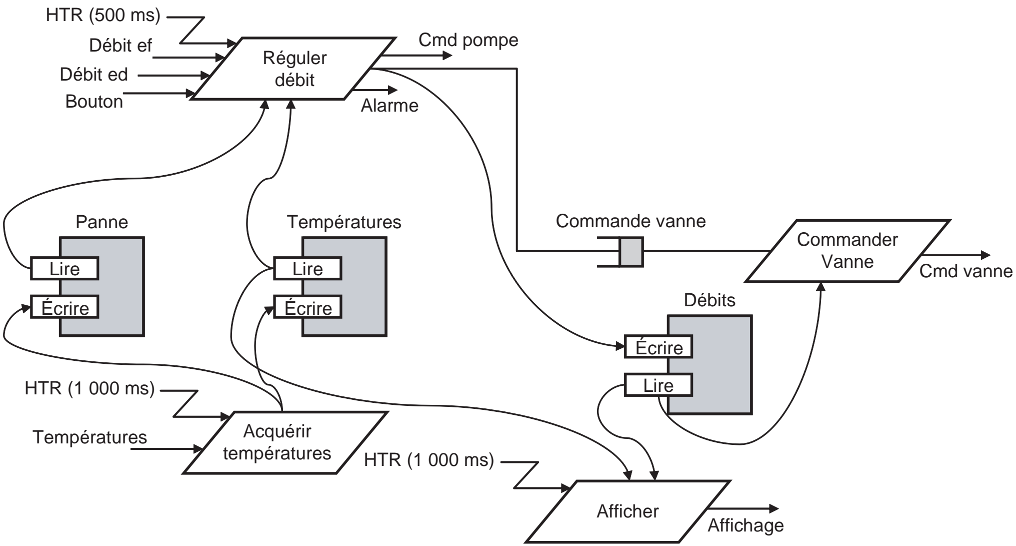


Figure 7.29 – Diagramme DARTS du contrôle de l'échangeur.

7.6 Implémentation de la commande réelle

L'implémentation va donc nécessiter l'implantation d'un algorithme d'asservissement dans la tâche *Commander Vanne*. Celui-ci pourra être par exemple un algorithme de type PID (Proportionnelle, Intégrale, Dérivée).

La prise en compte des alimentations nécessaires à certains capteurs et actionneurs, ainsi que la programmation de la centrale d'acquisition de températures, auront lieu avant le lancement des tâches.

7.6.1 Accès au matériel en C

La centrale d'acquisition de températures utilise un port série avec un protocole de type réseau de terrain MODBUS, et fournit une bibliothèque binaire sous la forme d'une libraire partagée MS Windows® (extension .dll) accompagnée d'un fichier d'en-tête des fonctions permettant de communiquer avec la centrale. Une documentation décrit le protocole de communication employé (configuration, lecture, etc.) pour communiquer avec la centrale. Voici un résumé commenté du fichier d'en-tête fourni :

```
/*
ComCentrale.h
Fichier d'en-tête de libraire de communication avec la centrale
*/
/* Différentes constantes de débit du port série */
#define BAUDS_600   0
#define BAUDS_1200  1
#define BAUDS_2400  2
#define BAUDS_4800  3
#define BAUDS_9600  4
#define BAUDS_19200 5
/* Différentes constantes de noms de port série */
#define COM1 0
#define COM2 1
#define COM3 2
#define COM4 3

int OpenModbusComm(const int port, const unsigned char bauds_rate);
/* Ouvre la communication MODBUS, à appeler avant toute autre fonction
ENTREES:
port: nom du port série utilisé (voir constantes de port série)
bauds_rate: débit du port série (voir constantes de débit)
Renvoie: 0 si pas d'erreur, code d'erreur différent de 0 sinon */

int CloseModbusComm(const int port);
/* Met fin à la communication MODBUS
ENTREES:
port: nom du port série utilisé (voir constantes de port série)
Renvoie: 0 si pas d'erreur, code d'erreur différent de 0 sinon */

int Run(const int port, const unsigned char address);
/* Lance l'acquisition sur la centrale
Nécessite: MODBUS ouvert sur le port
ENTREES:
port: nom du port série utilisé (voir constantes de port série)
address: adresse MODBUS de la centrale entre 1 et 127
Renvoie: 0 si pas d'erreur, code d'erreur différent de 0 sinon */
int Stop(const int port, const unsigned char address);
/* Arrête la centrale
Nécessite: MODBUS ouvert sur le port
ENTREES:
port: nom du port série utilisé (voir constantes de port série)
address: adresse MODBUS de la centrale entre 1 et 127
Renvoie: 0 si pas d'erreur, code d'erreur différent de 0 sinon */

int CommandKeyword(const int port, const unsigned char address, const
char * request, char *answer);
/* Envoie un mot de configuration à la centrale (voir protocole)
Nécessite: MODBUS ouvert sur le port
           Centrale non lancée
ENTREES:
port: nom du port série utilisé (voir constantes de port série)
address: adresse MODBUS de la centrale entre 1 et 127
request: chaîne de caractère de configuration (voir protocole)
SORTIES:
answer: réponse reçue
Renvoie: 0 si pas d'erreur, code d'erreur différent de 0 sinon */

int ReadChannels(const int port, const unsigned char address, const
unsigned char firstChannel,
```

© Dunod – La photocopie non autorisée est un délit.

```
 const unsigned char numberOfChannels, float *channelsValues);
/* Effectue une lecture sur des voies (chaque voie est reliée à un
thermocouple)
ENTREES:
Nécessite: MODBUS ouvert sur le port
           Centrale lancée
           la taille de channelsValues est >= à numberOfChannels
port: nom du port série utilisé (voir constantes de port série)
address: adresse MODBUS de la centrale entre 1 et 127
firstChannel: numéro de la première voie à lire
numberOfChannels: nombre de voies à lire à partir de la première
SORTIES:
channelsValues: tableau contenant les températures lues dans l'ordre
des voies
Renvoie: 0 si pas d'erreur, code d'erreur différent de 0 sinon */
```

Si le système d'exploitation utilisé est de type MS Windows®, l'utilisation des fonctionnalités de la centrale est immédiate. Si ce n'est pas le cas, il reste à chercher un portage de cette libraire sur le système utilisé, et, si cela n'existe pas, l'écrire soi-même. Ce processus est généralement très long et laborieux.
Après étude du protocole de communication de la centrale, nous pouvons établir une abstraction de la libraire fournie en trois fonctions :

– *Initialiser_Centrale*, qui ouvre une communication MODBUS (fonction *OpenModbusCom*), configure la centrale en lui envoyant des informations sur les voies à acquérir (appels à *CommandKeywords*), puis lance l'acquisition (appel à *Run*) ;
– *Lire_Centrale*, qui utilise *ReadChannels*, lecture bloquante sur le port série, et renvoie le tableau de nombres flottants reçus ;
– *Fermer_Centrale*, qui arrête la centrale (*Stop*) et ferme la communication ModBus (*CloseModbuscom*).

La spécification de ces fonctions est :

```
/* MaComCentrale.h
    Abstraction des accès à la centrale d'acquisition de températures
*/
#ifndef _MACOMCENTRALE_H_
#define _MACOMCENTRALE_H_
#include "ComCentrale.h"
#include "types_communs.h"
/* Constantes utilisées pour communiquer avec la centrale */
#define PORT_CENTRALE    COM1
#define ADRESSE_CENTRALE 1
#define BAUDS_CENTRALE   BAUDS_9600

int Initialiser_Centrale();
/* Initialise la centrale
Renvoie: 0 si tout s'est bien passé, code d'erreur sinon */
int Lire_Centrale(Temperatures_t *T);
/* Lit les températures (lecture bloquante sur port série)
Nécessite: centrale initialisée
Entraine: T contient les températures lues
Renvoie: 0 si tout s'est bien passé, code d'erreur sinon */
int Fermer_Centrale();
/* Termine la connexion avec la centrale
Nécessite: centrale initialisée
Entraine: la centrale n'est plus initialisée
```

```
Renvoie: 0 si tout s'est bien passé, code d'erreur sinon */
#endif
```

La carte d'acquisition, de marque National Instruments® pour le cas étudié, est fournie avec deux pilotes de périphérique unifiés pour toute carte fournie par ce constructeur. Ces *drivers*, existant sur plusieurs plates-formes, fournissent bien entendu des librairies binaires, associées à des fichiers en-tête (.h) de spécification des fonctions implémentées dans les librairies. La démarche est identique à celle utilisée pour la centrale. Pour plus de simplicité, nous choisirons la librairie traditionnelle, bien que la nouvelle librairie *mx* offre des performances bien meilleures.
Voici la spécification du module d'abstraction orienté procédé de la carte d'acquisition. On peut remarquer la présence de nombreuses constantes qui permettront, le cas échéant, de modifier certains paramètres matériels :

```
/* mondaq.h
Abstraction des accès à la carte d'acquisition pour l'échangeur */
#ifndef _MONDAQ_H_
#define _MONDAQ_H_

#define CARTE_DAQ_ID      1 /* Numéro de périphérique associé à la
carte */
#define PORT_CARTE_DAQ    0 /* Port d'E/S lié aux différents éléments
numériques */

#define LIGNE_BOUTON      0 /*Ligne liée au bouton */
#define LIGNE_ALIM_DBT_EF 1 /*Ligne liée à l'alimentation de Dbt_ef*/
#define LIGNE_ALIM_DBT_ED 2 /*Ligne liée à l'alimentation de Dbt_ed*/
#define LIGNE_POMPE       3 /*Ligne liée à la pompe */
#define LIGNE_ALIM_EV_EF  4 /*Ligne liée à l'alimentation de
l'électrovanne */
#define LIGNE_ALARME      5 /*Ligne liée à l'alarme */

#define PORT_DBT_EF      0 /*Port d'entrée analogique relié à Dbt_ef
*/
#define PORT_DBT_ED      1 /*Port d'entrée analogique relié à Dbt_ed
*/
#define PORT_EV_EF       0 /*Port de sortie analogique relié à EV_ef
*/

int Initialiser_DAQ();
/* Initialise les E/S numériques sur la carte
Renvoie: 0 si tout s'est bien passé, code d'erreur sinon */
float Lire_Debit_Ef_DAQ();
/* Renvoie le débit d'eau industrielle en l/h
  Nécessite: procédé initialisé préalablement*/
float Lire_Debit_Ed_DAQ();
/* Renvoie le débit d'eau distillée en l/h
   Nécessite: procédé initialisé préalablement */
BYTE Lire_Bouton_DAQ();
/* Renvoie l'état du bouton poussoir
   Retourne: vrai si appuyé, faux sinon
   Nécessite: procédé initialisé préalablement */
void Commander_Pompe_DAQ (BYTE Com );
/* Allume ou éteint la pompe
   Entraine: Com = 1 => allume la pompe
             Com = 0 => éteint la pompe
   Nécessite: procédé initialisé préalablement */
void Commander_Vanne_DAQ (float Com );
```

© Dunod – La photocopie non autorisée est un délit.

```
/* Commande l'électrovanne en l/h
   Nécessite: procédé initialisé préalablement */
void Commander_Alarme_DAQ(BYTE On);
/* Allume ou éteint l'alarme
   Entraine: On = 1 => allume l'alarme
             On = 0 => éteint l'alarme
   Nécessite: procédé initialisé préalablement */

#endif
```

L'implémentation de ce module se réalise très simplement à l'aide de la libraire fournie (voir ci-après). Notons que nous ne détaillons pas les fonctions de conversion d'unités (volts – litres/heure) qui se basent sur des tableaux de points et se contentent de faire des interpolations.

```
/* mondaq.c */
#include <nidaq.h>  /* Librairie NI-DAQ */
#include "mondaq.h"

int Initialiser_DAQ() {
    int status;
    /* Initialisation du port numérique en écriture */
    if ((status=DIG_Prt_Config(CARTE_DAQ_ID, PORT_CARTE_DAQ,0,1))!=0)
return status;
    /* La ligne correspondant au bouton est mise en lecture */
    if ((status=DIG_Line_Config(CARTE_DAQ_ID,
PORT_CARTE_DAQ,LIGNE_BOUTON,0))!=0) return status;
    /* Les débitmètres et l'électrovanne sont alimentés */
    if ((status=DIG_Out_Line(CARTE_DAQ_ID, PORT_CARTE_DAQ,
LIGNE_ALIM_DBT_EF,1))!=0) return status;
    if ((status=DIG_Out_Line(CARTE_DAQ_ID, PORT_CARTE_DAQ,
LIGNE_ALIM_DBT_ED,1))!=0) return status;
    if ((status=DIG_Out_Line(CARTE_DAQ_ID, PORT_CARTE_DAQ,
LIGNE_ALIM_EV_EF,1))!=0) return status;
    return 0;
}
BYTE Lire_Bouton_DAQ() {
    i16 etat;
    DIG_In_Line(CARTE_DAQ_ID, PORT_CARTE_DAQ, LIGNE_BOUTON,&etat);
    return etat;
}
void Commander_Pompe_DAQ (BYTE Com ) {
    DIG_Out_Line(CARTE_DAQ_ID, PORT_CARTE_DAQ, LIGNE_POMPE, Com);
}
void Commander_Alarme_DAQ(BYTE On) {
    DIG_Out_Line(CARTE_DAQ_ID, PORT_CARTE_DAQ, LIGNE_ALARME, On);
}
float Lire_Debit_Ef_DAQ() {
    f64 val;
    AI_VRead(CARTE_DAQ_ID, PORT_DBT_EF, 1, &val);
    return Conversion_volt_l_h(val);
}
float Lire_Debit_Ed_DAQ() {
    f64 val;
    AI_VRead(CARTE_DAQ_ID, PORT_DBT_ED, 1, &val);
    return Conversion_volt_l_h(val);
}
void Commander_Vanne_DAQ (float Com ) {
    AO_VWrite(CARTE_DAQ_ID, PORT_EV_EF, Conversion_l_h_volt(Com));
}
```

Il ne reste plus alors qu'à implémenter le corps du module *procede*, ce qui est trivial étant donné que tous les accès au matériel sont faits de façon haut niveau.

7.6.2 Accès au matériel en Ada

Le langage Ada a des qualités indéniables, notamment pour la programmation temps réel, mais il ne permet pas simplement d'exécuter du code avant le lancement des tâches du système si celles-ci sont déclarées dans un paquetage. Il existe bien évidemment différents moyens d'y remédier :

– Le paquetage *Controler_Echangeur*, contenant les différents éléments de communication et les tâches peut ne définir que les types des tâches, leur instanciation étant faite dynamiquement dans un bloc *declare* du programme principal. Cependant, cette technique n'est pas conforme au profil *Ravenscar*.
– Un objet protégé est utilisé afin de synchroniser le départ des tâches, qui n'aura alors lieu qu'après l'initialisation du système. C'est la solution que nous suivrons.

L'implémentation est alors modifiée comme suit : un objet protégé *Starter* est créé dans le paquetage *Controler_Echangeur*, c'est lui qui servira de déclencheur aux tâches : Sa spécification est :

```
protected Starter is
   procedure Go;
   -- Débloque toutes les tâches bloquées sur Attendre_Départ
   entry Attendre_Départ;
   -- Entrée bloquante jusqu'à un appel de Go
private
   Demarre: Boolean := False;
end;
Et son corps est le suivant:
protected body Starter is
   procedure Go is
   begin
      Demarre:=True;
   end;
   entry Attendre_Départ when Demarre is
   begin
      null;
   end;
end;
```

Chaque tâche n'étant pas en attente sur une synchronisation ou bien une boîte aux lettres (comme *Commander_Vanne*) se voit ajouter au début de son code un appel à *Starter.Attendre_Départ*. Ainsi, toutes les tâches se retrouvent en attente jusqu'à un appel à *Starter.Go*. Le programme principal devient donc :

```
pragma Queuing_Policy(Priority_Queuing);
-- L'accès aux objets protégés est géré par niveau de priorité
pragma Locking_Policy(Ceiling_Locking );
-- Utilisation du protocole à priorité plafond
pragma Task_Dispatching_Policy(Fifo_Within_Priorities);
-- Ordonnancement des tâches à priorités
with Controler_Echangeur;
with Procede;use Procede;
procedure Echangeur is
begin
```

© Dunod – La photocopie non autorisée est un délit.

```
   Initialiser;
   Starter.Go
end;
```

Nous pouvons constater que les clauses d'utilisation du simulateur ont été remplacées par les clauses d'utilisation du procédé. Il n'y a plus alors qu'à implémenter les accès au matériel dans le corps du paquetage *Procédé*.
La plupart des drivers de matériel (dans notre cas, par exemple, le driver de la carte d'acquisition et le driver de la centrale d'acquisition de température) sont fournis en C, sous forme par exemple d'une librairie au format binaire, accompagnée d'un ou plusieurs fichiers d'en-tête C (fichiers d'extension .h).
Il en résulte l'obligation, soit de trouver un *binding* (interfaçage entre Ada et une bibliothèque C) existant, comme on en trouve beaucoup sur *internet*, soit d'écrire soi-même ce *binding*. Dans le cas présent, nous n'avons trouvé aucun *binding* pour Ada. Le plus simple est alors d'adopter la même démarche que dans le paragraphe précédent, c'est-à-dire d'écrire en C des modules d'accès de haut niveau à la centrale, et à la carte d'acquisition. À partir de là, il reste à interfacer un paquetage Ada avec les fonctions C définies dans *mondaq.h* et *MaComCentrale.h* (voir section précédente).
Ainsi, par exemple, la fonction C *Lire_Debit_Ef_DAQ* serait interfacée de la façon suivante :

```
function Lire_Debit_Ef_Daq return Float;
pragma Import(C,Lire_Debit_Ef_Daq);
```

Le même principe appliqué à toutes les fonctions d'accès au procédé montre que, même si l'on a choisi le langage Ada, dans la plupart des cas, l'accès au matériel nécessite une partie de programmation en C, à moins que l'on ait le courage d'implémenter un *driver* spécifique en Ada.

7.6.3 Accès au matériel en LabVIEW

Quel que soit le matériel utilisé, il y a trois possibilités :

– le constructeur fournit une bibliothèque d'exploitation LabVIEW, dans ce cas, l'abstraction de la bibliothèque est triviale ;
– le constructeur fournit une librairie binaire, ou une libraire partagée (.so sous Linux, .dll sous MS Windows®…) ou bien des sources dans un langage, typiquement C. Dans ce cas, on pourra utiliser les *Codes Interface Nodes*, ou bien l'appel direct de fonctions de librairie partagée.

Dans le cas présent, le constructeur de la centrale d'acquisition fournit une bibliothèque d'exploitation LabVIEW. Ainsi, l'initialisation de la centrale est donnée sur la figure 7.30. Noter l'utilisation d'une variable globale stockant les paramètres (identificateur) de la centrale, qui seront utilisés ultérieurement pour la lecture de températures (figure 7.31). L'utilisation de variables globales à la place d'un vi non réentrant de type module de données n'est à conseiller pour sa simplicité que lorsque la variable n'est modifiée qu'au début du programme, et utilisée en lecture uniquement dans la suite.
Le langage LabVIEW fournit une bibliothèque unifiée d'accès aux cartes d'acquisition, basée sur deux drivers de carte NI-DAQ. Notons que ces drivers, NI-DAQ

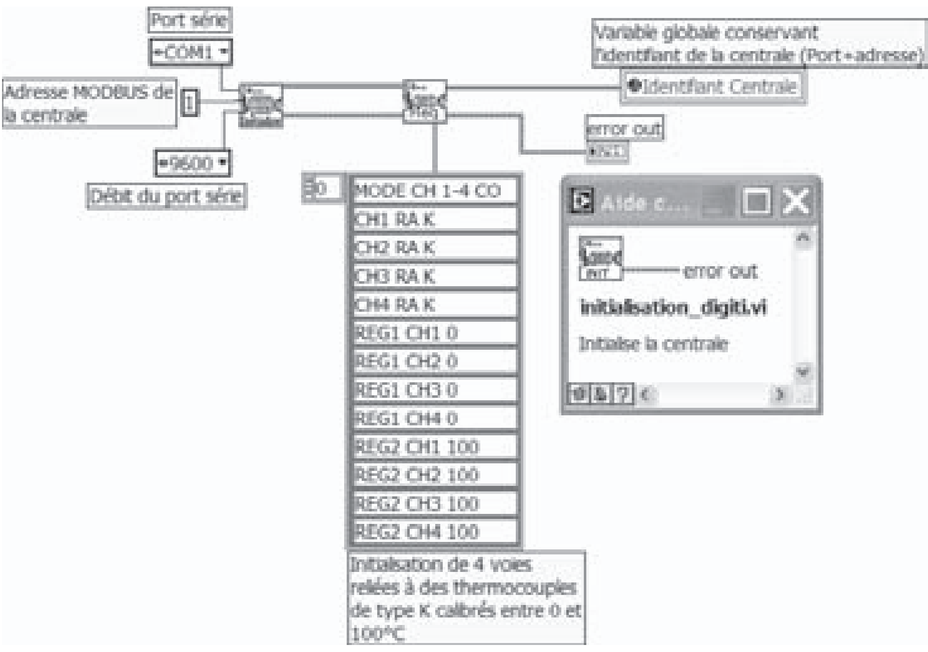


Figure 7.30 – Initialisation de la centrale en LabVIEW.

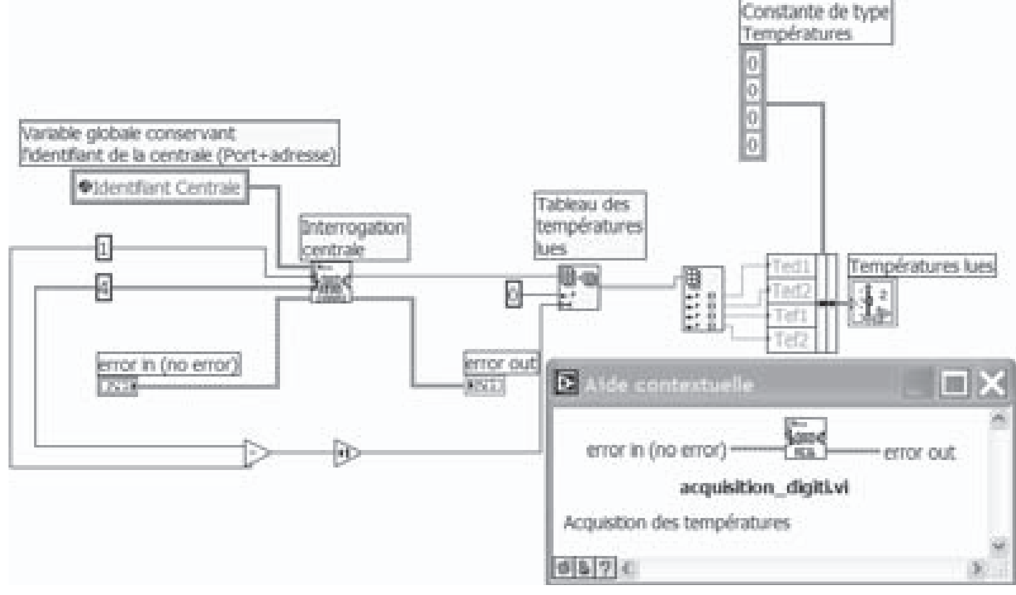


Figure 7.31 – Lecture des températures sur la centrale.

© Dunod – La photocopie non autorisée est un délit.

traditionnal et NI-DAQ *mx* (pour *multithread*) sont fondamentalement différents dans leur philosophie, et que la version *mx* offre des améliorations sensibles au niveau du temps d'accès aux périphériques, cependant, certaines cartes pour bus ISA, jugées obsolètes, ne sont pas supportées par ce driver.

La philosophie employée pour les ports numériques est de configurer une fois un port (notamment en configurant la direction des lignes), ce qui fournit un identificateur de port configuré, puis de lire ou écrire sur ce port autant que nécessaire. Les ports analogiques, quant à eux, ne se configurent pas dans un sens ou dans l'autre (voir § 4.1.4), et ne nécessitent pas de configuration préalable. La figure 7.32 montre l'initialisation du port numérique (voir l'affectation des lignes tableau 7.5). Nous utilisons un *vi* non réentrant, ce qui garantit l'exclusion mutuelle de toutes les actions ayant lieu sur le port numérique. La figure 7.33 donne les cas des 3 autres actions possibles sur le port numérique.

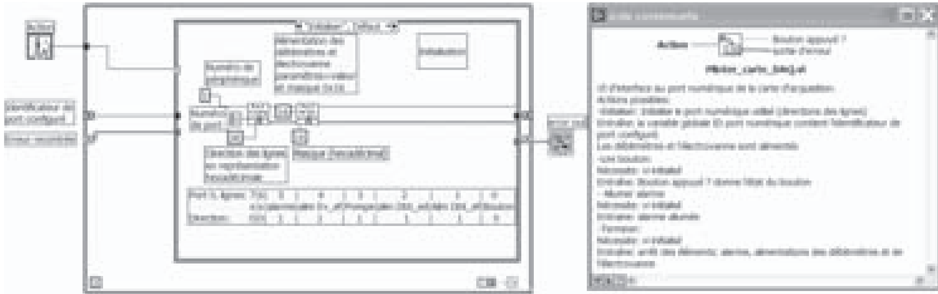
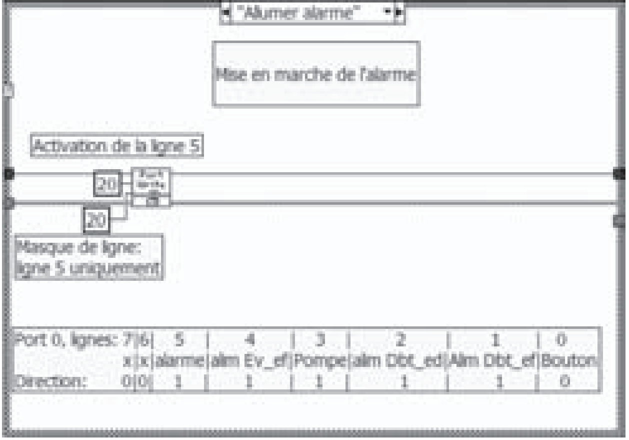
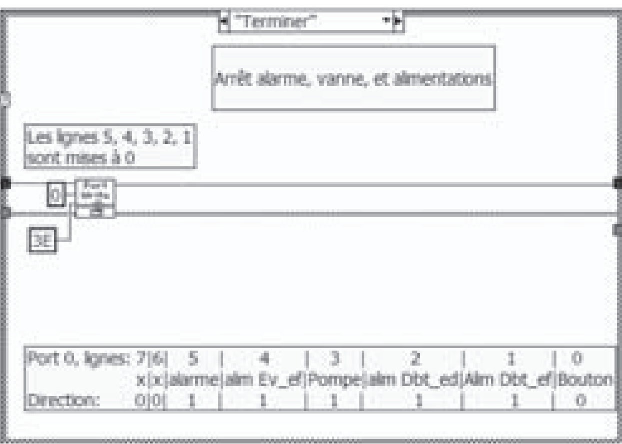


Figure 7.32 – Initialisation du port numérique et alimentation des débitmètres et électrovannes.

Les ports analogiques sont encore plus simples à commander ou lire, de plus, la mise à l'échelle est simplifiée grâce à l'utilitaire *Measurement and Automation eXplorer* (figure 7.34). Ce logiciel permet de créer très simplement un vi de lecture en litre/heure, comme le montre la figure 7.35 sur l'exemple de la lecture de débit d'eau distillée.

Après l'implémentation, généralement relativement simple, des *vi* d'accès au matériel, il ne reste plus alors qu'à remplacer les appels au simulateur contenus dans les tâches et le programme principal (figure 7.36) par des appels aux vis nouvellement créés.





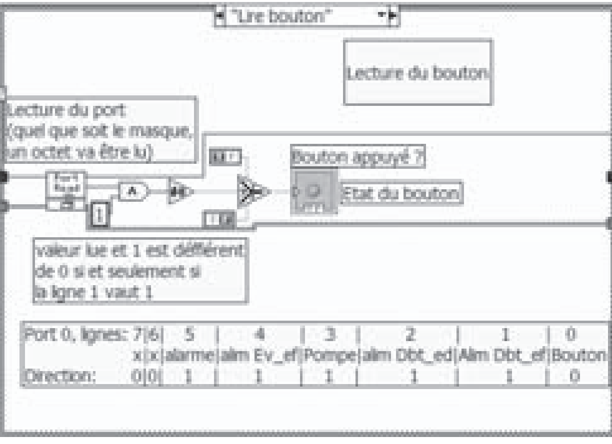


Figure 7.33 – Actions possibles sur le port numérique.

© Dunod – La photocopie non autorisée est un délit.

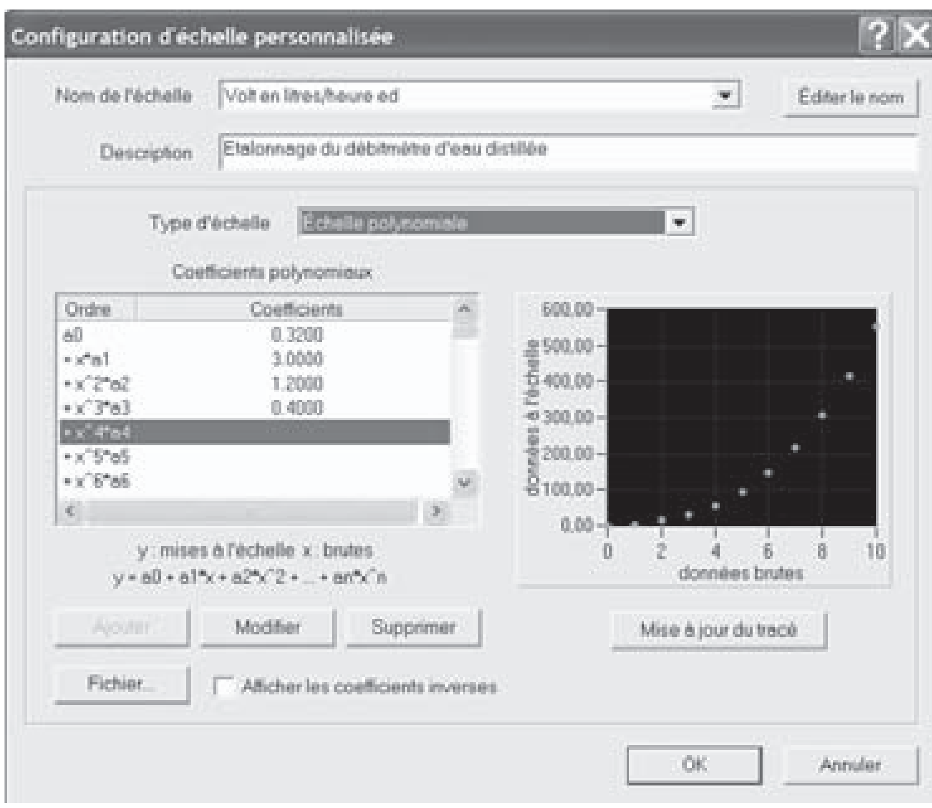


Figure 7.34 – Création d'une échelle liée à un étalonnage.

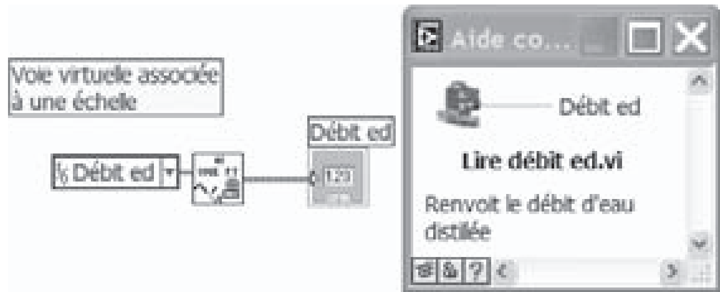


Figure 7.35 – Utilisation d'une voie virtuelle utilisant une échelle personnalisée.

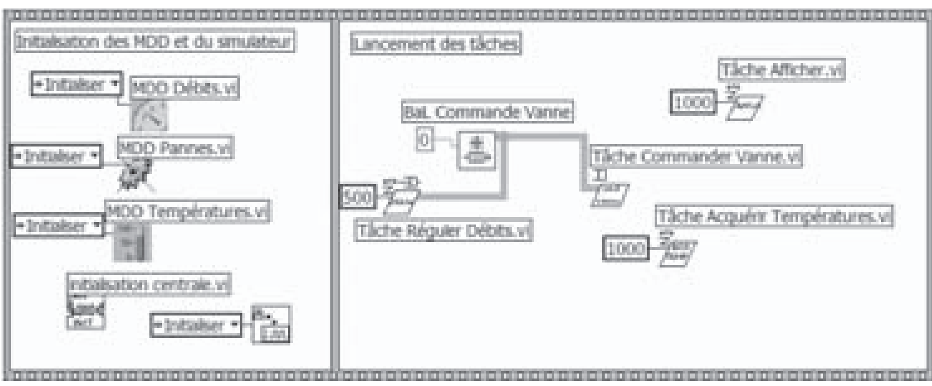


Figure 7.36 – Le programme principal du contrôle de l'échangeur en LabVIEW.

7.7 Conclusion

À travers cet exemple, nous pouvons constater que chaque langage a des avantages et des faiblesses spécifiques.

Le langage C présente l'avantage d'être le langage de base dans lequel on peut trouver tout *driver*. Cependant, son inconvénient majeur est son typage faible (c'est souvent au programmeur que revient l'effort de vérifier la cohérence des types), et sa relative difficulté de programmation, ainsi que la multitude d'implémentations partielles des normes POSIX.

Le langage Ada présente l'élégance inhérente au multitâche natif, et l'avantage du typage fort (le déverminage du programme écrit en C a pris aux auteurs environ 10 fois plus de temps que celui du programme écrit en Ada). Il présente cependant l'inconvénient de ne disposer d'aucun *driver*, et il échoit souvent au programmeur d'avoir à écrire un *binding* entre un *driver* fourni en langage C et Ada, le tout nécessitant une partie de programmation en C.

Le langage LabVIEW présente l'avantage d'offrir un multitâche implicite et naturel. Les *drivers* d'éléments matériels se trouvent relativement facilement (et même en leur absence, il est très simple d'effectuer un *binding* à l'aide d'une libraire partagée). Son inconvénient est que, pour le moment, un seul système d'exploitation temps réel, aux fonctionnalités relativement restreintes comparées à un exécutif temps réel pour C ou Ada, est disponible. Cela rend le langage particulièrement bien adapté aux programmes de contrôle de procédé sans contraintes de temps strictes (comme c'est le cas sur l'exemple de l'échangeur).

© Dunod – La photocopie non autorisée est un délit.

8 • ÉTUDE AVANCÉE DES SYSTÈMES TEMPS RÉEL

8.1 Introduction

8.1.1 Présentation générale de l'ordonnancement

Les méthodes d'ordonnancement, qui ont été décrites dans le chapitre 4, sont indépendantes des caractéristiques temporelles intrinsèques des tâches. En effet, l'affectation des priorités à un ensemble de tâches se fait de façon non formelle.

Les algorithmes basés sur les priorités peuvent être à priorités fixes ou variables. Dans le cas où les priorités sont variables, l'ordonnanceur met à jour les priorités à chaque réveil des tâches ou aux instants des appels de primitives, ou à chaque top d'horloge. Il s'appuie ensuite sur le répartiteur (*dispatcher*) qui choisit la tâche prête la plus prioritaire et lui octroie le processeur. On peut noter qu'à ce jour, tous les exécutifs du commerce se basent sur des priorités fixes. Cependant, il convient de nuancer l'aspect statique des priorités par l'utilisation de protocoles de gestion de ressources : dans ce cas, la priorité des tâches, bien que fixe au début, peut évoluer au cours de la vie de l'application.

Les algorithmes à priorités sont appelés algorithmes d'**ordonnancement en ligne** car ils se basent sur l'état instantané du système pour prendre une décision. Les algorithmes présentés dans le chapitre 4 appartiennent à cette catégorie. Les ordonnancements basés sur des séquences préétablies s'appellent des **algorithmes hors-ligne**. Dans ce cas, la séquence d'ordonnancement est construite à partir d'une vision complète du système, puis exécutée par un **séquenceur**.

Lorsque le système contrôlé impose des contraintes strictes de temps de réponse, se traduisant sous la forme d'échéances temporelles à respecter, les mécanismes d'ordonnancement par priorité affectée par le concepteur, ou par temps partagé ne sont pas toujours les plus intéressants et les plus efficaces pour séquencer l'exécution des tâches. Si on possède suffisamment d'informations concernant les tâches de l'application, il est possible d'obtenir une validation pour laquelle toutes les tâches s'exécutent en respectant leurs contraintes temporelles.

Le but recherché dans cette étude est de pouvoir valider une application multitâche de façon plus rigoureuse ou plus formelle que de soumettre cette application à un ensemble de tests qui ne seront jamais exhaustifs. Ce complément d'analyse formelle peut ainsi conduire à diminuer fortement les tests et à qualifier l'application pour un haut niveau de sûreté de fonctionnement. L'intégration dans ce cadre d'étude

© Dunod – La photocopie non autorisée est un délit.

d'une application multitâche quelconque nécessite de prendre en compte toutes les contraintes associées aux tâches : primitives de synchronisations et de communications, relation de précédence, partage de ressources, contraintes temporels (périodicité, échéance…), etc. Les modèles permettant la prise en compte de ces différents éléments sont présentés au fur et à mesure de ce chapitre.
Afin de simplifier la modélisation des tâches, mais sans modifier la généralité de la présentation, nous allons introduire la notion de tâche atomique ou normale, notion déjà abordée dans le chapitre 5 au niveau du noyau temps réel OSEK-VDX. Ainsi, nous avons la définition suivante :

> « Une **tâche atomique ou normale** est une entité d'exécution qui ne possède aucun appel à une primitive bloquante dans son code. »

Cette définition est précisée ultérieurement lors de la modélisation des relations de précédences entre les tâches. Ainsi, excepté dans des cas explicités, à chaque fois que le mot « tâche » est employé dans ce chapitre, il fait toujours référence à une tâche atomique.
L'analyse théorique de l'ordonnancement dans ce cadre applicatif complet (tâches dépendantes, contexte préemptif, ordonnancement en ligne) est d'une très grande complexité. Aussi l'ordonnnancement en ligne fait souvent appel à la mise en place d'heuristiques dont les propriétés sont à préciser : contexte d'applicabilité, optimalité, test d'ordonnançabilité associé, etc.
Les différents algorithmes d'ordonnancement ou heuristiques d'ordonnancement que nous allons étudier sont basés sur les priorités. Mais, à la différence d'une affectation de cette priorité des tâches selon les spécifications de l'application couplées aux remarques des concepteurs, les priorités des tâches vont être affectées en fonction des paramètres temporels des tâches. Les priorités pourront même changer au cours de l'exécution de l'application si elles doivent suivre des caractéristiques temporelles des tâches qui évoluent elles-mêmes en fonction de l'exécution.
Après avoir présenté les différents modèles des tâches et avoir fait une première analyse d'une configuration de tâches, nous allons étudier diverses façons de produire une séquence d'exécution des tâches en respectant les contraintes temporelles des tâches pour différentes configurations. La construction de la séquence se fait en suivant la règle de choix d'une tâche à exécuter selon un algorithme d'ordonnancement défini, équivalent à suivre les priorités affectées aux tâches.

8.1.2 Algorithmes d'ordonnancement

Un algorithme d'ordonnancement étant défini comme un algorithme capable de donner une description (séquence) du travail à effectuer par le ou les processeurs, une séquence est dite **valide** si les échéances des tâches sont respectées.
Un algorithme est dit **fiable** pour une configuration de tâches s'il produit une séquence valide sur une durée infinie quelles que soient les valeurs des premières dates de déclenchement des différentes tâches. Une configuration est dite **ordonnançable** s'il existe au moins un algorithme fiable.
Dans un contexte de tâches et d'algorithmes d'ordonnancement (affectation de priorités), nous allons qualifier l'algorithme d'ordonnancement étudié selon deux aspects :

– **optimalité** : si la configuration de tâches est ordonnançable dans cette catégorie d'algorithmes, alors elle le sera avec l'algorithme étudié ;
– **ordonnançabilité** : la capacité à pouvoir prévoir l'ordonnancement de la configuration de tâches en se basant sur des conditions nécessaires et/ou suffisantes ou des simulations de l'exécution.

Il est important de faire une remarque concernant le dernier point évoqué, c'est-à-dire une simulation de l'exécution consistant à construire tout ou partie de la séquence d'exécution et d'affirmer l'ordonnançabilité de l'application sur cette analyse. En effet, la construction de la séquence d'exécution en simulant l'enchaînement des tâches allouées au processeur selon l'algorithme d'ordonnancement suppose une connaissance *a priori* des caractéristiques temporelles des tâches et du déterminisme de ces paramètres. En particulier comme nous l'analyserons dans ce chapitre, la variation de la durée d'exécution du code de la tâche, due à l'algorithmique du code ou à l'environnement d'exécution, peut conduire à de grandes variations dans l'exécution des tâches et même à des anomalies : une diminution de la durée d'exécution d'une tâche peut rendre une configuration de tâches non ordonnançable.

8.1.3 Temps discret

Dans la suite de ce chapitre, nous allons utiliser un temps discret, c'est-à-dire que le temps est considéré comme une valeur entière qui évolue par incrément de 1. De même, les paramètres des tâches vont être affichés sous forme de valeurs entières sans unité temporelle précisée. Dans les applications industrielles, ces grandeurs sont exprimées en fraction de milliseconde, voire en microseconde et la précision dépend de l'horloge du processeur. Pour faire la traduction de l'application réelle (grandeurs réelles) vers l'application formalisée (grandeurs entières), nous pouvons soit procéder à un arrondi au plus proche entier, soit considérer un quantum équivalent au temps unité de base, par exemple 25 µs correspond à une unité de temps dans l'analyse formelle de l'application. Ce temps est aussi fonction de la granularité temporelle du noyau temps réel. De plus les temps affichés dans une application sont en général considérés comme des temps relatifs au lancement de l'application.

8.2 Modélisation des tâches

Dans un système temps réel, les **contraintes de temps** découlent de la dynamique du procédé contrôlé. Il existe différents types de contraintes de temps, qui seront affinés dans le chapitre portant sur l'ordonnancement :

– **contraintes de bout en bout** : le procédé doit effectuer une ou des réactions sur le procédé (typiquement via des actionneurs) en un temps contraint. Typiquement, ces contraintes influent sur les contraintes de tâches faisant partie d'une chaîne, de l'acquisition à la commande ;
– **contraintes de non réentrance** : dans un système temps réel, les tâches effectuent un traitement cyclique (*i.e.* soit périodique, soit répété à chaque événement attendu). Chaque itération de cette « boucle » s'appelle une instance de tâche. Une tâche ne pouvant pas être réentrante, il faut typiquement s'assurer que chaque

© Dunod – La photocopie non autorisée est un délit.

instance d'une tâche puisse se terminer avant le prochain événement déclencheur de la tâche, ou avant sa prochaine période ;

– **contraintes de régularité** (ou **gigue**) : lorsqu'une tâche fait de l'échantillonnage, elle est périodique et doit être la plus régulière possible. On retrouve le même type de contraintes lorsque la tâche doit délivrer un signal continuellement modifié.

Tous ces types de contraintes sur le système se traduisent en termes de contraintes temporelles individuelles sur les tâches. Chaque tâche est donc caractérisée par des contraintes temporelles, la difficulté réside alors dans l'étude du comportement temporel du système et du respect des contraintes définies sur chacune des tâches. L'algorithme d'ordonnancement choisi va satisfaire certaines contraintes de temps et être un compromis pour d'autres caractéristiques temporelles analysées *a posteriori*.

8.2.1 Modélisation formelle des tâches indépendantes

Afin de pouvoir analyser de manière rigoureuse l'ordonnançabilité d'une configuration de tâches, l'optimalité d'un algorithme d'ordonnancement ou la séquence d'exécution d'une application multitâche, il est nécessaire d'avoir un modèle mathématique des tâches. Ce modèle doit permettre de prendre en compte toutes les caractéristiques opérationnelles et temporelles d'une tâche d'une application quelconque. Pour cela nous allons considérer successivement les différents cas :

– tâches périodiques ;

– tâches apériodiques ;

– tâches avec contraintes de précédence ;

– tâches avec partage de ressources critiques.

La combinaison de ces différents modèles de tâches permet d'analyser une application multitâche réelle industrielle quelconque.

Le premier paramètre commun à l'ensemble des modèles de tâches est la durée $\mathbf{C_i}$ d'une tâche $\boldsymbol{\tau_i}$. La durée de la tâche est directement liée au code de la tâche. On évalue généralement la durée d'exécution pire cas $\mathbf{C_{max}}$ et une durée minimale $\mathbf{C_{min}}$. L'évaluation de cette durée de la tâche peut être effectuée de deux manières différentes. Soit le code est analysé instruction par instruction, et l'ensemble des durées de ces instructions est additionné (ensemble des durées d'exécution du processeur référencées pour un processeur et une fréquence quartz donnés). Soit la durée de la partie de code analysée est mesurée directement lors de son exécution par un système d'espionnage des signaux (analyseur logique, etc.). Mais, dans ces deux cas, une évaluation très précise de cette durée d'exécution est très difficile pour essentiellement trois raisons :

– une analyse exhaustive de tous les chemins d'exécution du code est parfois impossible (jeux de tests complexes, nombre de combinaisons très élevé) ;

– les capacités d'exécution des processeurs pour améliorer leur efficacité (mémoire cache à plusieurs niveaux, pipeline multiple, etc.) vont conduire à une incertitude sur la durée d'exécution qui dépend fortement du contexte précédent du processeur ;

– l'occurrence d'interruptions pendant l'exécution d'une tâche qui oblige le processeur à une prise en compte minimale.

Pour cette étude, nous considérerons une valeur de la durée d'exécution pire cas $\mathbf{C_i} = \mathbf{C_{max}}$ et, pour certains cas, nous analyserons l'effet de cette variabilité de la durée d'une tâche sur l'ordonnancement d'une configuration.

Modélisation des tâches périodiques

Comme nous l'avons vu dans les exemples précédents, ces tâches correspondent par exemple à des tâches de scrutation de capteurs pour effectuer des mesures régulières de grandeurs physiques. La modélisation des tâches périodiques va reposer sur trois paramètres temporels :

– $\mathbf{r_0}$: date de réveil de la tâche, c'est-à-dire la première date à laquelle la tâche demande le processeur ;
– $\mathbf{C} = \mathbf{C_{max}}$: durée d'exécution maximale définie avec les restrictions exposées précédemment ;
– **T** : période d'exécution, c'est-à-dire la fréquence de renouvellement de la demande d'exécution de la tâche.

La date de réveil $\mathbf{r_k}$ de la $\mathbf{k}^{\text{ième}}$ instance ou occurrence d'une tâche est donc définie par :

$$r_k = r_0 + kT \qquad (8.1)$$

Afin de limiter au maximum les indéterminismes d'exécution, nous supposons que les tâches sont non réentrantes, c'est-à-dire que, lors d'une nouvelle demande d'exécution, la tâche doit avoir terminé son exécution précédente. Comme nous nous sommes placés dans un environnement d'exécution stricte, la tâche doit avoir terminé son exécution avant la fin de sa période d'exécution ; la tâche est dite à **échéance sur requête**.

La figure 8.1 présente l'exécution de deux occurrences d'une tâche périodique à échéance sur requête dans un diagramme de Gantt. Nous pouvons noter que la tâche s'exécute de façon complète dans la première occurrence, c'est-à-dire que, lorsqu'elle obtient le processeur, elle garde pendant toute sa durée d'exécution $\mathbf{C} = \mathbf{C_{max}}$. En revanche, lors de la deuxième occurrence, la tâche est préemptée une fois lors de son exécution et son exécution s'effectue alors en deux fois.

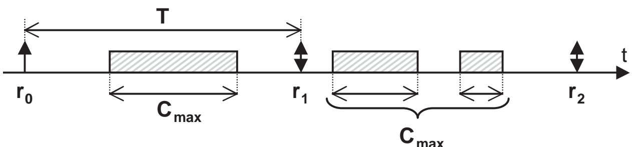


Figure 8.1 – Représentation de l'exécution d'une tâche périodique à échéance sur requête.

© Dunod – La photocopie non autorisée est un délit.

L'échéance de la tâche peut être plus courte que la fin de sa période. Dans ce cas, il est nécessaire de définir un nouveau paramètre qui est le **délai critique D** de la tâche qui correspond au délai au bout duquel la tâche doit être terminée, délai mesuré par rapport à la date de réveil de l'instance considérée.

La modélisation des tâches périodiques strictes va donc reposer sur quatre paramètres temporels :

– $\mathbf{r_0}$: date de réveil de la tâche, c'est-à-dire la première date à laquelle la tâche demande le processeur ;
– $\mathbf{C = C_{max}}$: durée d'exécution maximale définie avec les restrictions exposées précédemment ;
– **D** : délai critique, c'est-à-dire le délai au bout duquel la tâche doit être terminée par rapport à la date de réveil de l'instance en cours.
– **T** : période d'exécution, c'est-à-dire la fréquence de renouvellement de la demande d'exécution de la tâche.

Dans ce contexte, il est possible de définir l'échéance $\mathbf{d_k}$ de la $\mathbf{k}^{\text{ième}}$ instance d'une tâche par l'équation suivante :

$$d_k = r_k + D = r_0 + kT + D \qquad (8.2)$$

La figure 8.2 représente l'exécution d'une telle tâche pour deux occurrences dans un diagramme de Gantt. La zone d'exécution de la tâche se situe donc entre les dates $\mathbf{r_k}$ et $\mathbf{d_k}$, et la zone comprise entre $\mathbf{d_k}$ et $\mathbf{r_{k+1}}$ ne peut pas être utilisée pour l'exécution de la tâche.

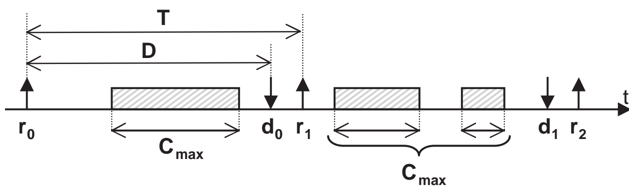


Figure 8.2 – Représentation de l'exécution d'une tâche périodique avec une échéance plus petite que la période.

■ Modélisation graphique des tâches périodiques

Nous allons nous intéresser aux trois paramètres temporels ($\mathbf{C_i}$, $\mathbf{D_i}$, $\mathbf{T_i}$) qui caractérisent une tâche $\boldsymbol{\tau_i}$. Afin d'avoir un comportement classique en temps réel (pas de réentrance des tâches : $\mathbf{D_i \leq T_i}$), ceux-ci doivent impérativement respecter les conditions de base suivantes :

$$0 \leq C_i \leq D_i \leq T_i \qquad (8.3)$$

Par conséquent, il est possible de représenter graphiquement l'espace décrivant les différents triplets caractérisant une tâche donnée. Il est nécessaire pour cela de se

donner la valeur de référence que peut prendre la durée d'exécution de la tâche $\mathbf{C_i} = \mathbf{C_{max}}$. Nous obtenons un trièdre ouvert limité par les trois demi-droites concourantes au point de coordonnées ($\mathbf{C_{max}}$, $\mathbf{C_{max}}$, $\mathbf{C_{max}}$) (figure 8.3).

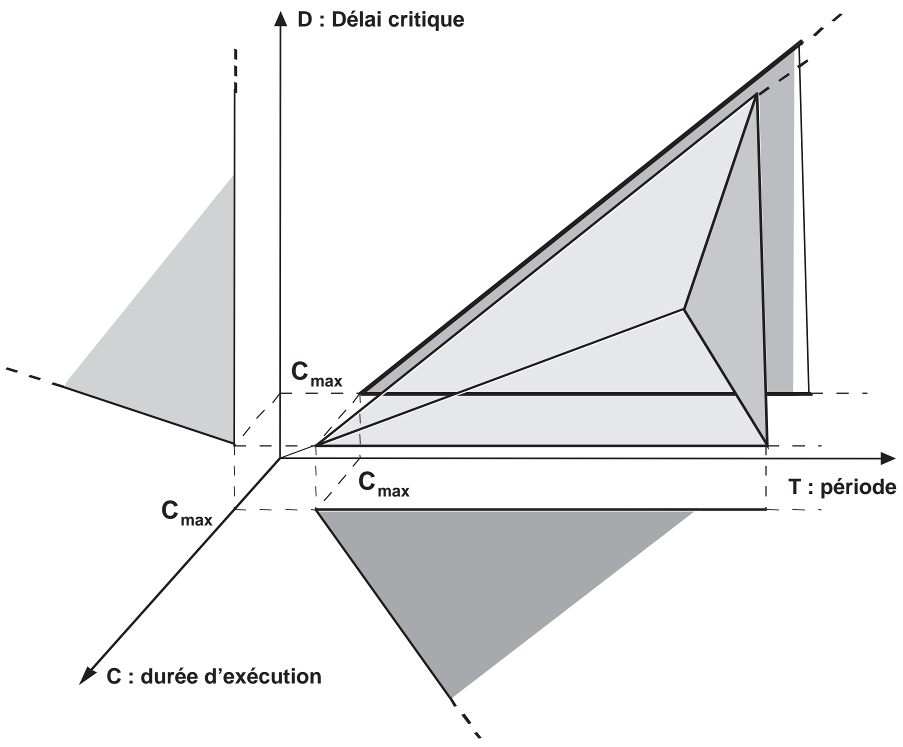


Figure 8.3 – Représentation graphique tridimensionnelle des paramètres temporels d'une tâche périodique avec une échéance plus petite que la période.

Il est évident qu'une telle représentation tridimensionnelle n'est pas d'une utilisation très aisée. Aussi il est préférable de travailler dans les projections de ce volume sur les différents plans. En particulier, nous focaliserons notre attention sur un plan **D**,**T** (délai critique-période) représenté sur la figure 8.3 et sur le plan **T**,**C** (période-durée d'exécution) représenté sur la figure 8.4.

Cette visualisation des paramètres temporels associés à une tâche est un outil d'analyse intéressant. Comme nous le verrons dans la suite, cette approche graphique s'avère particulièrement utile dans le cas où le concepteur doit soit ajouter une tâche à un ensemble de tâches avec des paramètres déjà fixés, soit modifier les paramètres d'une tâche existante afin d'améliorer ses caractéristiques temporelles. Cette visualisation graphique propose alors au concepteur toutes les possibilités de choix des paramètres temporels d'une tâche compatibles avec l'ordonnançabilité de la configuration des tâches.

© Dunod – La photocopie non autorisée est un délit.

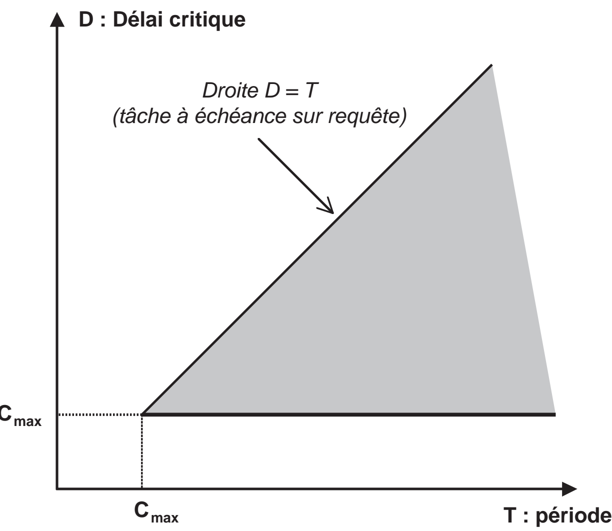


Figure 8.4 – Représentation graphique dans le plan **D,T** des paramètres temporels d'une tâche périodique avec une échéance plus petite que la période.

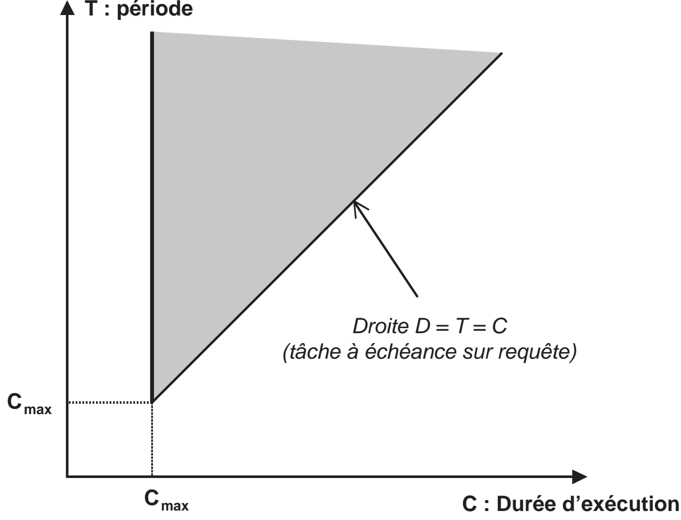


Figure 8.5 – Représentation graphique dans le plan **C,T** des paramètres temporels d'une tâche périodique avec une échéance plus petite que la période.

■ Modélisation des tâches apériodiques

En ce qui concerne les tâches dites apériodiques, le seul paramètre connu est la durée d'exécution **C** de la tâche. La date de réveil ou demande processeur est aléatoire car elle dépend du contexte d'évolution du procédé et ne peut donc pas être connue

a priori. Nous pouvons donc représenter son exécution dans le diagramme de Gantt de la figure 8.6. Les deux dates de réveil **r** et **r′** ont été choisies aléatoirement. De la même manière que pour les tâches périodiques les tâches apériodiques peuvent être préemptées au cours de leurs exécutions et donc d'exécuter en plusieurs fois lors d'une instance.

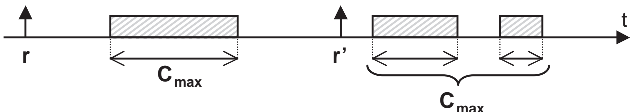


Figure 8.6 – Représentation de l'exécution d'une tâche apériodique.

Comme nous nous sommes placés dans un environnement d'exécution stricte, la tâche apériodique, appelée **apériodique stricte** ou **sporadique**, doit posséder un délai critique **D**. Nous appelons ces tâches apériodiques strictes des tâches sporadiques. L'exécution de ce type de tâches sporadiques est représentée sur la figure 8.7. Ce délai critique conduit à des dates d'échéance stricte pour chaque instance d'exécution, soit :

$$d = r + D \quad \text{ou} \quad d' = r' + D \tag{8.4}$$

Afin de limiter au maximum les indéterminismes d'exécution et de pouvoir faire une analyse de l'ordonnançabilité de configuration de tâches incluant des tâches apériodiques, nous supposons que les tâches apériodiques strictes possèdent un délai minimum $\mathbf{\Delta_{min}}$ entre deux occurrences ou instances successives ; soient **r** et **r′** deux dates de réveil successives d'une tâche sporadiques, nous avons alors la relation suivante (figure 8.7) :

$$r' - r \leq \Delta_{\text{min}} \tag{8.5}$$

Il est impératif d'avoir une tâche terminée avant une nouvelle demande d'exécution ; cela conduit aux inégalités suivantes :

$$0 \leq C \leq D \leq \Delta_{\text{min}} \tag{8.6}$$

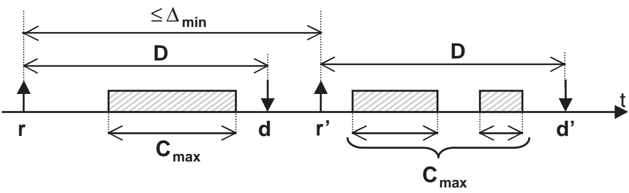


Figure 8.7 – Représentation de l'exécution d'une tâche apériodique avec une échéance stricte ou tâche sporadique.

© Dunod – La photocopie non autorisée est un délit.

8.2.2 Autres paramètres temporels des tâches

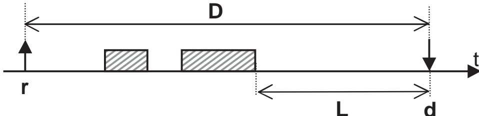
Nous allons nous intéresser à des paramètres qui sont utiles pour caractériser une ou plusieurs tâches dans une séquence d'exécution. Nous distinguons les paramètres statiques qui ne varient pas en fonction du temps d'avancement dans l'exécution de la tâche et des paramètres dynamiques qui dépendent de l'instant où ils sont calculés. Certains de ces paramètres peuvent servir de base pour un algorithme d'ordonnancement.

■ Paramètres statiques des tâches

Nous pouvons définir deux paramètres complémentaires du temps d'exécution par rapport à la période ou à l'échéance de la tâche :

– la **laxité L**, c'est-à-dire le temps restant entre la fin d'exécution de la tâche et son échéance (figure 8.8). Lors d'une exécution de la tâche au plus tôt, exécution immédiate et sans préemption après la date de réveil, la laxité maximale $\mathbf{L_{max}}$ représente tout le temps processeur restant et s'exprime par :

$$L \leq D - C \quad \text{et} \quad L_{max} = D - C \tag{8.7}$$



Exécution au plus tôt

Figure 8.8 – Illustration du paramètre « laxité » dans le cas général et dans le cas d'une exécution au plus tôt.

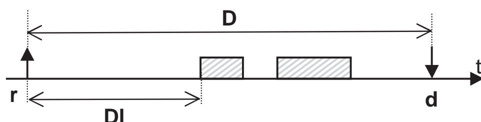
– le **délai de latence Dl**, c'est-à-dire le temps avant le début d'exécution de la tâche (figure 8.9). Lors d'une exécution de la tâche au plus tard, exécution retardée au maximum et sans préemption, le délai de latence maximum $\mathbf{Dl_{max}}$ est identique à la laxité maximale et s'exprime par :

$$Dl \leq D - C \quad \text{et} \quad Dl_{max} = D - C = L_{max} \tag{8.8}$$

En complément de ces deux paramètres statiques, il est habituel de définir les instants de début d'exécution et de fin d'exécution de la k ième instance d'une tâche, soit (figure 8.10) :

– début d'exécution de la $k^{iéme}$ instance de la tâche $\boldsymbol{\tau_i}$ **:** $\mathbf{s_{i,k}}$;

– fin d'exécution de la $k^{iéme}$ instance de la tâche $\boldsymbol{\tau_i}$ **:** $\mathbf{e_{i,k}}$.



Exécution au plus tard

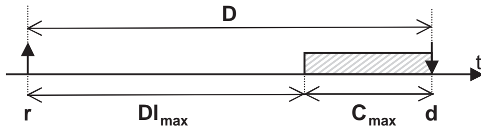


Figure 8.9 – Illustration du paramètre « délai de latence» dans le cas général et dans le cas d'une exécution au plus tard.

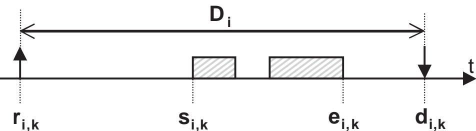


Figure 8.10 – Illustration des paramètres « début et fin d'exécution » d'une tâche τ_i.

Nous pouvons remarquer que le paramètre $\mathbf{s_{i,k}}$ correspond au délai de latence de la tâche $\boldsymbol{\tau_i}$. Étant données les caractéristiques d'exécution de la tâche, nous pouvons écrire les inégalités suivantes :

$$s_{i,k} \geq r_{i,k}, \quad e_{i,k} \leq d_{i,k} \quad \text{et} \quad e_{i,k} - s_{i,k} \geq C \tag{8.9}$$

La première inégalité est une égalité dans le cas d'une exécution au plus tôt ($\mathbf{s_{i,k} = r_k}$), la deuxième inégalité est une égalité dans le cas d'une exécution au plus tard ($\mathbf{e_{i,k} = d_k}$) et la troisième inégalité est une égalité dans le cas d'une exécution sans préemption de la tâche ($\mathbf{e_{i,k}\text{-}s_{i,k} = C}$).

Deux paramètres supplémentaires permettent de caractériser le comportement de la tâche lors de ses différentes instances d'exécution et de qualifier cette exécution. Nous avons ainsi :

– **Temps de réponse** de la k^{iéme} instance de la tâche $\boldsymbol{\tau_i}$ défini par (figure 8.11) :

$$TR_{i,k} = e_{i,k} - r_{i,k} \tag{8.10}$$

- Temps de réponse maximum de la tâche $\boldsymbol{\tau_i}$: $\mathbf{TR_i = \mathit{max}_k\{TR_{i,k}\}}$
- Temps de réponse minimum de la tâche $\boldsymbol{\tau_i}$: $\mathbf{TR_{i,min} = \mathit{min}_k\{TR_{i,k}\}}$
- Temps de réponse moyen de la tâche $\boldsymbol{\tau_i}$: $\mathbf{TR_{i,moy} = \Sigma_k\{TR_{i,k}\}/(k+1)}$

© Dunod – La photocopie non autorisée est un délit.

– **Gigue** (régularité d'exécution) entre deux instances consécutives de la tâche $\boldsymbol{\tau_i}$ définie par la relation :

$$g_{i,k} = [(s_{i,k+1} - s_{i,k}) - T_i] / T_i \tag{8.11}$$

- Gigue maximale de la tâche $\boldsymbol{\tau_i}$: $\mathbf{G_i} = \boldsymbol{max_k\{g_{i,k}\}}$
- Gigue minimale de la tâche $\boldsymbol{\tau_i}$: $\mathbf{G_{i,min}} = \boldsymbol{min_k\{g_{i,k}\}}$
- Gigue moyenne de la tâche $\boldsymbol{\tau_i}$: $\mathbf{G_{i,moy}} = \boldsymbol{\Sigma_k\{g_{i,k}\}/(k+1)}$

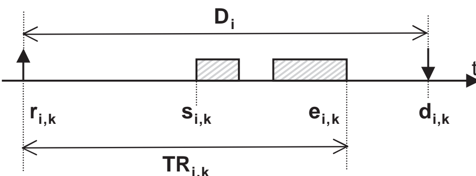


Figure 8.11 – Temps de réponse de la k^ième exécution d'une tâche τ_i.

Paramètres dynamiques des tâches

Au cours de l'exécution de l'instance d'une tâche, certains paramètres, qui sont fonction de l'instant **t**, peuvent être utiles à l'ordonnanceur. Étant donnée une tâche avec une durée d'exécution **C**, nous pouvons définir le temps d'exécution restant **C(t)** fonction du temps processeur déjà alloué à la tâche $\mathbf{C_{exécuté}}$ au cours de cette instance. Ainsi, nous pouvons identifier les trois paramètres suivants (figure 8.12) :

– **le temps d'exécution restant C(t)** : $C(t) = C - C_{exécuté}$

– **le délai critique dynamique D(t)**, c'est-à-dire le temps restant avant la prochaine échéance, soit :

$$D(t) = d - t \tag{8.12}$$

– **la laxité dynamique L(t)**, c'est-à-dire le temps avant le début d'exécution de la tâche, soit :

$$L(t) = D(t) - C(t) = d - t - C(t) \tag{8.13}$$

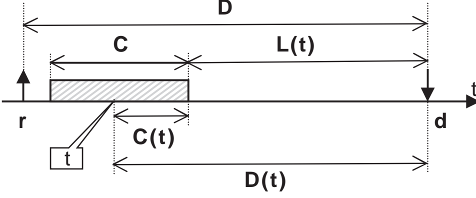


Figure 8.12 – Paramètres dynamiques de l'instance d'une tâche en cours d'exécution.

Il est intéressant d'analyser l'évolution de ces paramètres dynamiques au cours de l'exécution d'une tâche. Considérons une tâche $\boldsymbol{\tau_i}$ possédant les paramètres initiaux $\mathbf{C_i}$, $\mathbf{D_i}$ et $\mathbf{L_i}$ (par définition : $\mathbf{L_i} = \mathbf{D_i}$-$\mathbf{C_i}$). Dans l'intervalle d'exécution d'une instance de la tâche [**r**,**d**], les paramètres dynamiques **D(t)** et **L(t)** vont évoluer en fonction de l'allocation ou non du processeur à cette tâche entre les instants [**t**, **t**+1] selon les règles suivantes :

– La tâche obtient le processeur pendant l'intervalle [**t**, **t**+1], nous avons :

$$D(t+1) = D(t) - 1 \quad \text{et} \quad L(t+1) = L(t) \tag{8.14}$$

– La tâche n'obtient pas le processeur pendant l'intervalle [**t**, **t**+1], nous avons :

$$D(t+1) = D(t) - 1 \quad \text{et} \quad L(t+1) = L(t) - 1 \tag{8.15}$$

Nous pouvons remarquer que le délai dynamique diminue toujours d'une unité alors que la laxité diminue uniquement si la tâche ne s'exécute pas. Prenons un exemple simple où la tâche possède les paramètres initiaux suivants :

$$C_i = 2, \quad D_i = 7 \quad \text{et} \quad L_i = 7 - 2 = 5$$

Nous allons analyser l'évolution des deux paramètres dynamiques **D(t)** et **L(t)** en fonction du temps lors de l'exécution de la tâche. La figure 8.13 représente l'évolution de ces deux paramètres dans un plan (D,L) en considérant deux trajectoires d'exécution de la tâche au cours du temps. Le point initial de « fonctionnement » de la tâche à son réveil est (7,5). Étant donné les équations 8.14 et 8.15, une flèche horizontale représente l'exécution de la tâche (le processeur est alloué à cette tâche) et la flèche en diagonale note l'attente de la tâche. La durée de la tâche étant de 2 unités de temps, il suffit de deux flèches horizontales pour terminer l'exécution de la tâche. La première trajectoire d'exécution, qui commence par une exécution

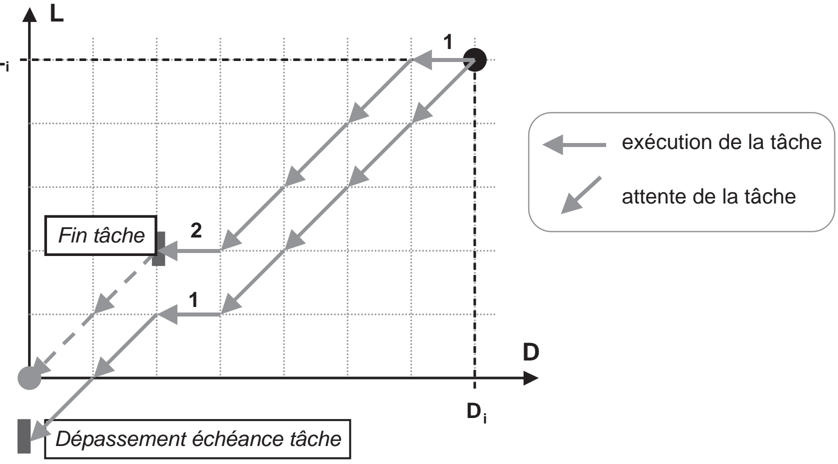


Figure 8.13 – Exemple de l'évolution des paramètres dynamiques de l'instance d'une tâche en cours d'exécution.

© Dunod – La photocopie non autorisée est un délit.

de la tâche au temps $\mathbf{t} \in [0,1]$, termine son exécution au temps $\mathbf{t} = 5$. La deuxième trajectoire d'exécution intègre une exécution effective au temps $\mathbf{t} \in [4,5]$; mais la deuxième exécution n'est pas réalisée avant l'échéance ce qui est visualisé par l'intersection de la trajectoire avec l'axe des abscisses.

■ Remarques sur les paramètres temporels des tâches

En ce qui concerne les quatre paramètres de base d'une tâche $\boldsymbol{\tau}_\mathbf{i}$ ($\mathbf{r_i}$, $\mathbf{C_i}$, $\mathbf{D_i}$, $\mathbf{T_i}$), nous pouvons analyser l'origine de leur détermination :

– $\mathbf{r_i}$: Dans le cas général, les tâches sont à départ simultané ; mais une tâche peut être retardée au réveil de l'application pour prendre en compte par exemple la précédence. Ce réveil non simultané est obtenu en insérant au début du code de la tâche une primitive de type « DELAI ».

– $\mathbf{C_i}$: La durée de la tâche est directement liée au code de la tâche. On évalue généralement la durée d'exécution pire cas $\mathbf{C_{max}}$ et une durée minimale $\mathbf{C_{min}}$.

– $\mathbf{D_i}$: Ce paramètre permet au concepteur de limiter le temps de réponse de la tâche. Il peut être codé au niveau d'une tâche en utilisant un temporisateur type « chien de garde ou *watchdog* ».

– $\mathbf{T_i}$: Cette périodicité de la tâche est fixée par les besoins de la fonction : tâche de scrutation ou acquisition (*polling*)

Ces choix sont donc soit liés aux besoins de l'application (début d'exécution, périodicité, durée du code) soit imposés par le concepteur (échéance). Aussi, dans la représentation graphique des paramètres possibles d'une tâche périodique, nous avons une limitation due à ces choix. Ainsi, viennent s'ajouter des contraintes temporelles exprimées dans le cahier des charges de l'application comme par exemple :

– une période maximale $\mathbf{T_{max}}$ ($\mathbf{T_i} = \mathbf{T_{max}}$), correspondant, par exemple, à la fréquence minimale d'échantillonnage ;

– une période minimale $\mathbf{T_{min}}$ ($\mathbf{T_i} = \mathbf{T_{min}}$), correspondant par exemple une période inutile d'acquisition (trop de données à analyser ou redondance des données) ;

– une échéance maximale $\mathbf{D_{max}}$ ($\mathbf{D_i} = \mathbf{D_{max}}$), afin d'obtenir un meilleur temps de réponse.

La figure 8.4, représentant une tâche périodique quelconque, est modifiée selon ces nouvelles contraintes.

8.2.3 Modélisation des tâches dépendantes

Comme nous l'avons déjà vu, la dépendance des tâches entre elles peut se traduire de deux manières : une relation de précédence au niveau de l'ordre d'exécution de deux ou plusieurs tâches ou le partage de ressources critiques. Il est donc nécessaire de pouvoir modéliser ces deux dépendances entre tâches.

■ Modélisation de la précédence entre les tâches

De façon générale, cette précédence entre tâches est mise en œuvre à l'aide de primitives de synchronisations ou communications intégrées dans le code des deux tâches. Prenons un exemple simple et considérons que deux tâches $\boldsymbol{\tau}_\mathbf{1}$ et $\boldsymbol{\tau}_\mathbf{2}$ ne sont pas

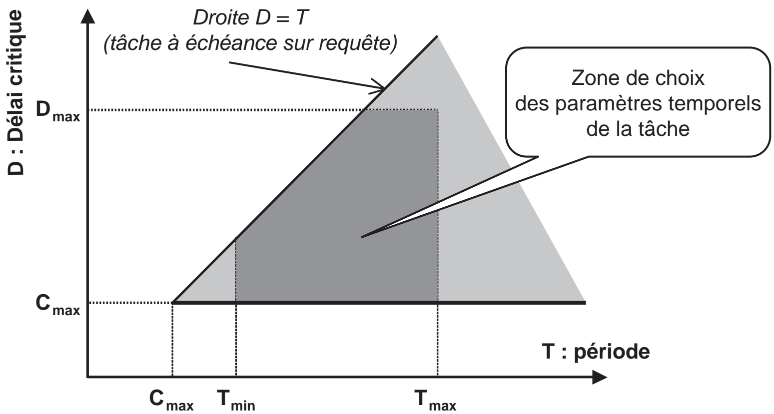


Figure 8.14 – Représentation graphique des paramètres (D,T) d'une tâche caractérisée par des contraintes temporelles émanant du cahier des charges.

atomiques, c'est-à-dire que ces primitives sont situées à l'intérieur du code de chacune des tâches (figure 8.15). La séquence d'exécution, présentée sur la figure 8.15, montre l'attente de l'exécution de la partie $\tau_{2\text{-}1}$ de la tâche τ_2 par la partie $\tau_{1\text{-}2}$ de la tâche τ_1. Nous avons l'effet de la synchronisation de la tâche τ_1 par la tâche τ_2.

La première étape à réaliser est la transformation des tâches en tâches atomiques afin d'avoir un modèle de tâches conforme à notre étude. Ce travail est simple à effectuer, il consiste à découper les tâches au niveau des primitives de synchronisations (dans notre exemple : figure 8.16) ou de communications. Ce découpage conduit à quatre tâches qui ne comportent maintenant aucune primitive bloquante à l'intérieur du code, à l'exception d'une primitive d'attente qui peut se situer au début du code. Ainsi, nous pouvons donner une règle générale de transformation pour obtenir des tâches sous forme atomique :

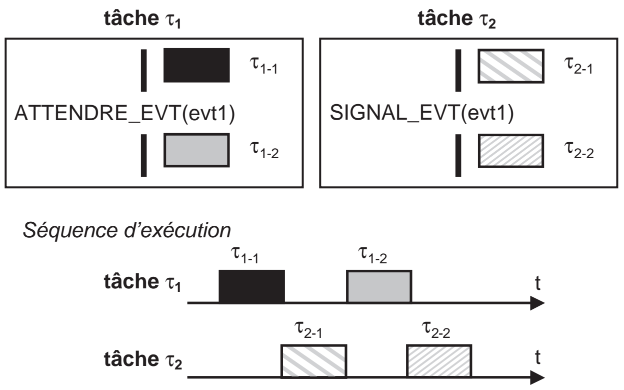


Figure 8.15 – Relation de précédence d'exécution entre deux tâches : codes et séquences d'exécution.

© Dunod – La photocopie non autorisée est un délit.

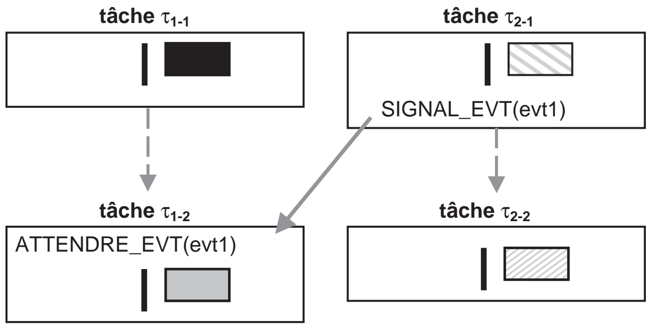


Figure 8.16 – Découpage des tâches en tâches atomiques.

– **Attente** de synchronisation ou de communication en début de tâche.

– **Envoi** d'événement de synchronisation ou de communication en fin de tâche.

L'ensemble des quatre tâches obtenues est rigoureusement équivalent au niveau de l'exécution à l'ensemble précédent des deux tâches. En revanche, afin de signifier clairement, les relations entre ces quatre tâches, il est nécessaire d'adjoindre à ces tâches un graphe de relation, présenté sur la figure 8.17 pour notre exemple. Nous pouvons noter que nous avons deux types de relations : des liens d'exécution traduisant le découpage effectué et le lien de précédence donné par les primitives initiales.

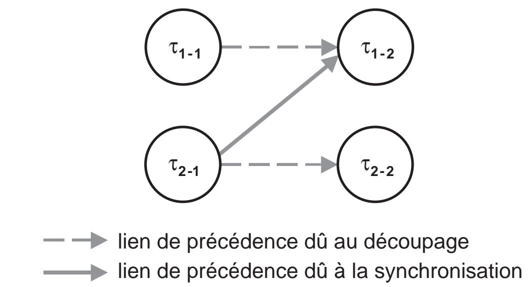


Figure 8.17 – Traduction de la relation de précédence d'exécution entre deux tâches par un graphe de précédence.

Une remarque importante peut être faite au niveau de ce découpage en tâche atomique et des primitives de synchronisation. En effet, après avoir réalisé le découpage en tâches atomiques, les primitives de synchronisation ne sont plus utiles si l'on associe aux tâches obtenues le graphe de relation. Ce graphe, dit graphe de précédence, traduit effectivement toutes les relations nécessaires au comportement correct de l'application conforme au comportement initial souhaité par le concepteur. Comme nous l'analyserons dans la suite de ce chapitre, l'existence ou non de ces primitives d'« attente » au début des codes des tâches atomiques peut conduire à des dysfonctionnements ou anomalies de comportement.

■ Modélisation du partage de ressources critiques entre tâches

Une tâche τ_i de durée totale C_i qui utilise une ressource critique **R** possède dans son code une zone protégée, appelée section critique, pendant laquelle elle accède à cette ressource. Cette section critique Sc_i est protégée par des primitives permettant de gérer l'exclusion mutuelle comme un sémaphore. Par conséquent, en termes de temps, l'exécution de cette tâche peut être décrite par 3 valeurs (figure 8.18) :

– $C_{i,\alpha}$: temps avant la section critique ;

– $C_{i,\beta}$: durée de la section critique (ressource utilisée) ;

– $C_{i,\gamma}$: temps après la section critique.

Ces trois valeurs doivent satisfaire à l'égalité suivante :

$$C_i = C_{i,\alpha} + C_{i,\beta} + C_{i,\gamma} \quad (8.16)$$

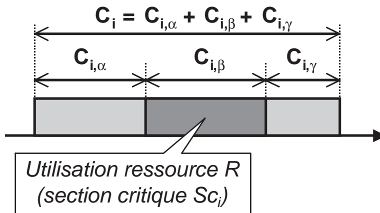


Figure 8.18 – Représentation temporelle d'une tâche contenant une section critique.

Si une tâche τ_i de durée totale C_i utilise plusieurs ressources, les sections critiques doivent être correctement imbriquées. Soit $Sc_{i,1}$ (resp. $Sc_{i,2}$) la section critique de la tâche τ_i utilisant la ressource R_1 (resp. R_2), nous devons avoir l'une des conditions suivantes :

$$Sc_{i,1} \subset Sc_{i,2} \quad \text{ou} \quad Sc_{i,2} \subset Sc_{i,1} \quad \text{ou} \quad Sc_{i,1} \cap Sc_{i,2} = \varnothing \quad (8.17)$$

Considérons l'exemple d'une tâche τ_i utilisant 3 ressources R_1, R_2 et R_3 ; sa durée d'exécution ($C_i = 10$) peut être décrite par le tableau 8.1.

Tableau 8.1 – Modèle temporelle d'une tâche partageant trois ressources critiques avec d'autres tâches.

Ressource	Durée $C_{i,\alpha}$	Durée $C_{i,\beta}$	Durée $C_{i,\gamma}$	Durée totale C_i
R_1	1	2	7	10
R_2	4	5	1	10
R_3	5	2	3	10

© Dunod – La photocopie non autorisée est un délit.

La figure 8.19 représente cette répartition des différentes sections critiques $\mathbf{Sc_{i,}}$, $\mathbf{Sc_{i,2}}$ et $\mathbf{Sc_{i,3}}$ sur la durée totale $\mathbf{C_i}$ de la tâche. Nous pouvons vérifier que ces sections critiques vérifient deux à deux l'une des trois conditions 8.17 :

$$Sc_{i,3} \subset Sc_{i,2} \quad \text{et} \quad Sc_{i,1} \cap Sc_{i,2} = \varnothing \quad \text{et} \quad Sc_{i,1} \cap Sc_{i,3} = \varnothing$$

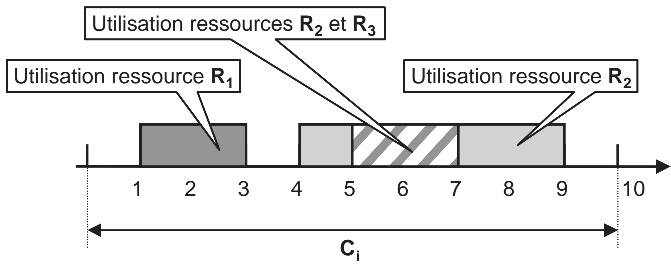


Figure 8.19 – Représentation temporelle d'une tâche contenant trois sections critiques.

8.2.4 Analyse d'une configuration de tâches périodiques

■ Charge du processeur

Pour une tâche périodique donnée $\boldsymbol{\tau_i}$, définie avec les quatre paramètres de base ($\mathbf{r_i}$, $\mathbf{C_i}$, $\mathbf{D_i}$, $\mathbf{T_i}$), nous pouvons définir les occupations du processeur :

– Le **facteur d'utilisation u** comme le pourcentage du processeur nécessaire à son exécution sur sa période $\mathbf{T_i}$, soit :

$$u_i = C_i / T_i \tag{8.18}$$

– Pour une tâche périodique donnée $\boldsymbol{\tau_i}$, qui n'est pas à échéance sur requête, on définit le **facteur de charge** $\mathbf{u_{l,i}}$ comme le pourcentage du processeur nécessaire à son exécution sur son délai critique $\mathbf{D_i}$:

$$u_{i,1} = C_i / D_i \tag{8.19}$$

Pour une configuration de **n** tâches périodiques : $T = \{\boldsymbol{\tau_1}, \boldsymbol{\tau_2}, \ldots, \boldsymbol{\tau_n}\}$, on définit le facteur d'utilisation **U** (resp. le facteur de charge $\mathbf{U_l}$) comme la somme des facteurs d'utilisation $\mathbf{u_i}$ des tâches (resp. des facteurs de charge des tâches $\mathbf{u_{l,i}}$) de la configuration de tâche T :

$$U = \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} C_i / T_i \tag{8.20}$$

$$U_i = \sum_{i=1}^{n} u_{i,1} = \sum_{i=1}^{n} C_i / D_i \tag{8.21}$$

Considérons l'exemple d'une configuration à trois tâches périodiques définies par les paramètres temporels donnés dans le tableau 8.2.

Tableau 8.2 – Exemple d'une configuration de trois tâches périodiques.

Tâche	r_i	C_i	D_i	T_i	u_i	$u_{l,i}$
τ_1	0	2	6	6	0,333	0,333
τ_2	0	1	8	8	0,125	0,125
τ_3	0	2	10	12	0,166	0,2

Les facteurs d'utilisation $\mathbf{u_i}$ et les facteurs de charge $\mathbf{u_{l,i}}$ de chacune des tâches sont calculés et présentés dans les deux dernières colonnes du tableau 8.2. Le facteur d'utilisation **U** et le facteur de charge $\mathbf{U_l}$ de la configuration sont donc les suivants :

$$U = \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} C_i / T_i = \frac{2}{6} + \frac{1}{8} + \frac{2}{12} = \frac{5}{8} = 0{,}625$$

$$U_l = \sum_{i=1}^{n} u_{l,i} = \sum_{i=1}^{n} C_i / D_i = \frac{2}{6} + \frac{1}{8} + \frac{2}{10} = \frac{79}{120} = 0{,}625$$

Nous pouvons noter que les deux facteurs **U** et $\mathbf{U_l}$ sont identiques si toutes les tâches sont à échéance sur requête.

De façon évidente, nous avons une condition nécessaire d'ordonnançabilité de la configuration définie à partir du facteur d'utilisation et du facteur de charge d'une configuration de tâches périodiques s'exécutant sur une plate-forme monoprocesseur. Cette condition exprime que l'utilisation du processeur ne peut pas dépasser les 100 %, soit l'inégalité à vérifier :

$$U = \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} C_i / D_i \leq 1 \tag{8.22}$$

■ Période d'étude

Pour une configuration de **n** tâches périodiques $T = \{\boldsymbol{\tau_1}, \boldsymbol{\tau_2}, \ldots, \boldsymbol{\tau_n}\}$ à départ simultané ($\forall$i, $\mathbf{r_i} = 0$), la séquence d'exécution se retrouve régulièrement dans une situation identique au niveau des demandes processeur lorsque toutes les tâches sont à nouveau en phase. Ainsi, l'étude de la séquence d'exécution, produite par un algorithme d'ordonnancement donné, peut se limiter à un temps **H** appelé période d'étude ou méta-période ou cycle majeur qui est défini par :

$$H = PPCM\{T_i\}_{i \in [1,n]} \tag{8.23}$$

© Dunod – La photocopie non autorisée est un délit.

Considérons l'exemple de la configuration à trois tâches périodiques à départ simultané définies par les paramètres temporels donnés dans le tableau précédent 8.2. La période d'étude est donc :

$$H = PPCM\{T_i\}_{i \in [1,3]} = PPCM\{6,8,12\} = 24$$

Ainsi, la séquence d'exécution se répète avec une période **H** selon un motif défini lors de la première période d'étude (figure 8.20).

Figure 8.20 – Séquence d'exécution d'une configuration de tâches périodiques à départ simultané.

Si au moins une des tâches possède une date de réveil différente de celles des autres tâches, alors la séquence d'exécution commence par une phase dite transitoire dont la durée maximale $\mathbf{H_{transitoire}}$ est donnée par :

$$H_{\text{transitoire}} = Max\{r_i\}_{i \in [1,n]} + PPCM\{T_i\}_{i \in [1,n]} \quad (8.24)$$

À partir d'un instant situé dans l'intervalle $[0,\mathbf{H_{transitoire}}]$, la séquence d'exécution va prendre une forme identique à celle précédemment étudiée, c'est-à-dire périodique et de période égale à **H**. Ainsi, la séquence d'exécution comporte deux phases : une phase transitoire de durée inférieure ou égale à $\mathbf{H_{transitoire}}$ et une phase stationnaire répétée à l'infini de durée **H** (figure 8.21).

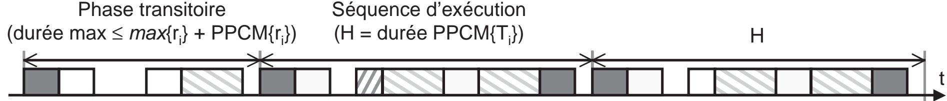


Figure 8.21 – Séquence d'exécution d'une configuration de tâches périodiques avec au moins une tâche possédant une date de réveil différente des autres tâches.

Considérons l'exemple précédent pour lequel nous avons décalé la date de réveil de la tâche $\boldsymbol{\tau_2}$ de 2 (tableau 8.3). La durée maximale de la phase transitoire de la séquence d'exécution est donc :

$$\begin{aligned} H_{\text{transitoire}} &= Max\{r_i\}_{i \in [1,3]} + PPCM\{T_i\}_{i \in [1,3]} \\ &= Max\{0,2,0\} + 24 = 2 + 24 = 26 \end{aligned}$$

En réalité, l'analyse de la séquence d'exécution montre que la périodicité **H** = 24 commence dès le début de la séquence (figure 8.22). En revanche, la séquence d'exécution n'est pas la même que celle de la configuration avec les tâches à départ simul-

Tableau 8.3 – Exemple d'une configuration de trois tâches périodiques avec une tâche à départ différée.

Tâche	r_i	C_i	D_i	T_i
τ_1	0	2	6	6
τ_2	2	1	8	8
τ_3	0	2	10	12

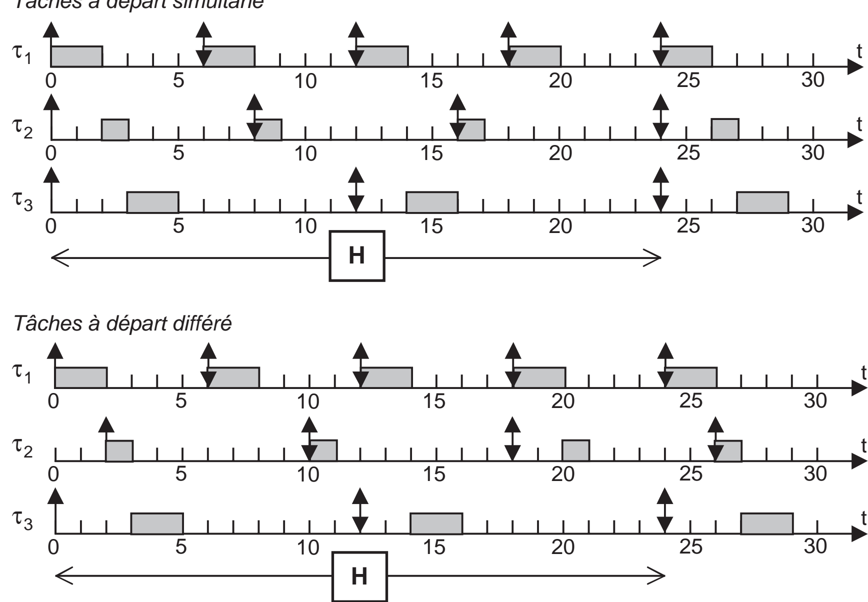


Figure 8.22 – Comparaisons des séquences d'exécution d'une configuration de tâches périodiques à départ simultané (tableau 8.2) et à départ différé (tableau 8.3).

tanée (tableau 8.2). Il est important de noter qu'il existe d'autres séquences d'exécution, et que la périodicité de la séquence et la durée de la phase transitoire sont indépendantes de l'algorithme d'ordonnancement utilisé puisqu'elles sont liées uniquement à la charge processeur à un instant donné.

Si beaucoup de configuration de tâches à départ différé se comporte d'un point de vue périodicité comme la même configuration de tâche mais à départ simultané, il existe des configurations de tâche présentant des périodicités d'exécution qui commencent après une phase transitoire.

Prenons l'exemple de la configuration de trois tâches décrite dans le tableau 8.4 qui possède une période d'étude **H** = 12. La durée maximale de la phase transitoire de la séquence d'exécution est donc :

© Dunod – La photocopie non autorisée est un délit.

$$
\begin{aligned}
H_{\text{transitoire}} &= Max\{r_i\}_{i \in [1,3]} + PPCM\{T_i\}_{i \in [1,3]} \\
&= Max\{0,1,3\} + 12 = 15
\end{aligned}
$$

L'analyse de la séquence d'exécution de la figure 8.23 montre que la phase transitoire se termine à l'instant 8, valeur comprise dans l'intervalle [0,15]. Ensuite la séquence d'exécution est périodique et de période **H** = 12.

Tableau 8.4 – Exemple d'une configuration de trois tâches périodiques avec une tâche à départ différée qui présente une phase transitoire au niveau de l'exécution.

Tâche	r_i	C_i	D_i	T_i
τ_1	0	1	4	4
τ_2	1	3	6	6
τ_3	3	1	4	4

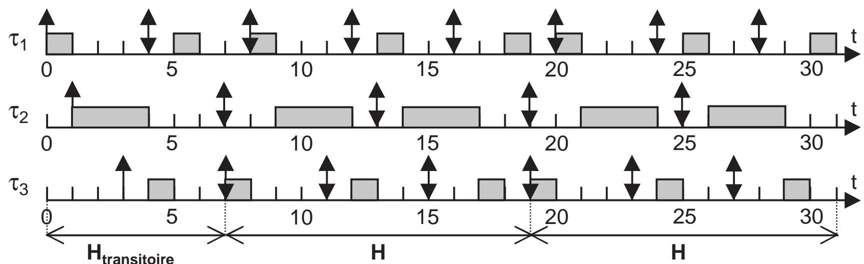


Figure 8.23 – Séquence d'exécution d'une configuration de tâches périodiques à départ différé (tableau 8.4) comprenant une phase transitoire.

■ Temps libre processeur

Avec les notions de charges processeur et de période d'étude, nous pouvons introduire une caractéristique supplémentaire d'une séquence d'exécution : le temps libre $\mathbf{T_{libre}}$, ou temps creux ou temps oisif processeur. Ainsi, pour une configuration de **n** tâches périodiques à départ simultané : $T = \{\boldsymbol{\tau_1}, \boldsymbol{\tau_2}, \ldots, \boldsymbol{\tau_n}\}$ s'exécutant sur un système monoprocesseur, le facteur d'utilisation du processeur **U** étant définie sur la période d'étude **H**, il est possible d'évaluer le temps non utilisé ou temps libre du processeur :

$$
T_{\text{libre}} = (1 - U) \cdot H = \left(1 - \sum_{i=1}^{n} C_i / T_i\right) \cdot PPCM\{T_i\}_{i \in [1,n]} \quad (8.25)
$$

Si nous considérons l'exemple de configuration du tableau 8.2, les temps libres processeur, visualisés sur la figure 8.24, sont :

$$T_{\text{libre}} = (1 - 0{,}625) \times 24 = 9$$

Il est important de remarquer que, travaillant en général avec des algorithmes d'ordonnancement au plus tôt (les temps creux surviennent lorsqu'aucune tâche n'est prête), les temps libres se situeront préférentiellement en fin de séquence d'exécution. Ce résultat peut être étendu pour une configuration de tâches à départ non simultané en considérant uniquement la partie de la séquence reproduite à l'infinie (de durée **H**) qui se situe après la phase transitoire. En effet, des temps libres processeur peuvent se situer dans la phase transitoire de la séquence d'exécution correspondant seulement au fait que les tâches ne sont pas à départ simultané.

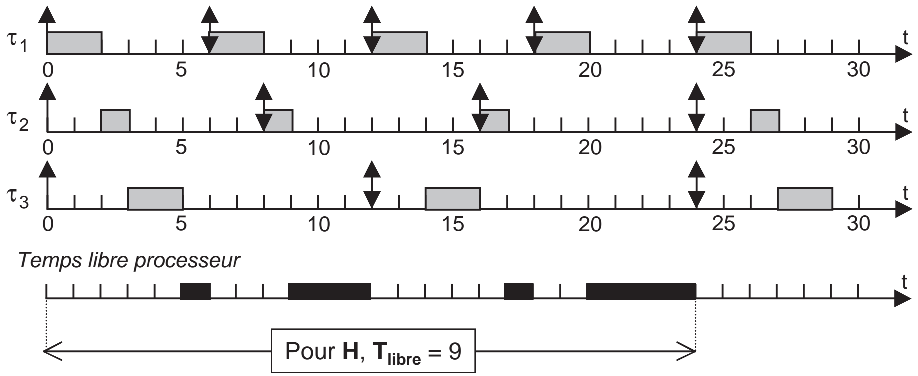


Figure 8.24 – Séquence d'exécution d'une configuration de tâches périodiques à départ différé (tableau 8.4) comprenant une phase transitoire.

Séquence saturée

Une séquence est dite **saturée** si l'allocation du processeur est complète (**U** = 1), c'est-à-dire que la séquence ne comporte pas de degré de liberté. Ainsi, pour une configuration de **n** tâches périodiques à départ simultané : $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$ s'exécutant sur un système monoprocesseur, cela peut s'exprimer en fonction des paramètres temporels des tâches sous la forme suivante :

$$Max\{d_i\}_{i \in [1,n]} - Min\{r_i\}_{i \in [1,n]} = \sum_{i=1}^{n} C_i \qquad (8.26)$$

La figure 8.25 représente une séquence d'exécution saturée pour une configuration à deux tâches τ_1 (r_1, C_1, D_1, T_1) et τ_2 (r_2, C_2, D_2, T_2) avec $r_1 < r_2$ et $d_1 < d_2$. Ainsi, en fonction de la relation 8.26, nous devons vérifier :

$$d_2 - r_1 = C_1 + C_2$$

© Dunod – La photocopie non autorisée est un délit.

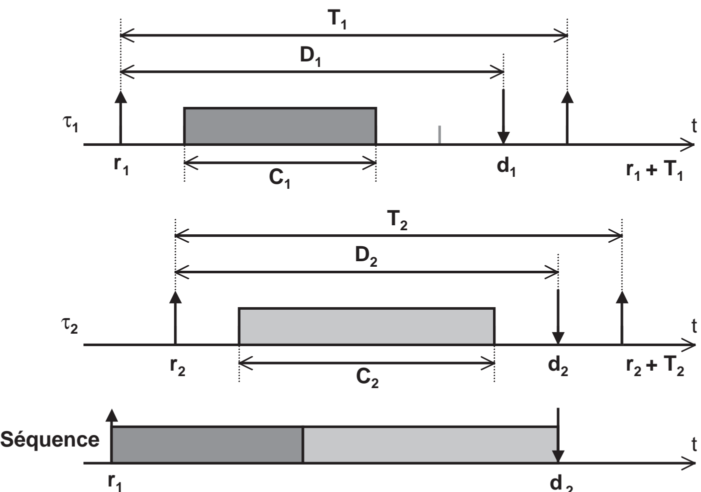


Figure 8.25 – Exemple de séquence d'exécution saturée
pour une configuration à deux tâches périodiques.

■ Notion d'instant critique

Dans la suite de ce chapitre, nous allons avoir à analyser des configurations de tâches qui peuvent avoir des départs différés. Nous avons vu dans les sections précédentes que le fait d'avoir un départ différé d'au moins une tâche peut avoir une incidence sur la durée d'analyse puisque cela introduit une phase transitoire dans l'exécution de l'application. De plus les tâches peuvent avoir des dates de réveil différées par rapport aux autres tâches qui évoluent lors de plusieurs lancements de l'application. Il serait donc intéressant de déterminer l'ordonnançabilité de la configuration dans la situation de pire cas en ce qui concerne le départ des tâches.

Sans réaliser une démonstration rigoureuse, il est aisé de voir que la situation de pire cas se produit lorsque toutes les tâches sont à départ simultané. Ainsi, l'ordonnançabilité d'une configuration peut être vérifiée lorsque les tâches sont toutes à départ simultané : activation au même instant appelé **instant critique** (pire cas). L'exemple, présenté sur la figure 8.26, montre les trois séquences d'exécution obtenues pour une configuration de deux tâches dont les paramètres sont donnés dans le tableau 8.5. La tâche τ_1 a une date de réveil qui prend les valeurs 4, 2 et 0. Pour ces différentes valeurs de dates de réveil, nous obtenons respectivement les temps de réponse de la deuxième tâche τ_2 égaux à 12, 13 et 14. Ainsi, nous pouvons conclure sur cet exemple que la situation la plus critique pour l'exécution de cette configuration se situe lorsque les deux tâches sont en phase.

Ainsi, nous étudierons autant que faire se peut des configurations de tâche à départ simultané (instant critique), sachant que la même configuration avec un ou plusieurs départs différés peut conduire à une situation ordonnançable. En effet, si une configuration de tâches est ordonnançable avec les tâches à départ simultané, elle le sera obligatoirement si une ou plusieurs tâches sont à départ différé. En revanche, si une

configuration de tâches n'est pas ordonnançable avec les tâches à départ simultané, la même configuration avec des tâches à départ différé peut être ordonnançable.

Tableau 8.5 – Exemple d'une configuration de deux tâches périodiques avec une tâche à départ variable.

Tâche	r_i	C_i	D_i	T_i
τ_1	4,2,0	1	4	4
τ_2	0	10	14	14

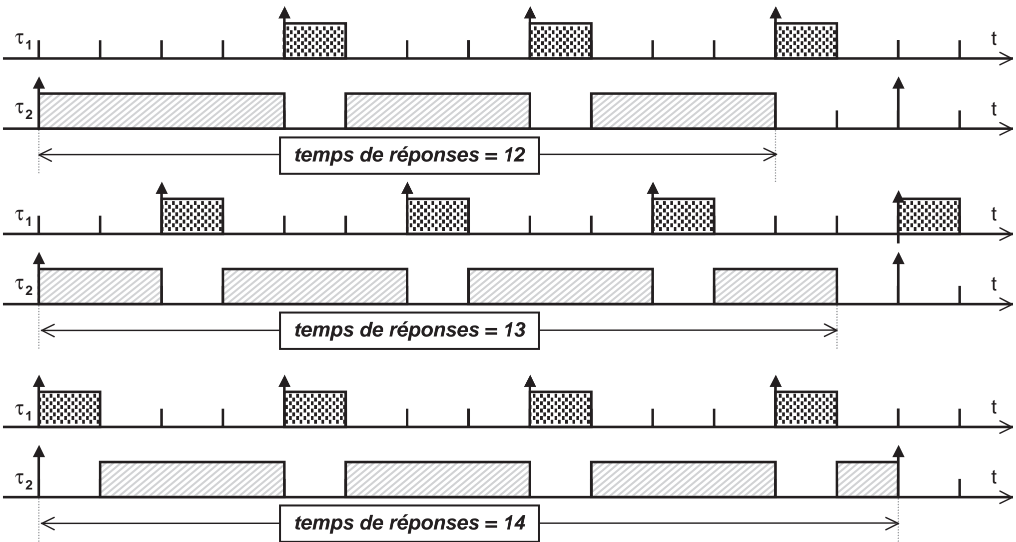


Figure 8.26 – Exemple de séquences d'exécution pour une configuration à deux tâches périodiques avec trois dates de réveil différentes : analyse de l'instant critique.

8.3 Ordonnancement des tâches indépendantes périodiques

8.3.1 Algorithmes d'ordonnancement à priorités fixes

■ Algorithme d'ordonnancement « Rate Monotonic »

Dans un contexte de tâches indépendantes, périodiques, à échéance sur requête et à départ simultané, nous allons effectuer une affectation des priorités aux tâches selon la période : plus la période de la tâche est petite, plus la priorité de la tâche est grande. La tâche conserve cette priorité pendant toute son exécution. Cette règle d'affectation

© Dunod – La photocopie non autorisée est un délit.

des priorités est appelée l'algorithme d'ordonnancement « Rate Monotonic ou RM ». Concernant cet algorithme d'ordonnancement pour une configuration de **n** tâches ayant les paramètres ($\mathbf{r_i}$, $\mathbf{C_i}$, $\mathbf{D_i}$, $\mathbf{T_i}$) avec $\mathbf{r_i} = 0$ pour tout i et $\mathbf{D_i} = \mathbf{T_i}$, nous avons les résultats suivants :

– l'algorithme d'ordonnancement RM est **optimal** dans la classe des algorithmes à priorités fixes, c'est-à-dire que, si une configuration de tâches est ordonnançable, elle le sera en affectant les priorités selon RM ;

– une **condition suffisante d'ordonnançabilité** d'une configuration est obtenue pour un facteur d'utilisation **U** du processeur suivant l'inégalité suivante :

$$U = \sum_{i=1}^{n} C_i / T_i \leq n\left(2^{\frac{1}{n}} - 1\right) \qquad (8.27)$$

La figure 8.27 représente la variation de la valeur de cette condition suffisante en fonction du nombre de tâches. Nous pouvons faire deux remarques concernant cet algorithme RM :

– l'asymptote de la courbe visualisée sur la figure 8.27 montre qu'une configuration de tâches indépendantes, périodiques, à échéance sur requête et à départ simultané sera ordonnançable si le facteur d'utilisation du processeur ne dépasse pas 69 % ;

– la condition étant suffisante, une configuration de tâches indépendantes, périodiques, à échéance sur requête et à départ simultané, dont le facteur d'utilisation du processeur dépasse la valeur obtenue par l'équation 8.27, peut être ordonnançable avec cette affectation de priorités selon RM.

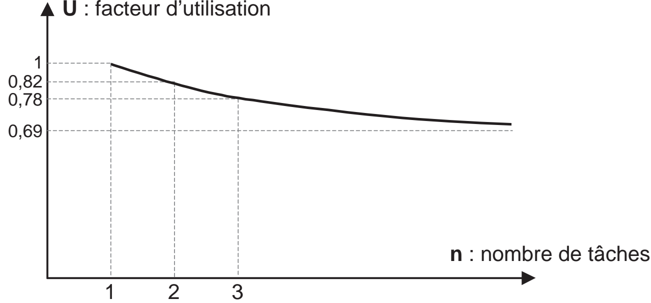


Figure 8.27 – Représentation de la condition suffisante d'ordonnançabilité d'une configuration traitée avec l'algorithme RM.

Prenons l'exemple de la configuration à trois tâches décrite dans le tableau 8.6. Nous vérifions que nous sommes en présence d'une configuration de tâches indépendantes, périodiques, à échéance sur requête et à départ simultané. Les priorités ont été affectées selon l'algorithme d'ordonnancement RM. Le facteur d'utilisation **U** de cette configuration est donné par :

$$U = \sum_{i=1}^{3} C_i / T_i = \frac{20}{100} + \frac{40}{150} + \frac{100}{350} \approx 0{,}75 \le 3\left(2^{\frac{1}{3}} - 1\right) \approx 0{,}779$$

Par conséquent, nous pouvons en déduire que cette configuration de tâches est ordonnançable. Nous pouvons construire une partie de la séquence qui aura pour période **H** :

$$H = PPCM\{T_i\}_{i \in [1,\,3]} = PPCM\{100{,}150{,}350\} = 2100$$

Cette séquence, représentée sur la figure 8.28, montre en particulier les trois préemptions de la tâche de plus faible priorité $\boldsymbol{\tau_3}$ alors que la tâche de plus forte priorité $\boldsymbol{\tau_1}$ s'exécute dés sa demande (dates de réveil).

Tableau 8.6 – Exemple d'une configuration de trois tâches périodiques avec une affectation des priorités selon RM.

Tâche	r_i	C_i	D_i	T_i	Priorité selon RM
τ_1	0	20	100	100	**3**
τ_2	0	40	150	150	**2**
τ_3	0	100	350	350	**1**

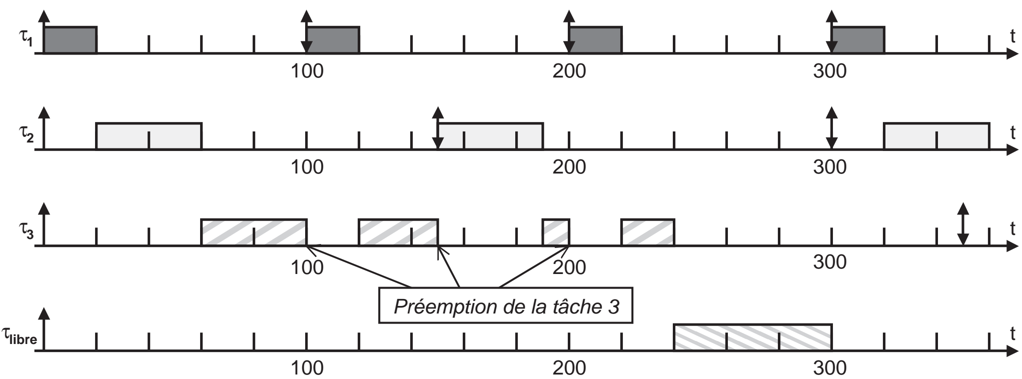


Figure 8.28 – Exemple d'une partie de la séquence d'exécution d'une configuration de trois tâches traitée avec l'algorithme RM.

Nous pouvons noter aussi la présence de temps libres du processeur qui sur l'ensemble de la période d'étude sont :

$$T_{\text{libre}} = (1 - 0{,}75238) \times 2100 = 520$$

© Dunod – La photocopie non autorisée est un délit.

La condition, exprimée par l'équation 8.27, est très restrictive (valeur minimale de **U**). Nous pouvons utiliser le théorème de la zone critique qui exprime le fait que si toutes les tâches sont à départ simultanée et si elles respectent leur première échéance alors la configuration est ordonnançable quel que soit l'instant d'arrivée des tâches dans la suite. Cette condition est nécessaire et suffisante si toutes les tâches sont à départ simultané et suffisante si les tâches sont à départ différé.
Un ensemble de **n** tâches $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$ ordonnées suivant les priorités (τ_1 la tâche la plus prioritaire et τ_n la tâche la moins prioritaire) définies par les paramètres temporels (r_i, C_i, D_i, T_i) est ordonnançable si et seulement si :

$$\forall i,\quad 1 \leq i \leq n \quad \min_{0 \leq t \leq D_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \leq 1 \qquad (8.28)$$

Cette condition correspond à tester intervalle de temps par intervalle de temps la possibilité d'exécuter les différentes tâches. La valeur $\lceil t/T_j \rceil$ correspond au nombre de réveils de la tâche τ_j dans l'intervalle de temps [0,t]. Le terme $C_j\lceil t/T_j \rceil$ représente la demande processeur de la tâche τ_j dans ce même intervalle. Le temps processeur demandé jusqu'à l'instant t par la tâche τ_j et toutes les tâches τ_j de plus forte priorité ($j \in [0,i-1]$) doit être inférieur à l'échéance d_i ($D_i = T_j$) de la tâche τ_i, soit :

$$\sum_{j=1}^{i-1} C_j \left| \frac{t}{T_j} \right| + C_i \leq D_i$$

Prenons l'exemple de la configuration à trois tâches décrite dans le tableau 8.7. Nous vérifions que nous sommes en présence d'une configuration de tâches indépendantes, périodiques, à échéance sur requête et à départ simultané. Les priorités ont été affectées selon l'algorithme d'ordonnancement RM. Le facteur d'utilisation **U** de cette configuration est donné par :

$$U = \sum_{i=1}^{3} C_i / T_i = \frac{1}{4} + \frac{2}{6} + \frac{2}{8} \approx 0{,}833 > 3\left(2^{\frac{1}{3}} - 1\right) \approx 0{,}779$$

La condition suffisante vue précédemment n'est pas satisfaite. Pour savoir si cette configuration est ordonnançable, il est donc nécessaire de procéder à une simulation complète sur la période d'étude (H = 24) ou d'utiliser la condition, dite de la zone critique exprimée par l'équation 8.28. Le calcul est effectué jusqu'au temps D_i à chaque nouvelle activation d'une tâche pour les trois tâches de la configuration. Ainsi, pour la première tâche avec $D_1 = 4$, nous avons :

$$\frac{C_1}{t}\left\lceil \frac{t}{T_1} \right\rceil \quad \text{pour } t = 4 \quad \frac{1}{4}\left\lceil \frac{4}{4} \right\rceil = \frac{1}{4} \quad \rightarrow \text{ minimum} \leq 1$$

Pour la deuxième tâche avec $D_2 = 6$, nous avons le minimum inférieur ou égal à 1 :

$$\frac{C_1}{t}\left\lceil\frac{t}{T_1}\right\rceil + \frac{C_2}{t}\left\lceil\frac{t}{T_2}\right\rceil \quad \text{pour } t = 4 \quad \frac{1}{4}\left\lceil\frac{4}{4}\right\rceil + \frac{2}{4}\left\lceil\frac{4}{6}\right\rceil = \frac{3}{4} \quad \rightarrow \leq 1$$

$$\text{pour } t = 6 \quad \frac{1}{6}\left\lceil\frac{6}{4}\right\rceil + \frac{2}{6}\left\lceil\frac{6}{6}\right\rceil = \frac{2}{3} \quad \rightarrow \geq 1$$

Pour la deuxième tâche avec $D_3 = 8$, nous avons le minimum inférieur ou égal à 1 :

$$\frac{C_1}{t}\left\lceil\frac{t}{T_1}\right\rceil + \frac{C_2}{t}\left\lceil\frac{t}{T_2}\right\rceil + \frac{C_3}{t}\left\lceil\frac{t}{T_3}\right\rceil \quad \text{pour } t = 4 \quad \frac{1}{4}\left\lceil\frac{4}{4}\right\rceil + \frac{2}{4}\left\lceil\frac{4}{6}\right\rceil + \frac{2}{4}\left\lceil\frac{4}{8}\right\rceil = \frac{5}{4} \rightarrow \; > 1$$

$$\text{pour } t = 6 \quad \frac{1}{6}\left\lceil\frac{6}{4}\right\rceil + \frac{2}{6}\left\lceil\frac{6}{6}\right\rceil + \frac{2}{6}\left\lceil\frac{6}{8}\right\rceil = 1 \rightarrow \; = 1$$

$$\text{pour } t = 8 \quad \frac{1}{8}\left\lceil\frac{8}{4}\right\rceil + \frac{2}{8}\left\lceil\frac{8}{6}\right\rceil + \frac{2}{8}\left\lceil\frac{8}{8}\right\rceil = 1 \rightarrow \; = 1$$

Nous pouvons donc conclure que la configuration est ordonnançable sans avoir à construire la séquence, représentée sur la figure 8.29, sur la période d'étude **H** = 24.

Tableau 8.7 – Exemple d'une configuration de trois tâches périodiques avec une affectation des priorités selon RM.

Tâche	r_i	C_i	D_i	T_i	Priorité selon RM
τ_1	0	1	4	4	**3**
τ_2	0	2	6	6	**2**
τ_3	0	2	8	8	**1**

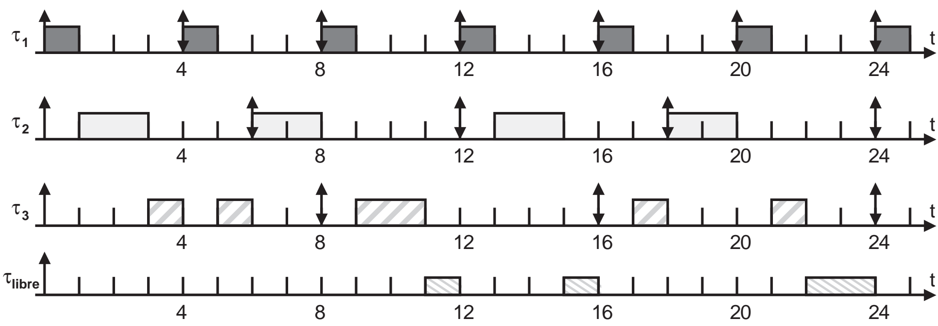


Figure 8.29 – Séquence d'exécution d'une configuration de trois tâches donnée dans le tableau 8.7 et traitée avec l'algorithme RM.

© Dunod – La photocopie non autorisée est un délit.

Algorithme d'ordonnancement « Deadline Monotonic »

S'il existe au moins une tâche qui n'est pas à échéance sur requête dans la configuration des tâches, alors nous allons utiliser un algorithme d'affectation des priorités basé sur les délais critiques des tâches au lieu des périodes. Cet algorithme d'ordonnancement, appelé « Deadline Monotonic » ou DM (ou Inverse Deadline ou ID), affecte la priorité la plus grande à la tâche dont le délai critique est le plus petit. Concernant cet algorithme d'ordonnancement pour une configuration de **n** tâches ayant les paramètres ($\mathbf{r_i}$, $\mathbf{C_i}$, $\mathbf{D_i}$, $\mathbf{T_i}$) avec $\mathbf{r_i} = 0$ pour tout i et au moins un délai critique $\mathbf{D_i}$ différent de la période $\mathbf{T_i}$, nous avons les résultats suivants :

– l'algorithme d'ordonnancement DM est **optimal** dans la classe des algorithmes à priorités fixes, c'est-à-dire que, si une configuration de tâches est ordonnançable, elle le sera en affectant les priorités selon DM ;

– une **condition suffisante d'ordonnançabilité** d'une configuration est obtenue pour un facteur de charge $\mathbf{U_l}$ du processeur suivant l'inégalité suivante :

$$U_l = \sum_{i=1}^{n} C_i / D_i \leq n\left(2^{\frac{1}{n}} - 1\right) \qquad (8.29)$$

Prenons l'exemple de la configuration à deux tâches décrite dans le tableau 8.8. Nous vérifions que nous sommes en présence d'une configuration de tâches indépendantes, périodiques et à départ simultané. Les priorités ont été affectées selon l'algorithme d'ordonnancement DM. Le facteur d'utilisation **U** et le facteur de charge $\mathbf{U_l}$ de cette configuration sont donnés par :

$$U = \sum_{i=1}^{2} C_i / T_i = \frac{1}{2} + \frac{1}{3} \approx 0{,}833 \leq 1$$

$$U_l = \sum_{i=1}^{2} C_i / D_i = \frac{1}{2} + \frac{1}{1} = 1{,}5 > n\left(2^{\frac{1}{n}} - 1\right) = 0{,}82$$

Par conséquent, nous ne pouvons pas en déduire que cette configuration de tâches est ordonnançable. Nous pouvons même remarquer que le facteur de charge est supérieur à 1, mais que le facteur d'utilisation reste inférieur à 1 (condition nécessaire pour toutes configurations de tâches s'exécutant sur un environnement monoprocesseur). Pour vérifier l'ordonnançabilité de cette configuration nous devons construire la partie de la séquence sur la période d'étude **H** = 6.
Cette séquence, représentée sur la figure 8.30, montre en particulier les zones d'exécution impossible qui se situent au-delà de l'échéance de la tâche $\boldsymbol{\tau_2}$.

Tableau 8.8 – Exemple d'une configuration de deux tâches périodiques avec une affectation des priorités selon DM.

Tâche	r_i	C_i	D_i	T_i	Priorité selon DM
τ_1	0	1	2	2	**1**
τ_2	0	1	1	3	**2**

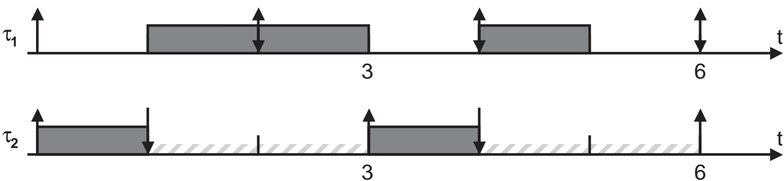


Figure 8.30 – Séquence d'exécution d'une configuration de deux tâches donnée dans le tableau 8.8 et traitée avec l'algorithme DM.

■ Comparaison des algorithmes d'ordonnancement « Rate Monotonic » et à tourniquet

Nous avons vu dans le chapitre 4 un ordonnancement classique de type tourniquet ou *Round Robin*. Les systèmes d'ordonnancement à priorité utilisent pour gérer des tâches dans une même file de priorité un ordonnancement de type « tourniquet », comme ce qui est énoncé dans la norme POSIX (chapitre 5). Dans le cadre de notre étude où la priorité est affectée en fonction des paramètres temporels de la tâche, il est intéressant de comparer l'ordonnancement « tourniquet » et l'ordonnancement à priorité fixe de type RM et leur couplage.

Sans approfondir une approche théorique, étudions deux exemples et comparons les ordonnancements dans les deux cas. Il est important de noter que l'ordonnancement « à tourniquet » est très dépendant des deux paramètres : gestion de la file d'attente et quantum d'affectation processeur. La file d'attente sera gérée selon une file de type FIFO. De plus nous supposons dans nos exemples que le quantum est fixé par le pas de temps discret d'exécution des tâches « 1 ». Remarquons aussi que, dans le cas de l'ordonnancement « à tourniquet », la notion de période d'étude n'est plus à considérer étant donné la gestion de la file d'attente asynchrone des périodes des tâches.

Prenons l'exemple de la configuration à deux tâches décrite dans le tableau 8.9. Nous vérifions que nous sommes en présence d'une configuration de tâches indépendantes, périodiques, à échéance sur requête et à départ simultané. Les priorités ont été affectées selon l'algorithme d'ordonnancement RM et le facteur d'utilisation est **U** = 0,94. La file d'attente du tourniquet est ordonnée au départ avec la tâche $\boldsymbol{\tau_1}$ en premier et la tâche $\boldsymbol{\tau_2}$ en second. Dans ces conditions, le tracé de la séquence

© Dunod – La photocopie non autorisée est un délit.

d'exécution montre que la séquence d'exécution, basée sur l'ordonnancement RM, ne permet pas de respecter les échéances (tâche τ_2 échoue).

Tableau 8.9 – Exemple d'une configuration de deux tâches périodiques avec une affectation des priorités selon RM : comparaison des ordonnancements « à tourniquet » et RM.

Tâche	r_i	C_i	D_i	T_i	Priorité selon RM
τ_1	0	3	6	6	**2**
τ_2	0	4	9	9	**1**

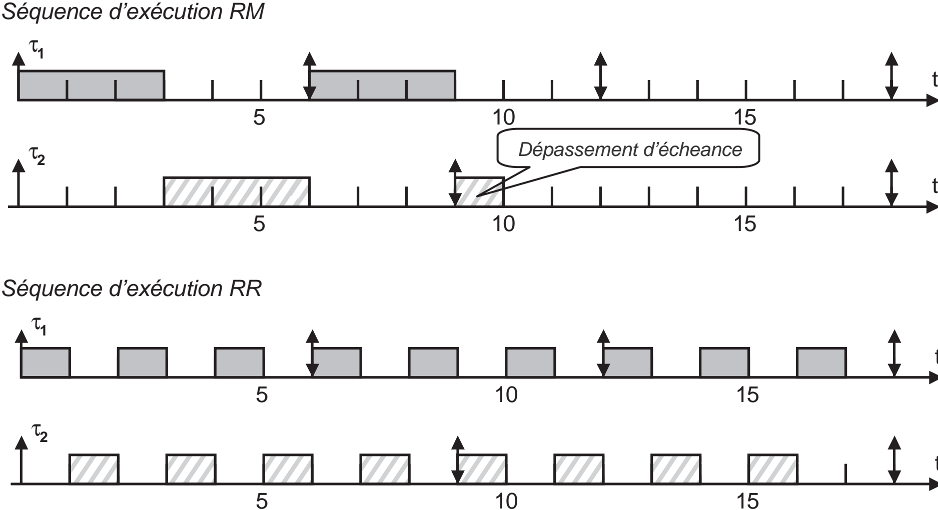


Figure 8.31 – Séquence d'exécution d'une configuration de deux tâches donnée dans le tableau 8.9 et traitée avec les deux algorithmes RM et « à tourniquet ».

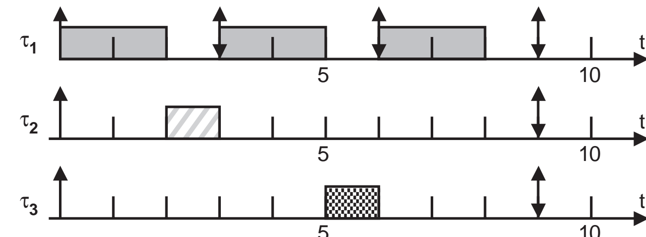
De cet exemple, nous pouvons conclure que l'algorithme basé sur le tourniquet permet d'ordonnancer plus de configurations que l'algorithme RM. Mais, considérons l'exemple de la configuration à trois tâches décrite dans le tableau 8.10. Nous vérifions que nous sommes en présence d'une configuration de tâches indépendantes, périodiques, à échéance sur requête et à départ simultané. Les priorités ont été affectées selon l'algorithme d'ordonnancement RM avec une liberté de choix pour les tâches τ_2 et τ_3. Le facteur d'utilisation est $\mathbf{U} = 0{,}88$. La file d'attente du tourniquet est composée au départ des tâches τ_1 en premier, ensuite τ_2 et enfin τ_3 en dernier. Dans ces conditions, le tracé de la séquence d'exécution montre que la séquence d'exécution, basée sur l'ordonnancement « à tourniquet », ne permet pas de respecter les échéances (tâche τ_1 échoue).

À partir de ce deuxième exemple, nous pouvons donc conclure que l'algorithme RM permet d'ordonnancer plus de configurations que l'algorithme basé sur le tourniquet. En réalité de nombreuses applications peuvent être ordonnancées à la fois par l'algorithme RM et l'algorithme « à tourniquet ».
Dans le cadre d'une utilisation couplée des deux algorithmes d'ordonnancement, tous les résultats, obtenus dans le cas de l'algorithme RM (conditions d'ordonnançabilité, période d'étude…), peuvent s'appliquer lorsque les tâches ont une priorité affectée selon RM et pour une même priorité (même période : cas des tâches $\boldsymbol{\tau_2}$ et $\boldsymbol{\tau_3}$ de la configuration du tableau 8.10) un ordonnancement à tourniquet.

Tableau 8.10 – Exemple d'une configuration de trois tâches périodiques avec une affectation des priorités selon RM : comparaison des ordonnancements « à tourniquet » et RM.

Tâche	r_i	C_i	D_i	T_i	Priorité selon RM
τ_1	0	2	3	3	**3**
τ_2	0	1	9	9	**2**
τ_3	0	1	9	9	**1**

Séquence d'exécution RM



Séquence d'exécution RR

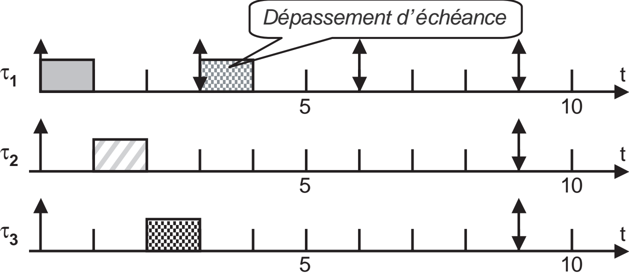


Figure 8.32 – Séquence d'exécution d'une configuration de deux tâches donnée dans le tableau 8.10 et traitée avec les deux algorithmes RM et « à tourniquet ».

© Dunod – La photocopie non autorisée est un délit.

8.3.2 Algorithmes d'ordonnancement à priorités variables

Nous avons jusqu'à présent considéré que la priorité affectée à une tâche restait constante pendant toute la durée de l'application. Nous allons nous intéresser à une autre catégorie d'algorithme d'ordonnancement pour laquelle la priorité des tâches varie au cours de l'exécution d'une tâche. Cette priorité est fonction d'une caractéristique temporelle dynamique de la tâche.

■ Algorithme d'ordonnancement « Earliest Deadline First »

Dans le cas de l'algorithme « Earliest Deadline First » ou EDF, la priorité des tâches est variable au cours de leur exécution et fonction de la prochaine échéance. Pour une instance **k** d'une tâche $\boldsymbol{\tau_i}$, la priorité est liée à la prochaine échéance $\mathbf{d_{i,k}}$ de cette tâche. À un instant t, la priorité peut être calculée à partir du délai critique dynamique $\mathbf{D_i(t)}$ qui s'exprime sous la forme :

$$D_i(t) = d_{i,k} - t = r_{i,k} + D_i - t$$

Nous pouvons faire les deux remarques suivantes :

– la priorité est variable, elle change au cours de l'exécution ;

– la priorité augmente si le délai critique dynamique de la tâche diminue.

Considérons l'exemple de la configuration à trois tâches décrite dans le tableau 8.11. Nous vérifions que nous sommes en présence d'une configuration de tâches indépendantes, périodiques, à échéance sur requête et à départ simultané. Les priorités initiales ont été affectées en fonction du délai critique qui correspond à la première échéance. Le facteur d'utilisation de la configuration est **U** = 0,983 et la période d'étude est **H** = 60. Par conséquent le temps libre est de 1. Cette configuration n'est pas ordonnançable par l'algorithme à priorité fixe RM comme le montre le diagramme de Gantt de la figure 8.33. Par conséquent, étant donné la propriété d'optimalité de l'algorithme RM, cette configuration n'est ordonnançable par aucun algorithme à priorité fixe. En revanche, la figure 8.34 présente la séquence d'exécution de cette configuration sur une partie de la période d'étude. L'évolution du délai critique dynamique, qui conditionne la priorité des tâches, est notée sur chaque séquence.

Comme pour les algorithmes à priorité fixe, nous disposons d'une condition d'ordonnançabilité pour l'algorithme EDF. Pour une configuration de tâches indépendantes, périodiques, à échéance sur requête et à départ simultané, la condition nécessaire et suffisante d'ordonnançabilité est exprimée par :

$$U = \sum_{i=1}^{n} C_i / T_i \leq 1 \qquad (8.30)$$

L'expression 8.30 montre la puissance d'ordonnançabilité de l'algorithme EDF puisque le processeur peut être utilisé à 100 % et la configuration validée formellement. L'algorithme EDF est optimal dans la catégorie des algorithmes à priorité variable.

Tableau 8.11 – Exemple d'une configuration de trois tâches périodiques pour l'étude de l'ordonnancement EDF.

Tâche	r_i	C_i	D_i	T_i	Priorité selon D_i
τ_1	0	1	3	3	**3**
τ_2	0	1	4	4	**2**
τ_3	0	2	5	5	**1**

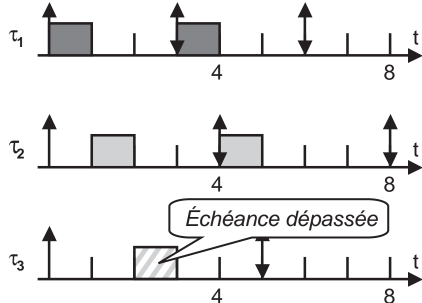


Figure 8.33 – Séquence d'exécution d'une configuration de trois tâches donnée dans le tableau 8.11 et traitée avec l'algorithme RM : non ordonnançable.

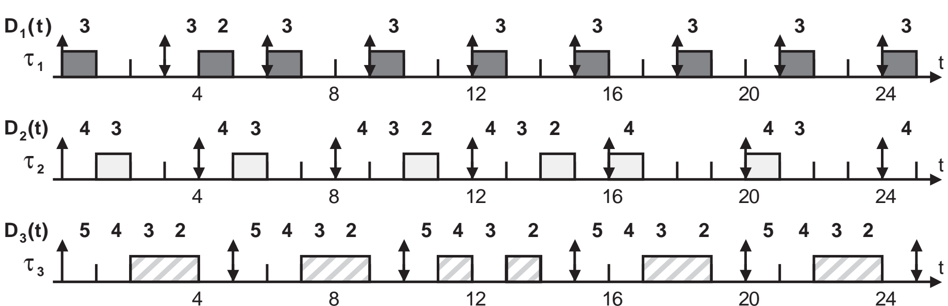


Figure 8.34 – Séquence d'exécution d'une configuration de trois tâches donnée dans le tableau 8.11 et traitée avec l'algorithme EDF : ordonnançable.

Pour tester l'ordonnançabilité d'une configuration de tâches dont au moins une n'est pas à échéance sur requête, il est possible d'utiliser une analyse basée sur l'occupation du processeur. Pour l'algorithme EDF, cette condition nécessaire et suffisante d'ordonnançabilité est la suivante : « Pour une longueur de séquence **H'** correspondant à la plus petite séquence d'exécution totalement occupée, dite période d'activité (**U** = 1), nous devons avoir la relation suivante » :

© Dunod – La photocopie non autorisée est un délit.

$$\forall h' \in H', \quad h' \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{h' - D_i}{T_j} \right\rfloor + 1 \right) \cdot C_i \qquad (8.31)$$

Il est très important de noter qu'une configuration de tâches dont certaines ne sont pas à échéance sur requête peut ne pas être ordonnançable par EDF même si le facteur d'utilisation du processeur est inférieur à 1.

Considérons l'exemple de la configuration à trois tâches décrite dans le tableau 8.12. Nous vérifions que nous sommes en présence d'une configuration de tâches indépendantes, périodiques et à départ simultané. Les priorités initiales ont été affectées en fonction du délai critique qui correspond à la première échéance. Le facteur d'utilisation de la configuration est **U** = 1 et la période d'étude est **H** = 12. En revanche, le facteur de charge est de $\mathbf{U_1}$ = 1,233. La figure 8.35 montre que la configuration n'est pas ordonnançable par EDF.

Tableau 8.12 – Exemple d'une configuration de trois tâches périodiques pour l'étude de l'ordonnancement EDF avec des tâches qui ne sont pas à échéance sur requête.

Tâche	r_i	C_i	D_i	T_i	Priorité selon D_i
τ_1	0	1	2	3	**3**
τ_2	0	1	3	4	**2**
τ_3	0	2	5	6	**1**

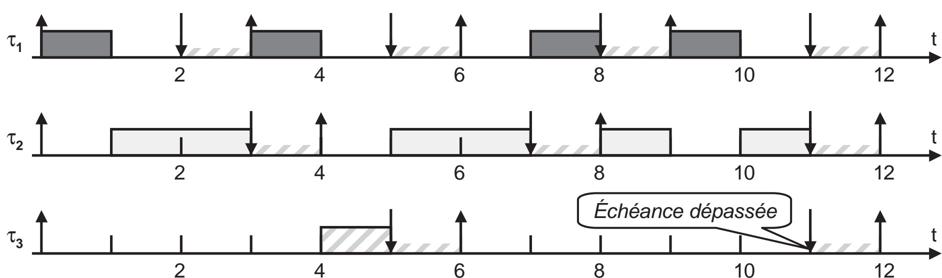


Figure 8.35 – Séquence d'exécution d'une configuration de trois tâches donnée dans le tableau 8.12 et traitée avec l'algorithme EDF : non ordonnançable.

Il est aisée de s'attendre à ce que la séquence échoue puisque le facteur d'utilisation de la configuration était 100 % et le dernier temps horloge de la séquence était inutilisable (hors échéances). De même ce résultat de « non ordonnançabilité » pouvait être prouvé à partir de l'équation 8.30 qui s'écrit dans ce cas sous la forme :

pour $h' = 11$,

$$[(11-2)/3+1] \cdot 1 + [(11-3)/4+1] \cdot 2 + [(11-5)/6+1] \cdot 1 = 12$$

Algorithme d'ordonnancement « Minimum Laxity »

Dans le cas de l'algorithme « Minimum Laxity » ou ML (ou « Least Laxity » LL), la priorité des tâches est variable au cours de leurs exécutions et fonction de la laxité dynamique. Pour une instance **k** d'une tâche $\boldsymbol{\tau_i}$, la priorité est liée à la laxité dynamique $\mathbf{L_{i,k}(t)}$ de cette tâche. À un instant t, la priorité peut être calculée à partir de :

$$L_i(t) = d_{i,k} - C_i(t) - t = r_{i,k} + D_i - C_i(t) - t$$
$$\text{ou} \quad L_i(t) = d_{i,k} - C_i - t = r_{i,k} + D_i - C_i - t \qquad (8.32)$$

Il est important de noter que les deux expressions 8.32 donnent le même résultat au niveau de l'ordonnancement. Cet algorithme ML a les mêmes caractéristiques que l'ordonnancement EFD : optimalité et ordonnançabilité.
Pour illustrer cet algorithme ML et le comparer à l'ensemble des algorithmes que nous avons vu, étudions un exemple de configuration à trois tâches décrite dans le tableau 8.13. Nous vérifions que nous sommes en présence d'une configuration de tâches indépendantes, périodiques et à départ simultané. Le facteur d'utilisation de la configuration est **U** = 0,93 et la période d'étude est **H** = 12. Les priorités initiales ont été reportées dans le tableau 8.14. Dans le cas des algorithmes à priorités variables (EDF et ML), les priorités sont des priorités initiales déterminées selon le délai critique et la laxité.

Tableau 8.13 – Exemple d'une configuration de trois tâches périodiques indépendantes.

Tâche	r_i	C_i	D_i	T_i
τ_1	0	1	3	3
τ_2	0	1	4	4
τ_3	0	2	3	6

Tableau 8.14 – Affectation des priorités selon les algorithmes RM, DM, EDF et ML.

Tâche	RM	DM	EDF	ML
τ_1	3	3	3	2
τ_2	2	1	1	1
τ_3	1	2	2	3

Étant donné que les tâches ne sont pas à échéance sur requête, il est inutile d'utiliser l'algorithme RM ; il est ici étudié à titre d'exemple. La figure 8.36 montre les quatre séquences d'exécution obtenues avec les quatre algorithmes. La configuration n'est pas ordonnançable avec les algorithmes à priorité fixe RM et DM. En revanche, les séquences tracées avec EDF et ML sont valides.

© Dunod – La photocopie non autorisée est un délit.

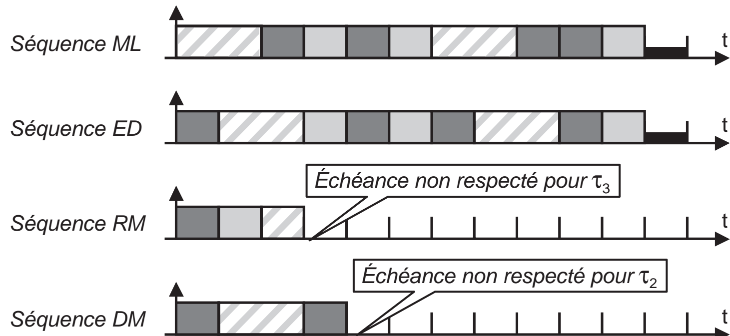


Figure 8.36 – Séquence d'exécution d'une configuration de trois tâches donnée dans le tableau 8.12 et traitée avec les quatre algorithmes RM, DM, EDF et ML.

Si la capacité d'ordonnançabilité des deux algorithmes à priorité variable EDF et ML est identique, les séquences obtenues dans les deux cas peuvent être très différentes et en particulier en ce qui concerne les changements de contexte. Ainsi, l'algorithme ML génère de façon intempestive des changements de contexte inutiles en augmentant ainsi le temps système. Prenons l'exemple de deux tâches périodiques, indépendantes, à échéance sur requête et à départ simultané (tableau 8.15).

Tableau 8.15 – Exemple d'une configuration de deux tâches périodiques indépendantes.

Tâche	r_i	C_i	D_i	T_i
τ_1	0	5	10	10
τ_2	0	4	9	9

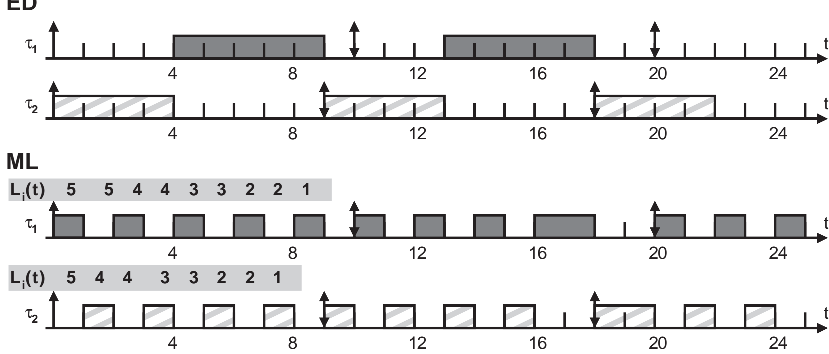


Figure 8.37 – Comparaison des séquences d'exécution d'une configuration de deux tâches donnée dans le tableau 8.15 et traitée avec les deux algorithmes EDF et ML.

Le facteur d'utilisation de la configuration est **U** = 0,94 et la période d'étude est **H** = 90. Cette configuration est ordonnançable par les deux algorithmes, mais les deux séquences produites sont très différentes. En effet, comme nous pouvons le voir sur la figure 8.37, la séquence obtenue par l'algorithme ML contient plus de 5 fois plus de changements de contexte que celle construite avec l'algorithme EDF.

Cet inconvénient de l'algorithme ML conduit à préférer l'algorithme EDF en environnement monoprocesseur. En revanche, dans un environnement multiprocesseur, nous étudierons l'efficacité de l'algorithme ML.

■ Analyse graphique de l'ordonnançabilité d'une configuration de tâches indépendantes

Dans la représentation graphique des paramètres possibles d'une tâche périodique présentée sur la figure 8.4 et la limitation due aux contraintes temporelles, exprimées dans le cahier des charges de l'application, illustrée sur la figure 8.14, nous pouvons ajouter le fait que la tâche s'exécute dans un environnement multitâche et donc restreindre encore la zone graphique des paramètres autorisés pour une tâche dans une configuration ordonnançable. Cette représentation graphique d'une tâche périodique offre alors un domaine de choix plus limité pour le concepteur (figure 8.38). Il est important de noter que $\mathbf{T_U}$ correspond à la valeur minimale de la période **T** calculée selon la condition nécessaire **U** = 1.
Prenons l'exemple de trois tâches périodiques, indépendantes et à départ simultané (tableau 8.16). Pour les deux paramètres de la tâche $\boldsymbol{\tau_3}$ à la valeur maximale, le facteur d'utilisation de la configuration est **U** = 0,75. En, considérant la condition nécessaire du facteur d'utilisation du processeur à 100 %, la période minimale de la tâche $\boldsymbol{\tau_3}$ est $\mathbf{T_u}$ = 6. L'algorithme d'ordonnancement utilisé est DM. Dans cette configuration, le concepteur peut choisir parmi les 11 points possibles au niveau des deux paramètres **D** et **T** de la tâche $\boldsymbol{\tau_3}$.

Tableau 8.16 – Exemple d'une configuration de trois tâches périodiques indépendantes, la troisième tâche ayant des paramètres à préciser.

Tâche	r_i	C_i	D_i	T_i
τ_1	0	2	12	12
τ_2	0	2	4	8
τ_3	0	3	≤ 7	≤ 9

© Dunod – La photocopie non autorisée est un délit.

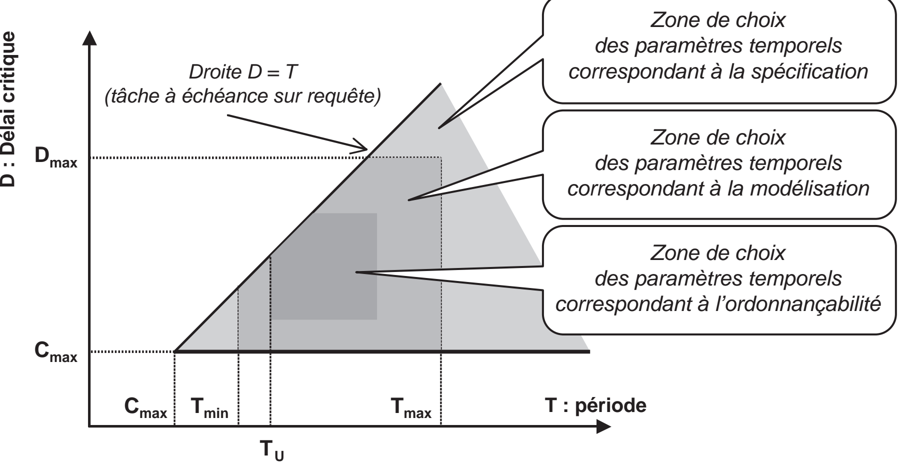


Figure 8.38 – Représentation graphique du choix des paramètres conditionnée par l'ordonnançabilité de la configuration.

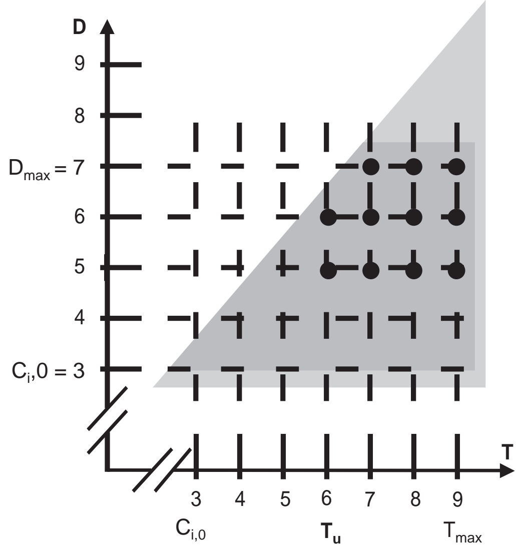


Figure 8.39 – Représentation graphique d'une tâche périodique avec la zone de choix possible des paramètres de la tâche τ_3 en conservant l'ordonnançabilité de la configuration décrite dans le tableau 8.16.

8.4 Ordonnancement des tâches indépendantes apériodiques

L'ordonnancement des tâches apériodiques est traité dans le cas des tâches périodiques à contraintes strictes et des tâches apériodiques à contraintes relatives ou à contraintes strictes (tâches sporadiques). Nous supposons que l'ordonnancement des tâches périodiques se fait suivant les algorithmes RM, DM ou EDF. Pour répondre à la demande d'événements déclenchant des tâches apériodiques, nous pouvons considérer trois méthodes :

- **Traitement en arrière plan** : les tâches apériodiques sont traitées pendant les temps d'oisiveté du processeur.
- **Traitement par utilisation d'un serveur périodique des tâches apériodiques en environnement à priorité fixe** pour les tâches périodiques (RM) : en plus de toutes les tâches périodiques, on insère une tâche, appelée serveur, qui est destinée à traiter les tâches apériodiques et qui possède différentes caractéristiques selon le modèle du serveur :
  - serveur à scrutation,
  - serveur ajournable,
  - serveur à échange de priorité,
  - serveur sporadique,
  - serveur à vol de temps creux,
  - serveur à échange de priorité étendu.
- **Traitement par utilisation d'un serveur périodique des tâches apériodiques en environnement à priorité variable** pour les tâches périodiques (EDF) : en plus de toutes les tâches périodiques, on insère une tâche, appelée serveur, qui est destinée à traiter les tâches apériodiques et qui possède différentes caractéristiques :
  - serveur dynamique à échange de priorité,
  - serveur dynamique sporadique,
  - serveur à largeur de bande maximale,
  - serveur Earliest Deadline Last (EDL),
  - serveur à échange de priorité amélioré.

L'objet de cet ouvrage n'est pas de faire une description exhaustive de l'ensemble de ces techniques de prise en compte des tâches apériodiques, mais de donner les principes généraux de ces différents traitements en choisissant dans chacune des catégories les plus représentatifs.

8.4.1 Traitement en arrière plan des tâches apériodiques à contraintes relatives

Les tâches périodiques étant ordonnancées par un algorithme à priorité fixe comme RM ou DM (ou éventuellement à priorité variable comme EDF), les tâches apériodiques sont traitées pendant les temps d'oisiveté du processeur, méthode appelée aussi « vol de temps creux ». L'ordonnançabilité de la configuration n'est nullement remise en cause puisque les tâches apériodiques sont stockées dans une file d'attente gérée en FIFO ou par priorité et traitée dans le temps libre processeur (figure 8.40).

© Dunod – La photocopie non autorisée est un délit.

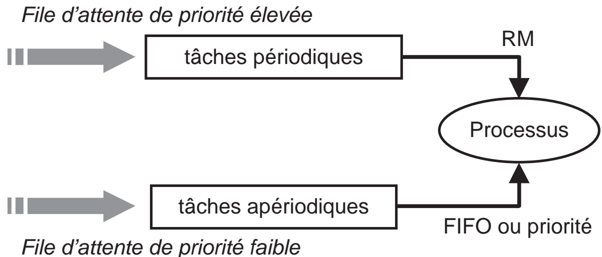


Figure 8.40 – Ordonnancement des tâches périodiques et apériodiques dans un traitement en arrière plan.

Cette méthode ne permet pas de traiter les tâches apériodiques strictes car le temps de réponse de ces tâches traitées en arrière plan n'est pas borné. Aussi, cette méthode fonctionne correctement si le processeur a un taux de charge pas trop élevé.

Prenons l'exemple de deux tâches périodiques, indépendantes, à échéance sur requête et à départ simultané (tableau 8.17). Le facteur d'utilisation de la configuration est **U** = 0,75, valeur inférieure à la condition d'ordonnançabilité pour l'algorithme d'ordonnancement RM (équation 8.27). Nous allons étudier le traitement de trois tâches apériodiques à contraintes relatives dont les paramètres $\mathbf{r_i}$ et $\mathbf{C_i}$ sont donnés dans le tableau 8.17. L'exécution de ces tâches s'effectue dans les temps libres laissés par les deux autres tâches comme le montre la figure 8.41.

Tableau 8.17 – Exemple d'une configuration de deux tâches périodiques indépendantes à ordonnancer avec l'arrivée de trois tâches apériodiques à contraintes relatives.

Tâche	r_i	C_i	D_i	T_i
τ_1	0	1	4	4
τ_2	0	2	6	6
τ_3	0	1		
τ_4	4	4		
τ_5	13	1		

Nous obtenons des temps de réponse qui sont fonction de la charge processeur due aux tâches périodiques, des dates d'arrivée des tâches apériodiques et des durées des traitements de ces tâches apériodiques.

En connaissant la charge du système par les tâches périodiques, il est possible d'évaluer le temps de réponse d'une tâche apériodique en supposant connues sa date d'arrivée et sa durée d'exécution. Ainsi, par définition les $\mathbf{r_i}$ des tâches apériodiques n'étant pas connus, il est nécessaire de faire cette évaluation pour toute la séquence d'exécution

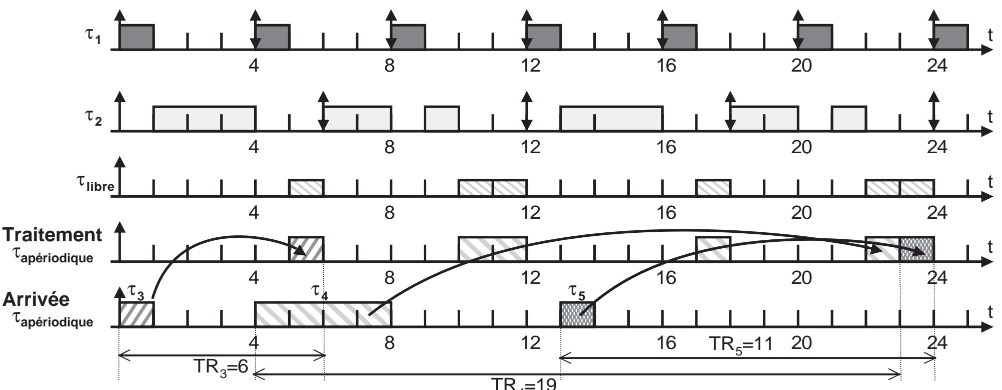


Figure 8.41 – Exemple d'ordonnancement des tâches périodiques et apériodiques dans un traitement en arrière plan pour la configuration donnée dans le tableau 8.17.

de la configuration (période d'étude **H**). Nous pouvons donc conclure que cette méthode de prise en compte des tâches apériodiques ne peut concerner que les tâches à contraintes non strictes, c'est-à-dire acceptant un temps de réponse non borné.

8.4.2 Traitement par serveur périodique des tâches apériodiques en environnement à priorité fixe

Afin de pouvoir limiter le temps de réponse associé aux requêtes apériodiques, une tâche périodique spécifique, appelé **serveur**, va être dédiée aux traitements de ces requêtes apériodiques. Ce serveur ou ces serveurs s'ajoutent à la configuration de base des tâches périodiques ; il est évident que cet ajout ne doit pas remettre en cause l'ordonnançabilité de la configuration qui est supposée en absence de tâches apériodiques. Ainsi, ces serveurs traitent les tâches apériodiques au moment où ils possèdent le processeur. Nous allons considérer qu'une tâche apériodique $\boldsymbol{\tau_{ap,i}}$ à contrainte stricte est caractérisée par les trois paramètres suivants :

– $\mathbf{C_i}$: durée d'exécution maximale ;
– $\mathbf{D_i}$: délai critique, c'est-à-dire le délai au bout duquel la tâche doit être terminée par rapport à la date de réveil ;
– $\boldsymbol{\Delta_{min}}$: distance minimum entre deux occurrences successives de la requête apériodique.

Un ou plusieurs serveurs périodiques à scrutation sont affectés à une ou plusieurs requêtes apériodiques. La définition des paramètres temporels de ces serveurs dépend d'une part des besoins de l'application et d'autre part des caractéristiques du serveur en termes de paramètres temporels et mode d'exécution (priorité, conservation de sa capacité de traitement…).

© Dunod – La photocopie non autorisée est un délit.

■ Serveur à scrutation

Le premier type de serveur le plus simple est le **serveur à scrutation**. Ce serveur à scrutation est une tâche périodique $\boldsymbol{\tau_s}$ possédant les paramètres temporels classiques ($\mathbf{r_s}$, $\mathbf{C_s}$, $\mathbf{D_s}$, $\mathbf{T_s}$). Ce serveur traite les tâches apériodiques qu'il trouve en attente lors de son activation ; s'il n'y a pas de tâches en attente, le serveur se suspend et attend la prochaine période d'activation. Le point crucial est donc de déterminer les paramètres du serveur par rapport aux paramètres temporels de la requête apériodique stricte à servir.

Nous allons nous placer dans le cas où un serveur est affecté à une requête apériodique stricte donnée ; par conséquent il est naturel de fixer la durée d'exécution du serveur égale à la durée de la tâche apériodique $\mathbf{C_s} = \mathbf{C_{ap}}$. Dans le cas où nous avons requête apériodique stricte, caractérisée par une échéance $\mathbf{D_{ap}}$ et une distance minimum entre deux occurrences successives de $\Delta_{\mathbf{min}}$, les caractéristiques du serveur ($\mathbf{D_s}$,$\mathbf{T_s}$) peuvent être élaborées dans le cas le plus défavorable où l'activation du serveur a été demandée mais annulée car l'événement n'était pas présent à l'instant de l'activation et l'occurrence effective de l'événement se produit immédiatement après avec un décalage très faible $\Delta t = \varepsilon$ (figure 8.42). Dans ce cas extrême pour satisfaire les contraintes temporelles attachées à cette requête, nous devons résoudre l'inéquation suivante :

$$T_s + D_s \leq D_{ap} \leq \Delta_{\min} \qquad (8.33)$$

Si nous considérons le serveur comme une tâche périodique à échéance sur requête ($\mathbf{D_s} = \mathbf{T_s}$), nous avons la relation simplifiée de l'équation 8.33 :

$$T_s \leq \frac{D_{ap}}{2} \qquad (8.34)$$

Il est important de noter que cette définition des paramètres temporels du serveur à scrutation permet de borner le temps de réponse du service de la tâche apériodique quel que soit sa date d'occurrence dans la séquence d'exécution. Le temps de réponse est donc égal à :

$$TR_{ap} \leq T_s + D_s \leq D_{ap} \quad \text{ou} \quad TR_{ap} \leq 2\,T_s \leq D_{ap} \qquad (8.35)$$

Dans ce cadre la définition des paramètres du serveur en fonction des paramètres de tâche apériodique stricte est donnée dans le tableau 8.18. De la même façon que pour les tâches périodiques (figure 8.38), nous pouvons représenter graphiquement le domaine des valeurs possibles pour ce serveur en fonction des paramètres de la tâche apériodique (figure 8.43). Dans cette représentation, il est possible de lire directement sur le graphique le temps de réponse maximum du serveur.

Prenons l'exemple de deux tâches périodiques, indépendantes et à départ simultané (tableau 8.19). En considérant une troisième tâche $\boldsymbol{\tau_s}$, serveur à scrutation, avec la valeur maximale de sa période égale à 12, le facteur d'utilisation de la configuration est $\mathbf{U} = 0{,}7$. Avec un facteur d'utilisation du processeur à 100%, la période minimale de la tâche $\boldsymbol{\tau_s}$ est $\mathbf{T} = 6$. Nous allons tester les deux algorithmes d'ordonnancement DM et EDF sur cette configuration composée de deux tâches périodiques et

Tableau 8.18 – Schéma d'adaptation d'un serveur à scrutation pour répondre à une requête apériodique.

	Tâche apériodique		Serveur à scrutation
r_i	?	→	0
C_i	C_{ap}	→	$C_s=C_{ap}$
D_i	D_{ap}	↘	D_s
T_i	(Δ_{min})		$T_s + D_s \leq D_{ap}$

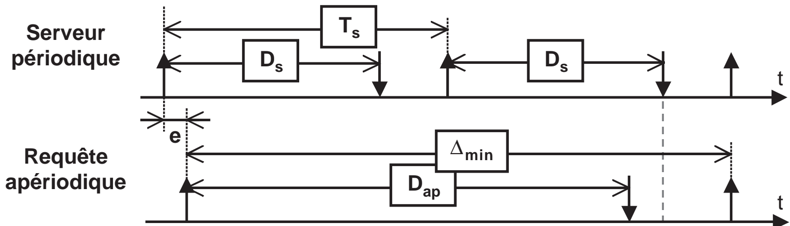


Figure 8.42 – Analyse temporelle de l'adaptation d'un serveur à scrutation pour répondre à une requête apériodique.

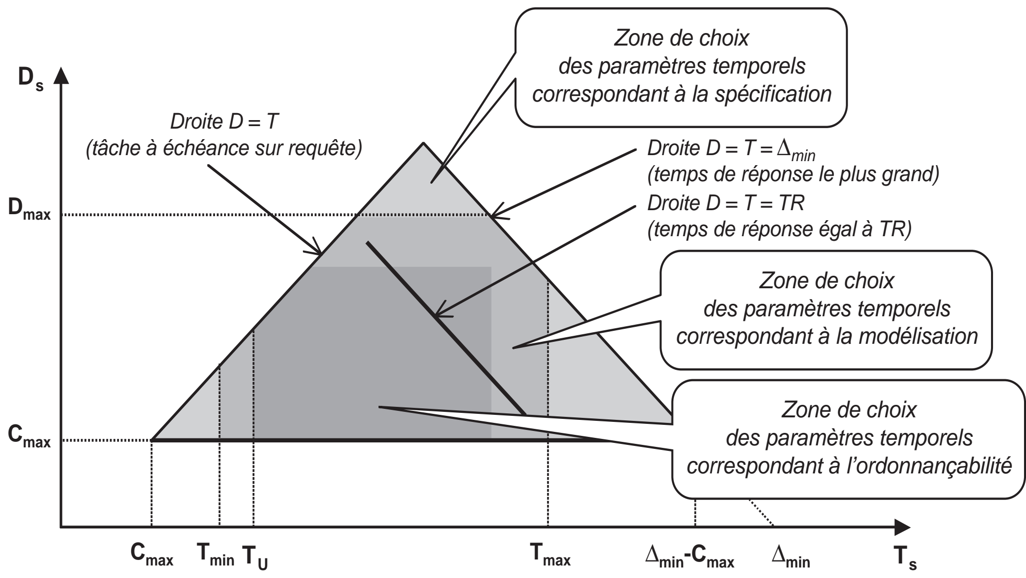


Figure 8.43 – Représentation graphique du choix des paramètres pour un serveur à scrutation correspondant à une requête apériodique.

© Dunod – La photocopie non autorisée est un délit.

Tableau 8.19 – Exemple d'une configuration de trois tâches périodiques indépendantes, la troisième tâche, serveur à scrutation de tâches apériodiques, ayant des paramètres à préciser.

Tâche	r_i	C_i	D_i	T_i
τ_1	0	2	8	10
τ_2	0	2	8	8
τ_3	0	3	–	≤ 12

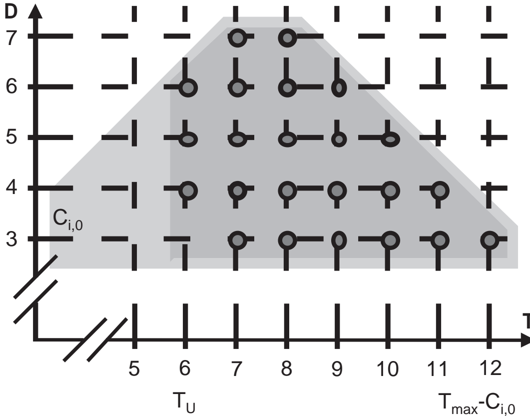


Figure 8.44 – Représentation graphique du choix des paramètres pour un serveur à scrutation correspondant à une requête apériodique stricte : configuration ordonnancée avec les priorités affectées selon l'algorithme DM.

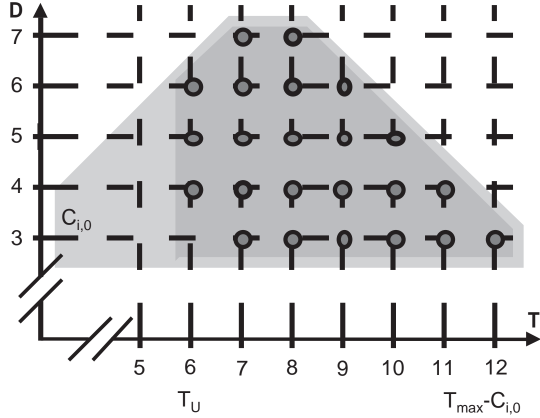


Figure 8.45 – Représentation graphique du choix des paramètres pour un serveur à scrutation correspondant à une requête apériodique : configuration ordonnancée par l'algorithme EDF.

d'un serveur périodique. Les résultats au niveau des deux paramètres **D** et **T** de la tâche $\boldsymbol{\tau_s}$, présentés respectivement sur les figures 8.44 et 8.45, montrent que le concepteur peut choisir parmi 20 points possibles pour l'ordonnancement DM et 23 points pour l'ordonnancement EDF.

De même que dans le cas des tâches périodiques gérées selon l'algorithme Rate Monotonic, il est possible d'avoir une autre condition suffisante d'ordonnançabilité. Soit un ensemble de **n** tâches périodiques T $\{\tau_1, \tau_2, \tau_3, \ldots, \tau_i, \ldots, \tau_n\}$ définies par les paramètres temporels ($\mathbf{r_i}, \mathbf{C_i}, \mathbf{D_i}, \mathbf{T_i}$) et un ensemble de **p** serveurs périodiques $\{\tau_{s1}, \tau_{s2}, \tau_{s3}, \ldots, \tau_{si}, \ldots, \tau_{sn}\}$ définies par les paramètres temporels ($\mathbf{r_{si}}, \mathbf{C_{si}}, \mathbf{D_{si}}, \mathbf{T_{si}}$), la configuration est ordonnançable si :

$$U_{\text{total}} = U + U_s = \sum_{i=1}^{n} C_i / T_i + \sum_{j=1}^{p} C_{ap,j} / T_{ap,j} \leq (n+p)\left(2^{\frac{1}{(n+p)}} - 1\right) \quad (8.36)$$

Considérons l'exemple de deux tâches périodiques, indépendantes, à échéance sur requête et à départ simultané (tableau 8.20). Une troisième tâche périodique est ajoutée à la configuration pour traiter les arrivées de tâches apériodiques. La capacité de traitement des requêtes apériodiques de ce serveur est : une durée d'exécution de 1 ($\mathbf{C_s} = 1$) et un temps de réponse maximum de 10 ($2\mathbf{T_s}$). Le facteur d'utilisation de la configuration complète est **U** = 1, valeur supérieure à la condition d'ordonnançabilité pour l'algorithme d'ordonnancement RM (équation 8.27). La période d'étude étant **H** = 20, il suffit de tester l'ordonnançabilité de la configuration sur cette durée. Il est aisé de vérifier que la configuration de ces trois tâches périodiques est ordonnançable avec l'algorithme RM.

Tableau 8.20 – Exemple d'une configuration de trois tâches périodiques indépendantes, la troisième tâche, serveur à scrutation de tâches apériodiques.

Tâche	r_i	C_i	D_i	T_i
τ_1	0	4	10	10
τ_2	0	8	20	20
τ_s	0	1	5	5
$\tau_{ap,1}$	5	1	10	
$\tau_{ap,2}$	12	2	15	

Étant donné que la troisième tâche est un serveur à scrutation de tâches apériodiques, celle-ci ne s'exécute pas à chaque réveil si aucune requête de traitement apériodique n'est arrivée. La figure 8.46 représente la séquence d'exécution avec l'arrivée de deux requêtes apériodiques strictes $\boldsymbol{\tau_{ap,1}}$ (durée $\mathbf{C_{ap,1}} = 1$) et $\boldsymbol{\tau_{ap,2}}$ (durée $\mathbf{C_{ap,2}} = 2$) aux instants respectifs $\mathbf{r_1} = 5$ et $\mathbf{r_2} = 12$. La première tâche apériodique stricte a une

© Dunod – La photocopie non autorisée est un délit.

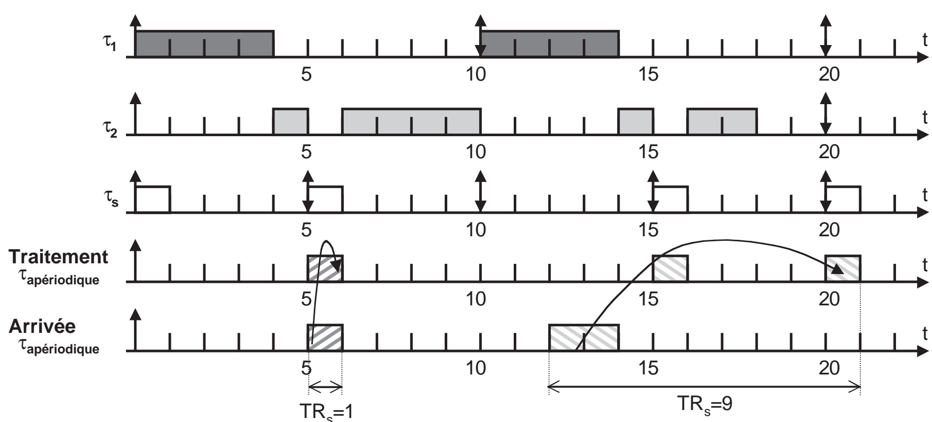


Figure 8.46 – Exemple de l'exécution de la configuration de tâches donnée dans le tableau 8.20 : ordonnancement avec l'algorithme RM.

échéance de 10 par rapport à son réveil, ce qui explique les paramètres du serveur ($2\mathbf{T_s} = 10 \leq \mathbf{D_{ap,1}}$). La deuxième tâche apériodique stricte a une échéance de 15 étant donné que son temps d'exécution est le double de celui du serveur ; il sera nécessaire d'avoir deux exécutions du serveur pour traiter complètement cette requête ($2\mathbf{T_s}+\mathbf{T_s} = 15 \leq \mathbf{D_{ap,2}}$).

Le temps de réponse du traitement de ces deux requêtes apériodiques strictes, respectivement 1 et 9, montre que les bornes maximales sont respectées. En revanche, dans le cas de la deuxième tâche apériodique stricte, le réveil inutile du serveur au temps 10, puis l'arrivée de cette requête apériodique stricte au temps 12 conduit à allonger le temps de réponse.

■ Serveur ajournable

Par rapport à la méthode précédente, la méthode du **serveur ajournable** autorise le serveur à **conserver sa capacité de traitement pendant toute la période**, c'est-à-dire qu'une tâche apériodique peut arriver pendant cette durée et être servie. En début de période, la charge du serveur est remise dans son état initial quelle que soit la capacité utilisée. En donnant généralement une priorité forte au serveur (tâche de période faible), cela permet de **réduire le temps de réponse moyen**. Ainsi, l'exemple de la figure 8.47 montre le principe du serveur ajournable (évolution de sa capacité de traitement en fonction du temps) et son application sur l'arrivée de requête apériodique stricte. Pour un serveur de caractéristiques temporelles (0, $\mathbf{C_s} = 2$, $\mathbf{D_s} = 4$, $\mathbf{T_s} = 4$), seule la dernière tâche apériodique n'est pas traitée immédiatement. Par rapport au serveur à scrutation, le temps de réponse à la requête apériodique est réduit à son minimum puisque le serveur peut prendre en compte cette tâche apériodique à tout instant de l'exécution. La seule condition requise est que la capacité du serveur soit en adéquation avec la durée d'exécution de la tâche apériodique.

Considérons l'exemple de deux tâches périodiques, indépendantes, à échéance sur requête et à départ simultané (tableau 8.21). Une troisième tâche périodique est

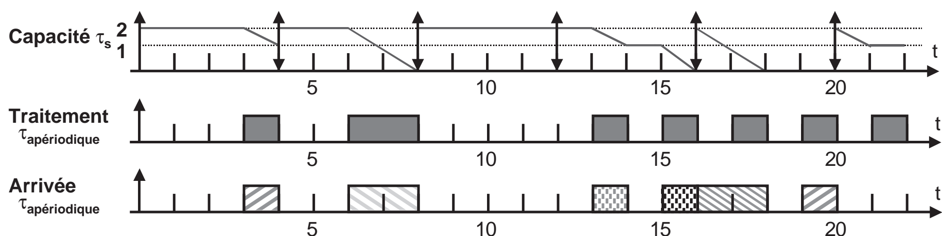


Figure 8.47 – Représentation du principe de fonctionnement du serveur ajournable avec la visualisation de sa capacité de traitement.

ajoutée à la configuration pour traiter les arrivées de tâches apériodiques strictes. La capacité de traitement des requêtes apériodiques strictes de ce serveur ajournable est : une durée d'exécution de 2 ($\mathbf{C_s} = 2$). Le facteur d'utilisation de la configuration complète est $\mathbf{U} = 0{,}87$, valeur supérieure à la condition d'ordonnançabilité pour l'algorithme d'ordonnancement RM (équation 8.27). La période d'étude étant $\mathbf{H} = 168$, il est nécessaire de tester l'ordonnançabilité de la configuration sur cette durée. Il est aisé de vérifier que la configuration de ces trois tâches périodiques est ordonnançable avec l'algorithme RM. La figure 8.48 présente une partie seulement de cette séquence d'exécution selon un ordonnancement RM. Dans ce test d'ordonnançabilité, il est important de remarquer que le serveur s'exécute à chaque réveil et dès l'instant de son réveil étant donné sa priorité supérieure aux deux autres tâches.

Tableau 8.21 – Exemple d'une configuration de trois tâches périodiques indépendantes, la troisième tâche, serveur ajournable de tâches apériodiques.

Tâche	r_i	C_i	D_i	T_i
τ_1	0	2	7	7
τ_2	0	8	8	8
τ_S	0	2	6	6

Pour analyser la réponse de ce serveur ajournable à des requêtes apériodiques strictes, nous considérons trois demandes d'exécution de tâches apériodiques strictes définies par : $\boldsymbol{\tau_{ap,1}}$ ($\mathbf{r_{ap,1}} = 1$, $\mathbf{C_{ap,1}} = 2$), $\boldsymbol{\tau_{ap,2}}$ ($\mathbf{r_{ap,2}} = 12$, $\mathbf{C_{ap,2}} = 2$) et $\boldsymbol{\tau_{ap,3}}$ ($\mathbf{r_{ap,3}} = 18$, $\mathbf{C_{ap,3}} = 1$). La figure 8.49 montre l'acceptation immédiate de ces requêtes apériodiques strictes par le serveur ajournable avec un temps de réponse minimum égal aux durées de traitements des tâches apériodiques strictes. Le serveur s'exécute donc à des instants non prévisibles de sa période.

En conséquence, le serveur ayant une priorité forte et conservant sa capacité d'exécution au cours de sa période, il y a contradiction avec l'algorithme d'ordonnancement : « une tâche de priorité forte activable doit s'exécuter ». Ce problème peut conduire à des dépassements d'échéance. Ainsi, considérons trois requêtes d'exécution de

© Dunod – La photocopie non autorisée est un délit.

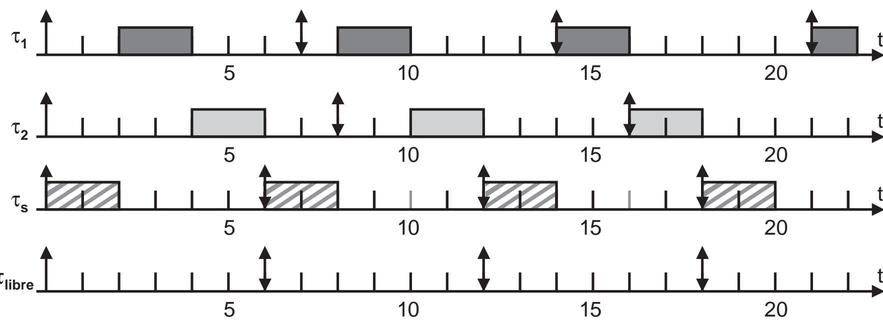


Figure 8.48 – Exemple de l'exécution de la configuration de tâches
donnée dans le tableau 8.21 : ordonnancement avec l'algorithme RM.

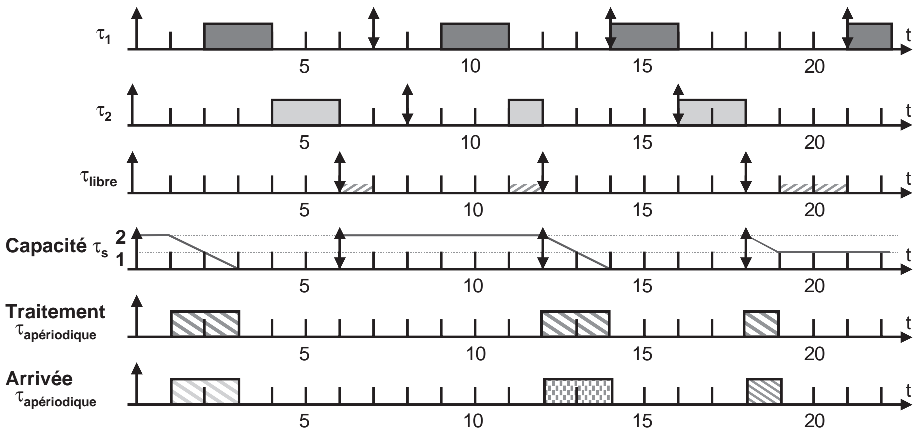


Figure 8.49 – Exemple de l'exécution valide de la configuration de tâches
donnée dans le tableau 8.21 avec l'arrivée de trois requêtes apériodiques strictes strictes.

tâches apériodiques strictes définies par : $\boldsymbol{\tau}_{\mathbf{ap,1}}$ ($\mathbf{r}_{\mathbf{ap,1}}$ = 1, $\mathbf{C}_{\mathbf{ap,1}}$ = 2), $\boldsymbol{\tau}_{\mathbf{ap,2}}$ ($\mathbf{r}_{\mathbf{ap,2}}$ = 7, $\mathbf{C}_{\mathbf{ap,2}}$ = 2) et $\boldsymbol{\tau}_{\mathbf{ap,3}}$ ($\mathbf{r}_{\mathbf{ap,3}}$ = 12, $\mathbf{C}_{\mathbf{ap,3}}$ = 2). La figure 8.50 montre l'acceptation immédiate de ces requêtes apériodiques par le serveur ajournable. Cette acceptation à n'importe quel instant de la séquence d'exécution conduit la tâche périodique $\boldsymbol{\tau}_{\mathbf{2}}$ à dépasser son échéance.

De même que dans le cas des tâches périodiques gérées selon l'algorithme Rate Monotonic, il est possible d'avoir une autre condition suffisante d'ordonnançabilité dans le cas de l'algorithme RM associé au serveur ajournable. Soit un ensemble de **n** tâches périodiques {$\boldsymbol{\tau}_{\mathbf{1}}$, $\boldsymbol{\tau}_{\mathbf{2}}$, $\boldsymbol{\tau}_{\mathbf{3}}$, …, $\boldsymbol{\tau}_{\mathbf{i}}$, …, $\boldsymbol{\tau}_{\mathbf{n}}$} définies par les paramètres temporels ($\mathbf{r}_{\mathbf{i}}$, $\mathbf{C}_{\mathbf{i}}$, $\mathbf{D}_{\mathbf{i}}$, $\mathbf{T}_{\mathbf{i}}$) et un serveur périodique $\boldsymbol{\tau}_{\mathbf{s}}$ définie par les paramètres temporels ($\mathbf{r}_{\mathbf{s}}$, $\mathbf{C}_{\mathbf{s}}$, $\mathbf{D}_{\mathbf{s}}$, $\mathbf{T}_{\mathbf{s}}$) avec $\mathbf{U}_{\mathbf{s}} = \mathbf{C}_{\mathbf{s}}/\mathbf{T}_{\mathbf{s}}$, la configuration est ordonnançable si :

$$U = \sum_{i=1}^{n} C_i / T_i \leq \ln\left[\frac{U_s + 2}{2\,U_s + 1}\right] \quad \text{avec} \quad U_s = \frac{C_s}{T_s} \qquad (8.37)$$

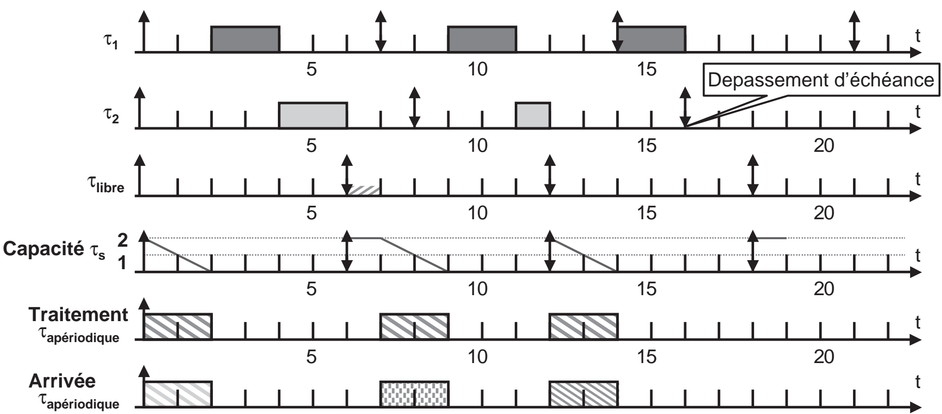


Figure 8.50 – Exemple de l'exécution non valide de la configuration de tâches donnée dans le tableau 8.21 avec l'arrivée de trois requêtes apériodiques strictes.

Donc le facteur de charge total est donné par :

$$U_{\text{total}} \leq U + U_s = \sum_{i=1}^{n} C_i / T_i + \ln\left[\frac{U_s + 2}{2\,U_s + 1}\right] \quad \text{avec} \quad U_s = \frac{C_s}{T_s} \qquad (8.38)$$

Serveur sporadique

Pour améliorer les performances du serveur précédent, serveur ajournable**,** il faut diminuer cette possibilité totale à conserver la capacité pendant toute la période, ce problème pouvant conduire à des dépassements d'échéance pour les tâches périodiques. Deux méthodes peuvent être utilisées pour cela :

– Le serveur à échange de priorité : le serveur ne conserve pas sa forte priorité pendant toute sa séquence. Mais si aucune requête apériodique ne survient, il prend la priorité de la dernière tâche en exécution. Cette diminution de priorité conduit à préserver l'exécution de tâches périodiques moins prioritaires initialement.

– Le **serveur sporadique** : la méthode est basée sur le fait que sa capacité ne se recharge qu'à l'instant de sa consommation augmenté de sa période.

Ainsi, l'exemple de la figure 8.51 montre le principe du serveur sporadique (évolution de sa capacité de traitement en fonction du temps) et son application sur l'arrivée de requête apériodique stricte. Pour un serveur de caractéristiques temporelles (0, $\mathbf{C_s} = 2$, $\mathbf{D_s} = 4$, $\mathbf{T_s} = 4$), seules les deux dernières tâches apériodiques stricte**s** n'ont pas été traitées immédiatement, résultat meilleur que le serveur à scrutation, mais moins bon que le serveur ajournable.

Comparons le fonctionnement de ce serveur sporadique avec le serveur ajournable pour le même exemple décrit dans le tableau 8.21 et la figure 8.50 intégrant trois requêtes d'exécution de tâches apériodiques strictes définies par : $\boldsymbol{\tau_{ap,1}}$ ($\mathbf{r_{ap,1}} = 1$, $\mathbf{C_{ap,1}} = 2$), $\boldsymbol{\tau_{ap,2}}$ ($\mathbf{r_{ap,2}} = 7$, $\mathbf{C_{ap,2}} = 2$) et $\boldsymbol{\tau_{ap,3}}$ ($\mathbf{r_{ap,3}} = 12$, $\mathbf{C_{ap,3}} = 2$). La séquence d'exécution, présentée sur la figure 8.52, est valide en comparaison de celle obtenue

© Dunod – La photocopie non autorisée est un délit.

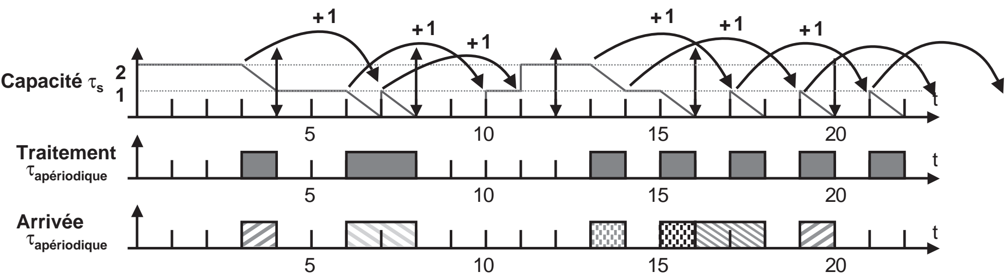


Figure 8.51 – Représentation du principe de fonctionnement du serveur sporadique avec la visualisation de sa capacité de traitement.

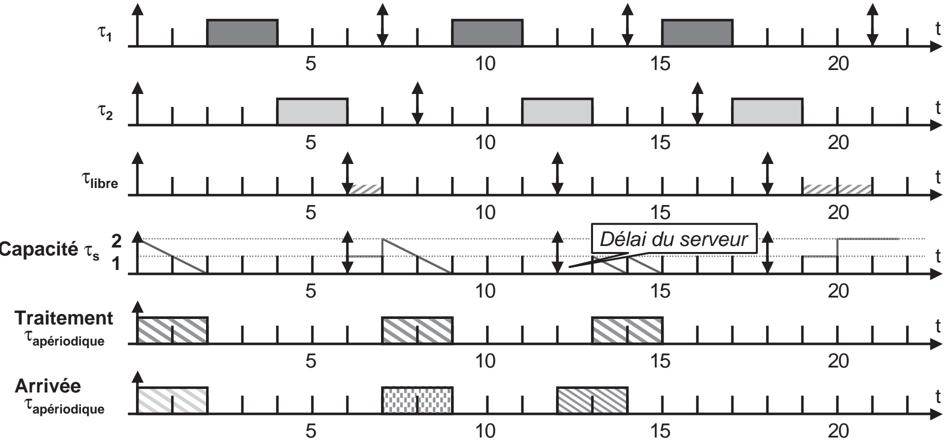


Figure 8.52 – Exemple de l'exécution valide de la configuration de tâches donnée dans le tableau 8.21 avec l'arrivée de trois requêtes apériodiques strictes.

avec le serveur ajournable. Cette ordonnançabilité est obtenue grâce au décalage du serveur et conduit donc à un temps de réponse augmenté de la troisième tâche apériodique stricte.

Il est possible d'avoir une autre condition suffisante d'ordonnançabilité dans le cas de l'algorithme RM associé au serveur ajournable. Soit un ensemble de **n** tâches périodiques $\{\tau_1, \tau_2, \tau_3, \ldots, \tau_i, \ldots, \tau_n\}$ définies par les paramètres temporels ($\mathbf{r_i}$, $\mathbf{C_i}$, $\mathbf{D_i}$, $\mathbf{T_i}$) et un serveur périodique τ_s définie par les paramètres temporels ($\mathbf{r_s}$, $\mathbf{C_s}$, $\mathbf{D_s}$, $\mathbf{T_s}$) avec $\mathbf{U_s} = \mathbf{C_s/T_s}$, la configuration est ordonnançable si :

$$U = \sum_{i=1}^{n} C_i / T_i \leq \ln\left[\frac{2}{U_s + 1}\right] \quad \text{avec} \quad U_s = \frac{C_s}{T_s} \qquad (8.39)$$

Donc, le facteur de charge total est donné par :

$$U_{\text{total}} \leq U + U_s = \sum_{i=1}^{n} C_i / T_i + \ln\left[\frac{2}{U_s + 1}\right] \quad \text{avec} \quad U_s = \frac{C_s}{T_s} \qquad (8.40)$$

Il est intéressant de comparer cette limite du serveur sporadique (équation 8.38) et du serveur ajournable (équation 8.40). La figure 8.53 montre que la courbe associée au serveur ajournable passe en dessous de la limite de l'algorithme RM, alors que la courbe associée au serveur sporadique reste au-dessus de cette limite.

En conclusion, dans le contexte d'algorithme à priorité fixe (RM ou DM), le serveur sporadique permet de répondre aux requêtes apériodiques et de traiter la tâche apériodique associée avec un temps de réponse minimum en conservant l'ordonnançabilité de la configuration initiale, serveur inclus.

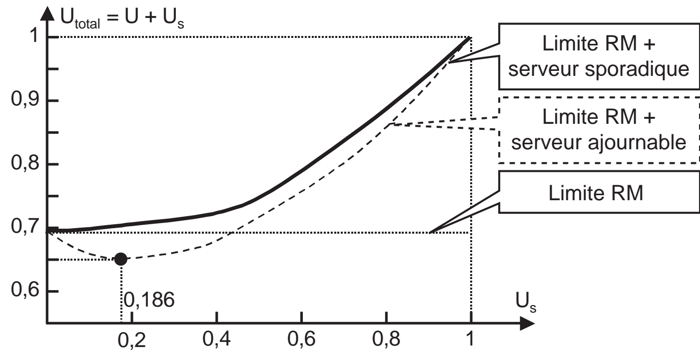


Figure 8.53 – Exemple de l'exécution valide de la configuration de tâches donnée dans le tableau 8.21 avec l'arrivée de trois requêtes apériodiques strictes.

8.4.3 Traitement par serveur périodique des tâches apériodiques en environnement à priorité variable

■ Serveur sporadique dynamique

Dans un contexte d'ordonnancement basé sur une priorité variable (EDF), la méthode du serveur sporadique peut-être reprise. Contrairement au fonctionnement du serveur sporadique avec des priorités fixes, le contexte d'attribution dynamique des priorités conduit à affecter au serveur une priorité variable fonction de la date future de réapprovisionnement de sa capacité. Ce fonctionnement est illustré sur la figure 8.54. Pour un serveur de caractéristiques temporelles (0, $\mathbf{C_s} = 2$, $\mathbf{D_s} = 4$, $\mathbf{T_s} = 4$), nous pouvons remarquer que son échéance se place toujours à un délai temporel de 4 (= $\mathbf{T_s}$) unités de temps par rapport à la consommation de la capacité du serveur par une tâche apériodique stricte.

Considérons l'exemple de deux tâches périodiques, indépendantes, à échéance sur requête et à départ simultané (tableau 8.22). Une troisième tâche périodique est

© Dunod – La photocopie non autorisée est un délit.

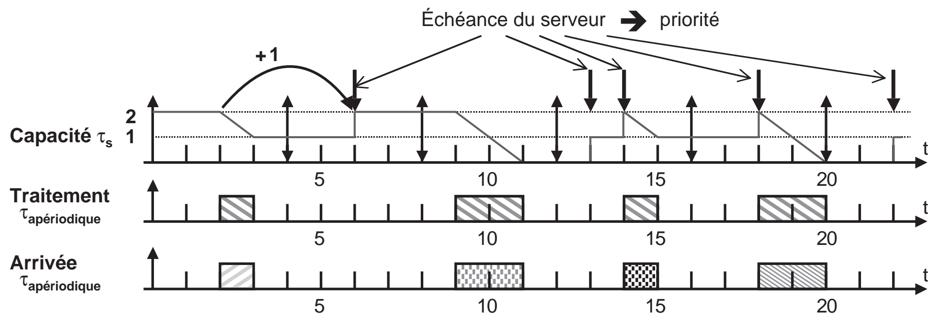


Figure 8.54 – Représentation du principe de fonctionnement du serveur sporadique dynamique avec la visualisation de sa capacité de traitement.

ajoutée à la configuration pour traiter les arrivées de tâches apériodiques strictes. La capacité de traitement des requêtes apériodiques de ce serveur est une durée d'exécution de 3 ($\mathbf{C_s} = 3$). Le facteur d'utilisation de la configuration complète est $\mathbf{U} = 1$, égale à la valeur de la condition d'ordonnançabilité pour l'algorithme d'ordonnancement EDF (équation 8.30). La configuration est donc ordonnançable avec l'algorithme EDF. Nous ajoutons à cette configuration des tâches apériodiques dont les caractéristiques sont référencées dans le tableau 8.22.

Tableau 8.22 – Exemple d'une configuration de trois tâches périodiques indépendantes, la troisième tâche, serveur sporadique dynamique de tâches apériodiques. Les arrivées des requêtes et les durées des tâches apériodiques sont notées.

Tâche	r_i	C_i	D_i	T_i
τ_1	0	2	8	8
τ_2	0	3	12	12
τ_s	0	3	6	6
$\tau_{ap,1}$	3	2		
$\tau_{ap,2}$	6	1		
$\tau_{ap,3}$	9	1		
$\tau_{ap,4}$	14	2		
$\tau_{ap,5}$	16	1		

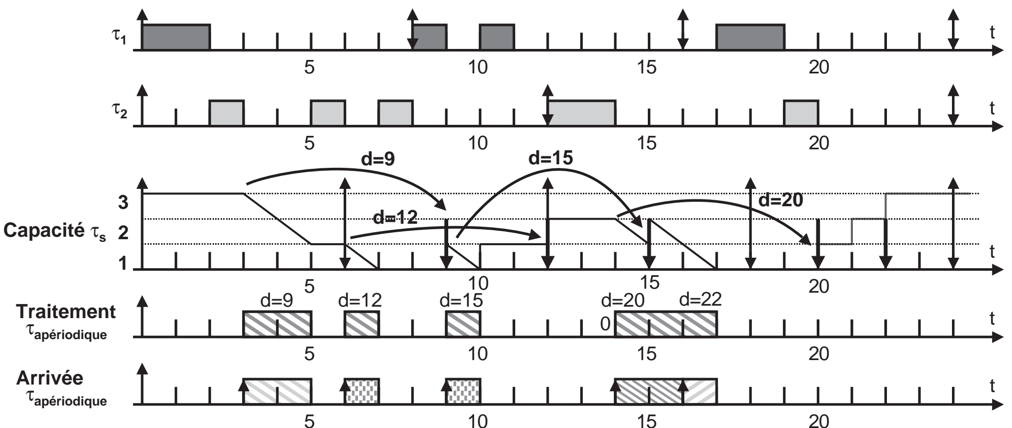


Figure 8.55 – Exemple de l'exécution valide de la configuration de tâches donnée dans le tableau 8.21 avec l'arrivée de trois requêtes apériodiques strictes.

La séquence obtenue avec le traitement par le serveur de type sporadique dynamique est donnée sur la figure 8.55. L'acceptation des tâches apériodiques est immédiate et les temps de réponse sont égaux aux durées d'exécution. Le fonctionnement du serveur sporadique dynamique, basé sur une recharge de capacité avec un décalage correspondant à la période, comme pour le serveur sporadique, conduit à une séquence valide.

■ Utilisation de l'algorithme Earliest Deadline Last (EDL)

L'ordonnancement, basé sur une priorité variable Earliest Deadline First (EDF), est réalisé au plus tôt (si une tâche est activable, elle est exécutée). De fait, les temps creux processeur se retrouvent en fin de séquence d'exécution. Si nous voulons utiliser ces temps creux pour exécuter les requêtes apériodiques, il serait souhaitable de les placer au moment de la demande, c'est-à-dire à l'instant de la requête. Ainsi, la méthode du traitement par EDL consiste à exécuter une séquence selon EDF ; puis, lorsque survient une requête apériodique, on l'exécute dans les temps creux disponibles de la séquence créée par Earliest Deadline Last (EDL). Cette caractéristique est visualisée sur la figure 8.56 où une même configuration de deux tâches, $\boldsymbol{\tau_1}$ (0,1,3,3) et $\boldsymbol{\tau_2}$ (0,2,5,5), est ordonnancée par EDF et EDL.

Pour étudier ce moyen de traitement des requêtes apériodiques, considérons une configuration à deux tâches, $\boldsymbol{\tau_1}$ (0,3,6,6) et $\boldsymbol{\tau_2}$ (0,2,8,8). Cette configuration de facteur d'utilisation $\mathbf{U} = 0{,}75$ et de période d'étude $\mathbf{H} = 24$ est ordonnancée avec les algorithmes EDF et EDL à partir du temps t = 8 (figure 8.57). Cela démontre bien qu'il est alors possible d'insérer des temps libres au milieu de la séquence.

Ces temps libres au nombre de 6 peuvent être utilisés pour répondre à des requêtes apériodiques. Un exemple est donné pour une tâche apériodique définie par $\boldsymbol{\tau_{ap}}$ ($\mathbf{r_{ap,3}} = 8$, $\mathbf{C_{ap,3}} = 4$). Jusqu'à cette demande apériodique, la séquence est celle obtenue avec l'algorithme EDF. Ensuite le maximum de temps libres est mis en place et la séquence reprend avec l'algorithme EDL.

© Dunod – La photocopie non autorisée est un délit.

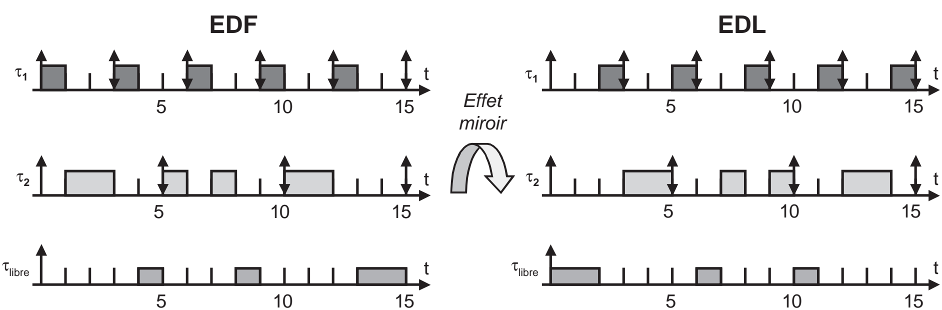


Figure 8.56 – Exemple de l'exécution valide d'une même configuration de deux tâches avec les algorithmes EDF et EDL.

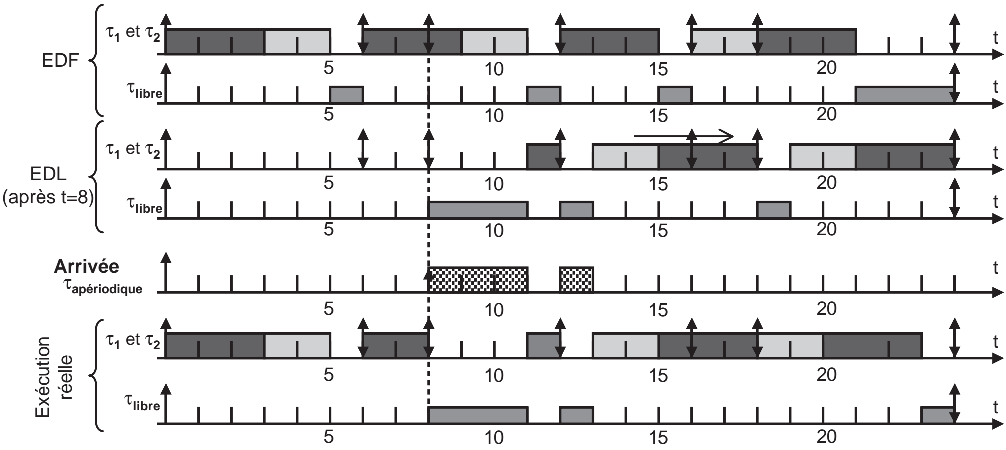


Figure 8.57 – Exemple de l'exécution d'une configuration de deux tâches avec la prise en compte d'une tâche apériodique en utilisant l'association des algorithmes EDF et EDL.

8.4.4 Conclusion sur le traitement des tâches apériodiques

Nous avons vu les principales méthodes pour répondre aux requêtes apériodiques. L'efficacité en termes de temps de réponse à ces tâches apériodiques et la complexité du traitement associé à la méthode conduisent à faire un compromis. Une comparaison entre les différentes méthodes est montrée sur la figure 8.58. Cette comparaison est réalisée sur les bases de quatre critères :

– performance basée sur le temps de réponse moyen obtenu ;
– complexité du calcul liée à la prise de décision au niveau de l'ordonnanceur ;
– besoin mémoire conditionné par une anticipation de la séquence d'ordonnancement ;
– complexité de l'implémentation relative à la mise en œuvre de cette méthode.

Dans le cas des environnements à priorité fixe (cas industriel classique), le serveur sporadique semble être un bon compromis au niveau de ces paramètres.

	Technique de traitement	Performance	Complexité calcul	Performance	Complexité implémentation
Priorité fixe	Traitement en arrière plan	↘	↗	↗	↗
	Serveur à scrutation	↘	↗	↗	↗
	Serveur ajournable	→	↗	↗	↗
	Serveur sporadique	→	→	→	→
Priorité variable	Serveur sporadique dynamique	→	→	→	→
	Serveur EDL	↗	↘	↘	↘

↗ excellent → moyen ↘ mauvais

Figure 8.58 – Comparaison entre les méthodes de prise en compte d'une tâche apériodique.

8.5 Ordonnancement des tâches périodiques dépendantes

8.5.1 Ordonnancement des tâches avec contraintes de précédence

■ Définition générale de la précédence

Des tâches peuvent être liées par des contraintes de précédence lorsqu'elles ont des relations de **synchronisation** (sémaphores, événements) ou de **communication** (boîtes aux lettres). On appelle une contrainte de précédence entre la tâche τ_i et la tâche τ_j le cas où τ_i précède τ_j si τ_j doit attendre la fin d'exécution de τ_i pour commencer sa propre exécution (figure 8.59). Nous supposons dans cette section que les tâches ont une forme atomique telle que nous l'avons décrite dans la section 8.2.3, c'est-à-dire : attente de synchronisation ou de communication en début de tâche et envoi d'un événement de synchronisation ou de communication en fin de tâche. L'expression des contraintes de précédence (ordre partiel sur l'ensemble des tâches) peut se faire par exemple sous la forme d'un graphe comme celui de la figure 8.60. Ainsi, les six tâches de la configuration (τ_1, τ_2, τ_3, τ_4, τ_5, τ_6) sont liées par des relations de précédences décrites par deux graphes.

Nous pouvons remarquer que nous avons jusqu'à présent noté seulement les précédences dites simples. Pour être complet, il est nécessaire de distinguer les deux cas de précédence :

– **Contraintes de précédence simple** : une contrainte de précédence entre la tâche $\boldsymbol{\tau_i}$ et la tâche $\boldsymbol{\tau_j}$ ou $\boldsymbol{\tau_i}$ précède $\boldsymbol{\tau_j}$ si $\boldsymbol{\tau_j}$ doit attendre la fin d'exécution de $\boldsymbol{\tau_i}$ pour commencer sa propre exécution. Dans ce cas, nous faisons l'hypothèse suivante :

© Dunod – La photocopie non autorisée est un délit.

« Deux tâches périodiques liées par une contrainte de précédence simple sont de même période. »

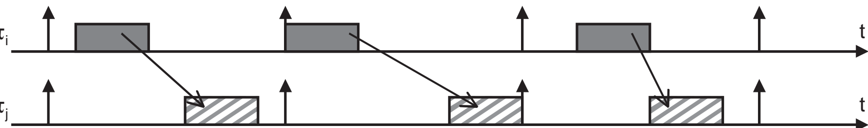


Figure 8.59 – Visualisation de la contrainte de précédence entre deux tâches.

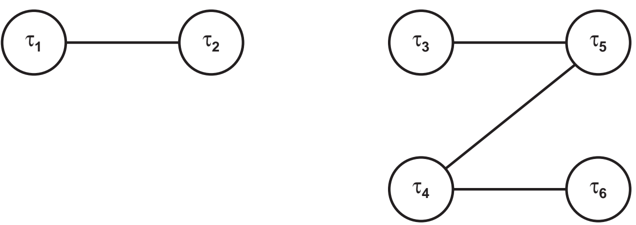


Figure 8.60 – Graphe de représentation des contraintes de précédence entre les tâches.

– **Contraintes de précédence généralisée** : le nombre des exécutions de deux tâches liées par une telle relation n'est pas nécessairement le même. C'est le cas lorsque $\boldsymbol{\tau_i}$ peut s'exécuter **n** fois avant l'exécution de $\boldsymbol{\tau_j}$, ou lorsque $\boldsymbol{\tau_i}$ s'exécute une seule fois avant **n** exécutions de la tâche $\boldsymbol{\tau_j}$ (figure 8.61).

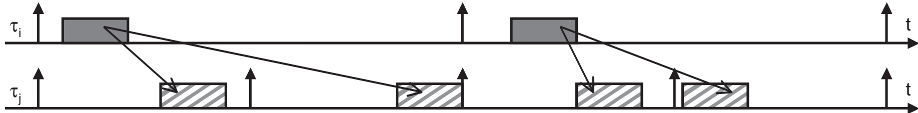


Figure 8.61 – Visualisation de la contrainte de précédence généralisée entre deux tâches.

Des exemples de ces deux cas de précédence généralisée sont présentés sur la figure 8.62. Pour le premier exemple, la tâche « Mesure de température » doit s'exécuter quatre fois avant l'exécution de la tâche qui calcule une moyenne sur quatre points mesurés. Pour le second cas, la tâche « Mesure de la pression P » s'exécute à la même cadence que la tâche « Calcul de la fonction P/T » ; par contre la tâche « Mesure de température T » n'a pas besoin de s'exécuter aussi rapidement étant donné la variation lente de ce paramètre physique.

Pour pouvoir utiliser les algorithmes d'ordonnancement précédemment étudiés, il est nécessaire d'obtenir une configuration de tâches **indépendantes**. Par conséquent, les caractéristiques des tâches avec contraintes de précédence sont transformées afin de prendre en compte implicitement cette relation entre les tâches.

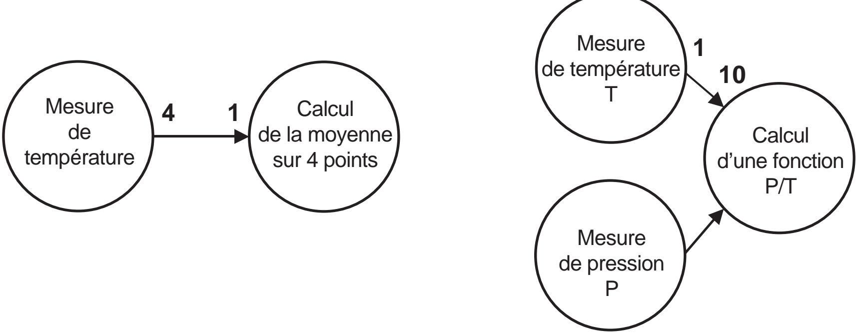


Figure 8.62 – Exemples d'applications faisant intervenir des précédences généralisées entre plusieurs tâches.

Ce problème de précédence doit être abordé selon deux points de vue : exécution et validation :

– Dans le cadre d'un ordonnancement préemptif basé sur la priorité, quelle est la modification des paramètres de tâches qui permettra une exécution dans le respect des échéances ?
– Est-il possible de valider *a priori* l'ordonnançabilité d'une configuration de tâches dépendantes ?

Une réponse à la première question est donnée par : si $\boldsymbol{\tau_i} \rightarrow \boldsymbol{\tau_j}$, la transformation des paramètres doit respecter les règles de précédence suivantes :

– la date de réveil de la deuxième tâche doit être plus grande ou égale à la date de réveil de la première tâche : $\mathbf{r_j} \geq \mathbf{r_i}$;
– dans le respect de la politique d'ordonnancement choisie, la priorité de la première tâche doit être plus grande que la priorité de la deuxième tâche : $\mathbf{Prio_i} > \mathbf{Prio_j}$.

■ Anomalie de comportement en présence de relations de précédence et de primitives de synchronisation

Pour mettre en exergue la difficulté de prise en compte correcte des contraintes de précédence, considérons l'exemple de trois tâches dont les paramètres sont donnés dans le tableau 8.23. De plus deux de ces tâches intègrent des primitives de synchronisation : la tâche $\boldsymbol{\tau_1}$ commence par une attente d'un événement et la tâche $\boldsymbol{\tau_3}$ finit par l'envoi de ce même événement. À cette structure de tâches visualisée sur la figure 8.63, s'ajoute une relation de précédence entre les deux tâches $\boldsymbol{\tau_1}$ et $\boldsymbol{\tau_3}$ représentée par un graphe de précédence (figure 8.63).

Dans une première étape, nous allons respecter les conditions sur les paramètres que nous avons énoncées précédemment ; donc, dans notre cas, nous avons les deux tâches $\boldsymbol{\tau_1}$ et $\boldsymbol{\tau_3}$ liées par une contrainte de précédence qui doivent satisfaire à :

$$r_1 \geq r_3 \quad \text{et} \quad \text{Prio}_3 > \text{Prio}_1 \qquad (8.41)$$

© Dunod – La photocopie non autorisée est un délit.

Tableau 8.23 – Exemple d'une configuration de trois tâches ayant des liens de précédence.

Tâche	r_i	C_i	D_i	T_i
τ_1	0	1	3	4
τ_2	0	1	2	2
τ_3	0	1	2	4

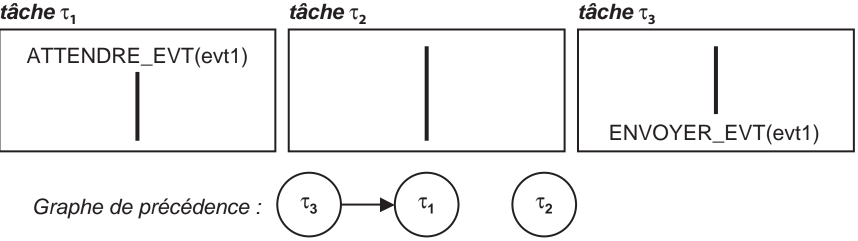


Figure 8.63 – Description des codes et des relations de précédence des tâches décrites dans le tableau 8.23.

Les paramètres temporels de ces tâches permettent de respecter ces conditions étant donné que les deux dates de réveil sont égales et, les périodes étant identiques, nous pouvons affecter les priorités avec les valeurs désirées dans le cas de l'algorithme à priorité fixe RM. La figure 8.64 retrace les trois exécutions possibles en respectant la condition énoncée 8.41. Ces exécutions montrent des séquences non valides (séquences numérotées 1, 2 et 3), c'est-à-dire qu'une des trois tâches ne respecte pas son échéance : tâche $\boldsymbol{\tau_1}$ pour la séquence 1, tâche $\boldsymbol{\tau_1}$ pour la séquence 2 et tâche $\boldsymbol{\tau_2}$ pour la séquence 3. Il est important de noter que ces exécutions sont construites en prenant en compte les primitives de synchronisation des tâches $\boldsymbol{\tau_1}$ et $\boldsymbol{\tau_3}$. Nous pouvons alors tester les autres affectations de priorités ne respectant pas la condition de contrainte de précédence 8.41. La figure 8.64 montre ces trois autres exécutions possibles (séquences numérotées 4, 5 et 6). Les deux premières séquences sont non valides, c'est-à-dire qu'une des trois tâches ne respecte pas son échéance : tâche $\boldsymbol{\tau_1}$ pour la séquence 4 et tâche $\boldsymbol{\tau_2}$ pour la séquence 5. Seule la dernière séquence d'exécution est valide et permet de respecter toutes les échéances des tâches. Cette séquence a pour priorités des tâches la relation suivante :

$$\text{Prio}_1 \geq \text{Prio}_2 \geq \text{Prio}_3$$

Cette relation d'ordre est en effet contradictoire avec la relation 8.41 qui aurait dû être respectée dans le cas de cette relation de précédence entre $\boldsymbol{\tau_1}$ et $\boldsymbol{\tau_3}$. Nous pouvons donc constater une anomalie de comportement. Cette anomalie provient essentiellement du fait de la contradiction entre la relation de précédence et les primitives de synchronisation implémentées dans le code des tâches.

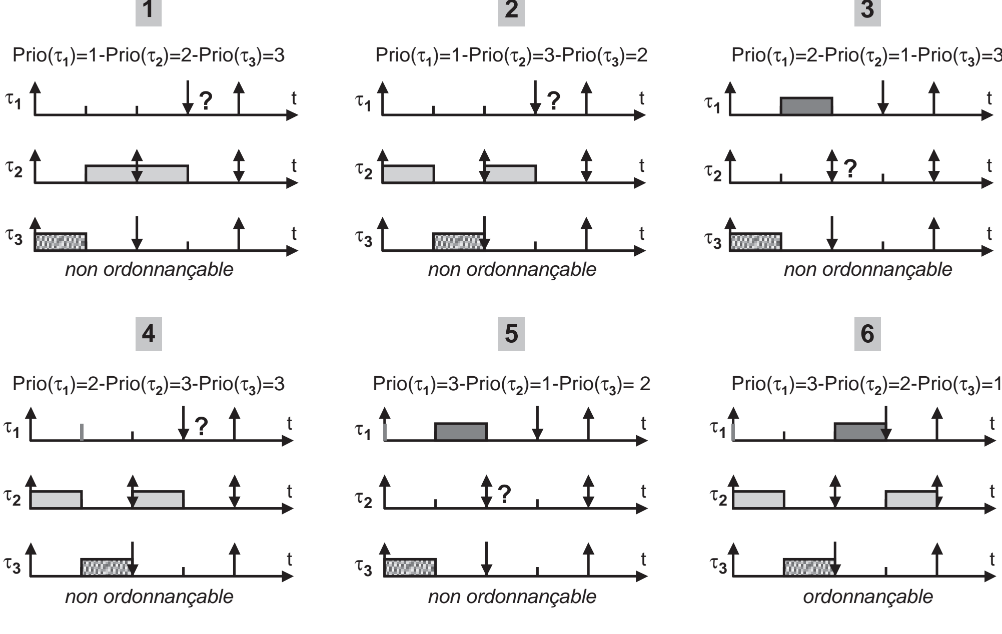


Figure 8.64 – Les différentes séquences d'exécution possibles pour la configuration des trois tâches présentées dans le tableau 8.23 et la figure 8.63.

Nous pouvons donc conclure par :

– Cas du respect de $Prio_3 > Prio_1$: les tâches sont indépendantes dans le sens où les primitives de synchronisation sont inutiles car les priorités traduisent cette synchronisation ou précédence. Les séquences obtenues sont non valides.

– Cas du non respect de $Prio_3 > Prio_1$: il est possible de trouver un exemple d'affectation des priorités qui donne une configuration ordonnançable avec une synchronisation due aux primitives efficientes (la configuration est non ordonnançable sans cette précédence).

En conclusion, nous pouvons noter que les transformations des priorités pour la prise en compte de la précédence ne sont pas optimales.

■ Prise en compte des relations de précédence avec l'algorithme RM

Nous allons considérer que les tâches ne contiennent pas de primitives de synchronisation qui sont, en particulier, en contradiction avec la ou les relations de précédence souhaitées. Dans le cas des algorithmes à priorité fixe et pour des tâches de même période avec des contraintes de précédence, on affecte les priorités afin de satisfaire les relations énoncées 8.41. Cette affectation des priorités ne doit pas contredire les ordres de priorité entre des tâches de périodes différentes.

Considérons l'exemple du graphe de précédence de la figure 8.60 avec les six tâches. En supposant que les tâches $\boldsymbol{\tau_1}$ et $\boldsymbol{\tau_2}$ ont une période de 4 et les quatre autres tâches $\boldsymbol{\tau_1}$, $\boldsymbol{\tau_2}$, $\boldsymbol{\tau_3}$ et $\boldsymbol{\tau_4}$ ont une période supérieure de 5. Pour respecter à la fois l'algorithme

© Dunod – La photocopie non autorisée est un délit.

RM d'affectation de priorités entre les deux groupes de tâches et les contraintes de précédence nous pouvons avoir les quatre affectations de priorités, notées dans le tableau 8.24, qui suivent les règles suivantes :

règle RM : $\text{Prio}(\tau_1, \tau_2) > \text{Prio}(\tau_3, \tau_4, \tau_5, \tau_6)$

régle précédence : $\text{Prio}(\tau_1) > \text{Prio}(\tau_2)$, $\text{Prio}(\tau_3) > \text{Prio}(\tau_5)$

$\text{Prio}(\tau_4) > \text{Prio}(\tau_5)$ et $\text{Prio}(\tau_4) > \text{Prio}(\tau_6)$.

Tableau 8.24 – Affectations des priorités aux tâches selon l'algorithme RM et les contraintes de précédence du graphe de la figure 8.60.

Cas	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
I	6	5	4	3	2	1
II	6	5	3	4	1	2
III	6	5	3	4	2	1
IV	6	5	2	4	1	3

Prise en compte des relations de précédence avec l'algorithme EDF

Dans le cas d'un algorithme à priorité variable, la prise en compte de la précédence est réalisée par la modification de $\mathbf{r_i}$ et $\mathbf{d_i}$, en $\mathbf{r_i}^*$ et $\mathbf{d_i}^*$ suivant les relations :

– Une tâche ne sera activable que si tous ses prédécesseurs ont terminé leur exécution, c'est-à-dire que la date de début de cette tâche doit être supérieure à toutes les dates de début de ces **prédécesseurs immédiats** augmentées de leur durée d'exécution :

$$r_i^* = Max(r_i, Max(r_j^* + C_j \text{ ; si } \tau_j \text{ avant } \tau_i \text{ pour } \tau_j \in \{\text{prédécesseurs immédiats}\}))$$

– Pour une instance donnée, l'échéance d'une tâche doit être inférieure à toutes les échéances de ses **successeurs immédiats** diminuées de leur temps d'exécution :

$$d_i^* = Max(d_i, Max(d_j^* + C_j \text{ ; si } \tau_j \text{ avant } \tau_i \text{ pour } \tau_j \in \{\text{prédécesseurs immédiats}\}))$$

Comme le montre la figure 8.65, cette transformation doit être effectuée en partant des tâches sans prédécesseur pour le calcul des $\mathbf{r_i}^*$ et en commençant par les tâches sans successeur pour le calcul des $\mathbf{d_i}^*$.

Pour appliquer cette transformation, considérons l'exemple d'une configuration de cinq tâches dont les paramètres temporels sont donnés dans le tableau 8.25. Dans une première phase considérons ces tâches sans relation de précédence et construisons la séquence avec l'algorithme à priorité variable EDF. Le facteur d'utilisation est $\mathbf{U} = 0{,}917$ et la période d'étude est $\mathbf{H} = 12$. Étant donné que certaines tâches ne

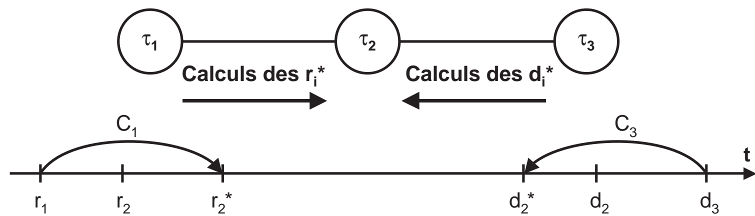


Figure 8.65 – Transformation des paramètres des tâches pour la prise en compte des relations de précédence entre les tâches avec un algorithme à priorité variable de type EDF.

Tableau 8.25 – Exemple d'une configuration de cinq tâches ayant des liens de précédence.

Tâche	r_i	C_i	D_i	T_i
τ_1	0	3	12	12
τ_2	0	2	11	12
τ_3	0	3	12	12
τ_4	0	1	11	12
τ_5	0	2	9	12

sont pas à échéance sur requête, il est nécessaire de tracer la séquence d'exécution sur la période d'étude **H**. Cela donne une séquence valide qui est visualisée sur la figure 8.66. Cette séquence est obtenue en considérant des tâches indépendantes, supposons maintenant que ces tâches sont liées par des relations de précédence données sur la figure 8.67. Il est donc nécessaire de faire le calcul des paramètres $\mathbf{r_i}$* et $\mathbf{d_i}$* des cinq tâches.
Soit le calcul des $\mathbf{r_i}$* :

$$r_1^* = Max(r_1, Max(\varnothing)) = 0$$

$$r_2^* = Max(r_2, Max(r_1^* + C_1)) = Max(0, Max(3)) = 3$$

$$r_3^* = Max(r_3, Max(r_2^* + C_2)) = Max(0, Max(5)) = 5$$

$$r_4^* = Max(r_4, Max(r_3^* + C_1)) = Max(0, Max(3)) = 3$$

$$r_5^* = Max(r_5, Max(r_4^* + C_4, r_2^* + C_2)) = Max(0, Max(4, 5)) = 5$$

Et le calcul des $\mathbf{d_i}$* :

$$d_5^* = Min(d_5, Min(\varnothing)) = 9$$

$$d_4^* = Min(d_4, Min(d_5^* - C_5)) = Min(11, Min(7)) = 7$$

© Dunod – La photocopie non autorisée est un délit.

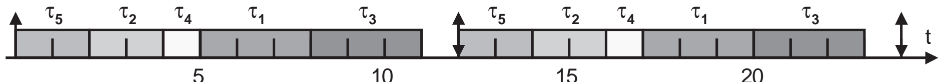


Figure 8.66 – Séquence d'exécution des tâches du tableau 8.25, considérées comme indépendantes, avec l'algorithme à priorité variable EDF.

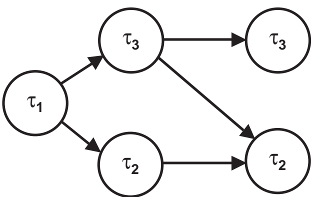


Figure 8.67 – Graphe de précédence reliant les tâches de la configuration décrite dans le tableau 8.25.

$$d_3^* = Min(d_3, Min(\varnothing)) = 12$$

$$d_2^* = Min(d_2, Min(d_3^* - C_3, d_5^* - C_5)) = Min(11, Min(9, 7)) = 7$$

$$d_1^* = Min(d_1, Min(d_2^* - C_2, d_4^* - C_4)) = Min(12, Min(5, 6)) = 5$$

Dans cette nouvelle configuration les tâches sont devenues indépendantes et à départ différé ; alors nous obtenons la séquence donnée dans la figure 8.68.

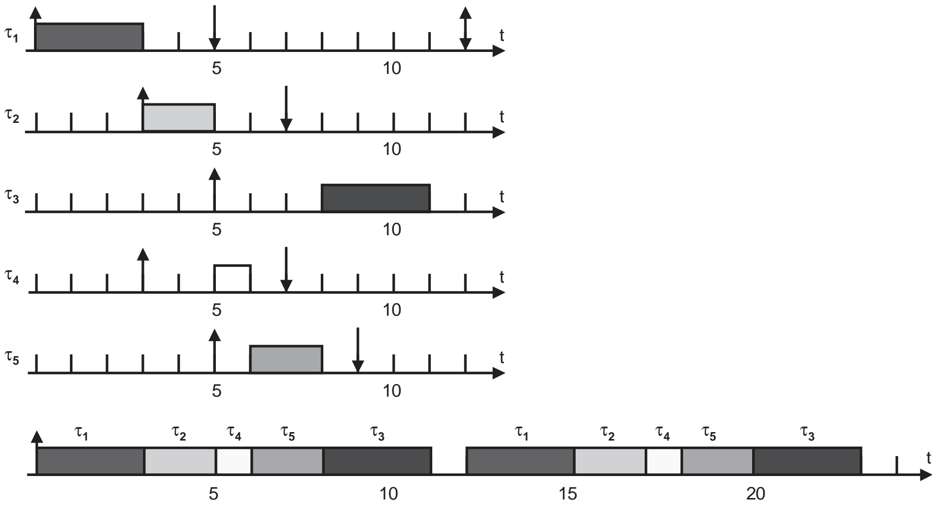


Figure 8.68 – Séquence d'exécution des tâches du tableau 8.25, considérées comme dépendantes (graphe de précédence de la figure 8.67), avec l'algorithme à priorité variable EDF.

Cette séquence (τ_1, τ_2, τ_4, τ_5, τ_3) respecte les relations de précédence imposées par la modification des paramètres ; remarquons que la séquence est différente de la séquence (τ_5, τ_2, τ_4, τ_1, τ_3) élaborée avec les tâches considérées sans relation de précédence (figure 8.66). Rappelons que pour tester l'ordonnançabilité de cette nouvelle configuration avec des tâches à départ différé (équation 8.24), il est théoriquement nécessaire de tracer la séquence sur une durée maximale de 29 unités de temps correspondant à $\mathbf{r_{i,max}}+2\mathbf{H}$. En réalité cette nouvelle séquence a la même période d'étude que précédemment sans phase transitoire.
Il est important de remarquer qu'il y a équivalence de la configuration modifiée T* avec la configuration initiale T, c'est-à-dire :

> « Si la configuration initiale était ordonnançable avec le respect des contraintes de précédence, cette configuration modifiée le sera et réciproquement : T $\Leftrightarrow$ T*. »

8.5.2 Ordonnancement des tâches en présence de ressources critiques

■ Les difficultés de l'ordonnancement en présence de ressources critiques

□ Inversion de priorité

Le principal but, qui conditionne l'utilisation d'algorithme d'ordonnancement basé sur les priorités, est la limitation du temps de réponse d'une ou plusieurs tâches de la configuration. Ainsi, soit un ensemble de **n** tâches partageant un ensemble de **m** ressources critiques en exclusion mutuelle, pour une tâche $\boldsymbol{\tau_i}$ de la configuration ayant une pire durée d'exécution égale à $\mathbf{C_i}$, la question primordiale est :

> « Quel est le pire temps de réponse de la tâche au cours de son exécution (sur la période d'étude **H**) ? »

Nous allons donc analyser le comportement d'une application temps réel composée de tâches partageant des ressources critiques pour déterminer s'il est possible de déterminer pour les tâches de haute priorité un temps de réponse borné : qualité première requise pour un système critique. Pour cette étude, nous allons faire les hypothèses suivantes :

- ordonnancement en ligne préemptif basé sur une priorité fixe ou variable ;
- gestion de la file d'attente des ressources selon la priorité ;
- temps d'exécution de la demande ou de la libération des ressources négligeables.

Ainsi, pour garantir un temps de réponse borné à une tâche donnée $\boldsymbol{\tau_0}$, il faut pouvoir déterminer de façon précise les autres tâches qui peuvent interrompre l'exécution de cette tâche $\boldsymbol{\tau_0}$. Pour une configuration de **n** tâches, la tache $\boldsymbol{\tau_0}$ peut être retardée ou suspendue par deux types de tâches :

- les tâches plus prioritaires que la tâche $\boldsymbol{\tau_0}$;
- les tâches qui partagent au moins une ressource critique avec cette tâche $\boldsymbol{\tau_0}$ et qui l'utilisent avant la demande par la tâche $\boldsymbol{\tau_0}$.

Dans une configuration donnée, les deux ensembles de tâches peuvent être parfaitement identifiés et il est alors possible d'effectuer le calcul du retard maximum dans

© Dunod – La photocopie non autorisée est un délit.

le pire des cas. Soit p tâches de priorités supérieures à la tâche τ_0, n tâches moins prioritaires accédant à une ressource critique utilisée par cette tâche τ_0 ou m ressources utilisées par cette tâche τ_0 et accédées par des tâches moins prioritaires. Dans ces conditions, nous pouvons exprimer le temps de réponse TR_0, défini selon l'équation 8.10, de la tâche τ_0 de la façon suivante :

$$TR_0 = C_0 + \sum_{i=1}^{p} \left\lceil \frac{T_i}{T_0} \right\rceil C_i + \inf\{n, m\} Sc_{\max} \qquad (8.42)$$

avec Sc_{max} la durée maximale des sections critiques.

Pour illustrer ce comportement prenons un exemple simple d'une configuration à quatre tâches dont la tâche τ_0 que nous désirons étudier. La tâche τ_1 est une tâche plus prioritaire que la tâche τ_0. La tâche τ_3 partage une ressource critique avec la tâche τ_0 et la tâche τ_2 est indépendante des autres tâches. En appliquant la relation 8.42, nous devons avoir une estimation du temps de réponse :

$$TR_0 = C_0 + [T_1 / T_0] C_1 + Sc_{\max} \qquad (8.43)$$

La figure 8.69 présente une séquence d'exécution de cette configuration pour des dates de réveil particulières qui vont amener à une situation où la tâche τ_0 va être interrompue dans le pire des cas. Nous pouvons faire le commentaire suivant sur cet exemple de privation d'exécution par occupation d'une ressource critique : le retard pour la tâche τ_0, engendré par la tâche τ_1, est normal (τ_1 plus prioritaire que τ_0), le retard pour la tâche τ_0, engendré par la tâche τ_3, est aussi normal (τ_3 partage une ressource critique avec τ_0). Mais le retard engendré par la tâche τ_2 est un mauvais fonctionnement de l'ordonnancement, car la tâche τ_2 est moins prioritaire que la tâche τ_0 et ne partage aucune ressource critique avec celle-ci. Nous sommes en présence du phénomène dit d'**inversion de priorité**, abordé dans le chapitre 4. Dans ce contexte, il est impossible de déterminer un temps de réponse borné pour une tâche donnée et donc d'obtenir une application temps réel au comportement prévisible

☐ Blocage fatal

Un autre inconvénient, déjà identifié, du partage de ressources critiques est le blocage fatal de deux ou plusieurs tâches lors de l'utilisation d'au moins deux mêmes ressources critiques par deux tâches. La figure 8.70 illustre ce phénomène de blocage fatal (*deadlock*). La demande des ressources R_1 et R_2 de la part des deux tâches τ_1 et τ_2 dans un ordre inverse conduit les deux tâches à se mettre dans une situation d'attente réciproque (par exemple attente de prise de sémaphore).

Une méthode, très utilisée dans les systèmes temps réel critique où le blocage de deux tâches ou plus peut être fatal à une application critique, est la mise en place d'un chien de garde (*watchdog*), gardien d'un temps de blocage maximum d'une tâche. Lorsqu'une tâche se trouve bloquée pendant un temps supérieur à un temps fixé, une alerte de type interruption peut être utilisée pour lancer une tâche de reconfiguration ou faire une remise à zéro du système.

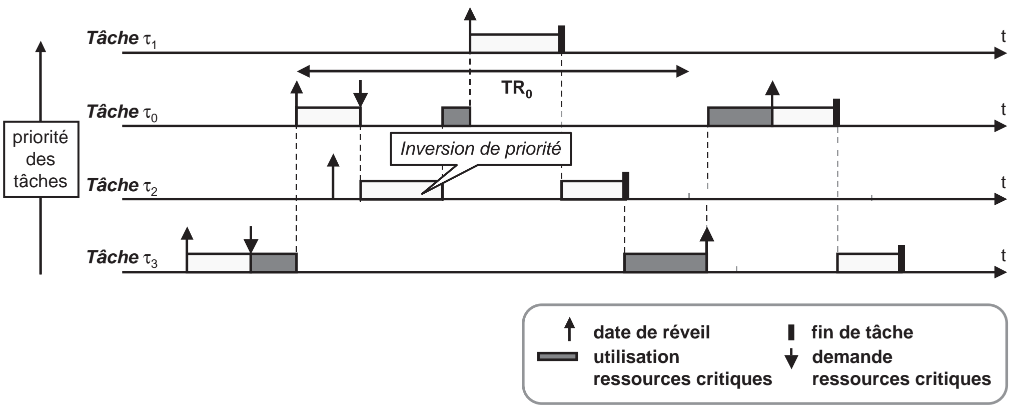


Figure 8.69 – Représentation du phénomène d'inversion de priorité dans un ordonnancement en présence de ressources critiques.

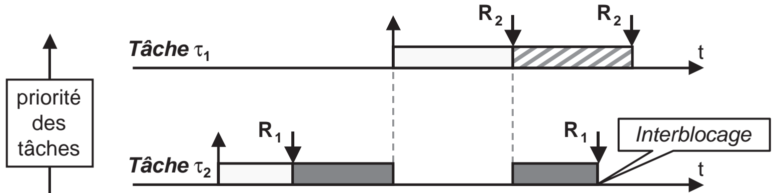


Figure 8.70 – Représentation du phénomène de blocage fatal dans un ordonnancement en présence de ressources critiques.

Une autre des méthodes classiques pour éviter le blocage fatal est d'utiliser dans l'écriture des codes des tâches la méthode des classes ordonnées. L'ensemble des ressources du système est numéroté et les tâches doivent prendre les ressources dans l'ordre croissant des numéros des ressources et les rendre dans l'ordre inverse. Cette méthode permet d'obtenir des résultats très satisfaisants. Ce principe est illustré sur la figure 8.71.

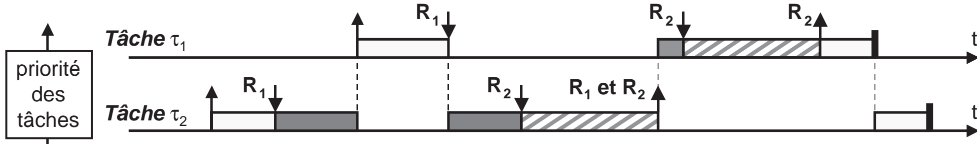


Figure 8.71 – Représentation du procédé d'évitement du blocage fatal dans un ordonnancement en présence de ressources critiques par la méthode des classes ordonnées.

© Dunod – La photocopie non autorisée est un délit.

□ Anomalie de comportement en présence de ressources critiques

Nous avons étudié deux inconvénients majeurs de l'ordonnancement en présence de ressources critiques : l'inversion de priorité qui interdit une évaluation bornée du temps de réponse d'une tâche et le blocage fatal qui conduit à une situation critique de l'application. Un autre inconvénient est plus dissimulé puisqu'il peut faire croire à un bon fonctionnement de l'application testée dans le pire des cas et conduire à un dysfonctionnement dans un autre cas semblant moins contraint en termes de paramètres temporels (durée d'exécution, échéance…).

Considérons l'exécution d'une configuration de trois tâches périodiques dont les paramètres sont donnés dans le tableau 8.26. Le facteur d'utilisation est $\mathbf{U} = 1$ et la période d'étude $\mathbf{H} = 16$. L'affectation des priorités a été effectuée selon l'algorithme DM. Les deux premières tâches $\boldsymbol{\tau_1}$ et $\boldsymbol{\tau_2}$ partagent une ressource critique pendant toute la durée de leur exécution. La dernière tâche $\boldsymbol{\tau_3}$, complètement indépendante des deux autres, a une durée d'exécution qui peut varier entre 5 et 6.

Tableau 8.26 – Exemple d'une configuration de trois tâches partageant une ressource critique.

Tâche	r_i	$C_{i,\alpha}$	$C_{i,\beta}$	$C_{i,\gamma}$	C_i	D_i	T_i	priorité
τ_1	0	0	6	0	6	16	16	**1**
τ_2	0	0	2	0	2	6	8	**3**
τ_3	0	**5 ou 6**	0	0	5	15	16	**2**

Il paraît logique de tester la configuration avec la durée pire cas de la tâche $\boldsymbol{\tau_3}$. Cette séquence d'exécution, présentée sur la figure 8.72, est valide. Un deuxième test peut être effectué pour une durée d'exécution plus petite de la tâche $\boldsymbol{\tau_3}$. Si l'ordonnancement en présence de ressources critiques avait un comportement normal, le relâchement d'une contrainte temporelle comme la durée d'exécution devrait conduire à une ordonnançabilité plus aisée de la configuration. Or la séquence d'exécution, obtenue pour une durée d'exécution de 5 pour la tâche $\boldsymbol{\tau_3}$, est non valide comme le montre la figure 8.73. Lors de son réveil au temps 8, la tâche $\boldsymbol{\tau_2}$ trouve la ressource prise par la tâche $\boldsymbol{\tau_1}$ et doit donc se mettre en attente. Quand la ressource est libérée par la tâche $\boldsymbol{\tau_1}$, la tâche $\boldsymbol{\tau_2}$ n'a plus le temps de s'exécuter avant son échéance.

> Cette anomalie de comportement permet de conclure à la fragilité des tests, en particulier si seules les situations de pire cas sont testées avec des tâches dont la durée d'exécution peut varier en deux instances d'exécution.

En dehors de toute considération sur la durée de la tâche $\boldsymbol{\tau_3}$, ce dysfonctionnement était prévisible. En effet, la tâche $\boldsymbol{\tau_2}$, la plus prioritaire, partage une ressource critique avec la tâche $\boldsymbol{\tau_1}$ moins prioritaire. Donc, il est normal que la tâche $\boldsymbol{\tau_2}$ puisse se retrouver bloquée par cette tâche moins prioritaire qui partage une ressource critique avec elle. Donc le temps de réponse maximum $\mathbf{TR_2}$ de la tâche $\boldsymbol{\tau_1}$ peut être calculé sans tenir compte du phénomène d'inversion de priorité :

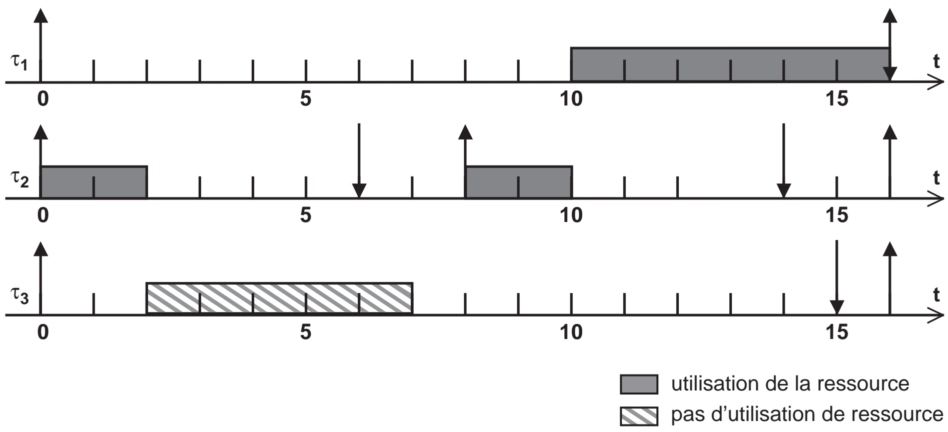


Figure 8.72 – Exemple d'une séquence d'exécution avec un ordonnancement en présence de ressources critiques : séquence valide.

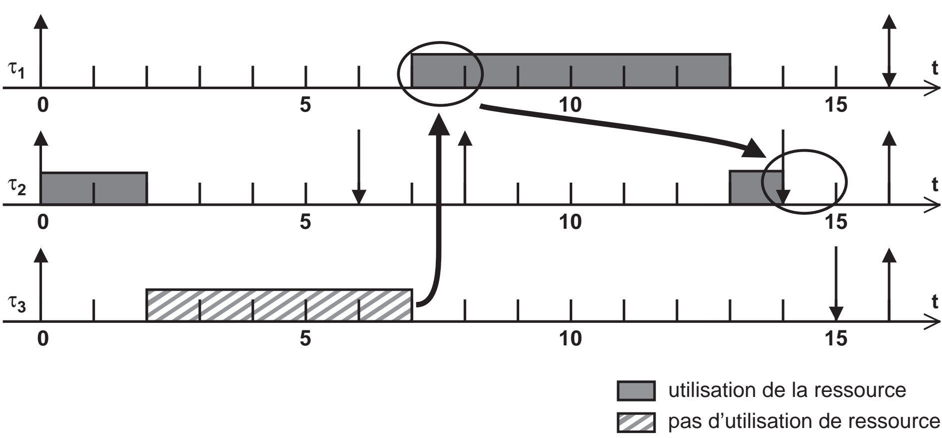


Figure 8.73 – Exemple d'une séquence d'exécution avec un ordonnancement en présence de ressources critiques : séquence non valide.

$$TR_2 = C_1 + C_1 = 2 + 6 = 8 > D_2 = 6$$

Ce temps de réponse est donc supérieur à l'échéance de la tâche ; par conséquent, il était probable que ce dysfonctionnement se produise lors d'une exécution. Ainsi, la durée plus courte de la tâche $\boldsymbol{\tau_3}$, a conduit la tâche $\boldsymbol{\tau_1}$ à prendre la ressource avant la tâche $\boldsymbol{\tau_2}$.

■ Protocole à héritage de priorité pour le traitement des ressources critiques

Afin de répondre à la limitation déterministe du temps de réponse d'une tâche, il est possible d'utiliser le **protocole à héritage de priorité** ou **à priorité héritée**. Ce protocole permet de gérer l'accès (début et fin de section critique d'une tâche)

© Dunod – La photocopie non autorisée est un délit.

de telle façon que les règles de priorité, qui ont conduit à l'évaluation du temps de réponse de l'équation 8.42, soient vraies. Ce protocole à héritage de priorité peut se décrire ainsi (figure 8.74) :

– si la ressource est libre : une tâche accède à cette ressource ;
– si la ressource n'est pas libre : la tâche est bloquée et la tâche possédant la ressource hérite de la priorité de la tâche bloquée.

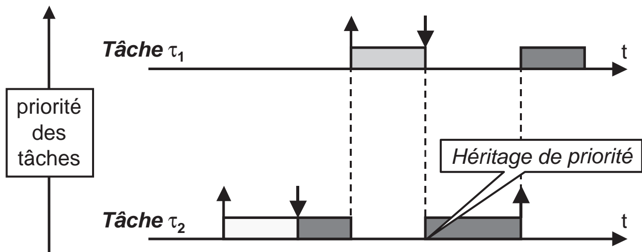


Figure 8.74 – Principe du protocole d'héritage de priorité permettant d'éviter le phénomène d'inversion de priorité dans un ordonnancement en présence de ressources critiques.

Avec ce protocole à héritage de priorité, l'exemple, décrit sur la figure 8.69, a un comportement différent. La tâche $\boldsymbol{\tau_2}$, qui s'était immiscée dans l'attente de la tâche étudiée $\boldsymbol{\tau_0}$, est maintenant rejetée après celle-ci, et cela à cause de la priorité de la tâche $\boldsymbol{\tau_3}$ héritée de la tâche $\boldsymbol{\tau_0}$ jusqu'à la fin de sa section critique (figure 8.75). Les calculs du temps de réponse donnés dans les équations 8.42 et 8.43 pour cet exemple sont exacts.

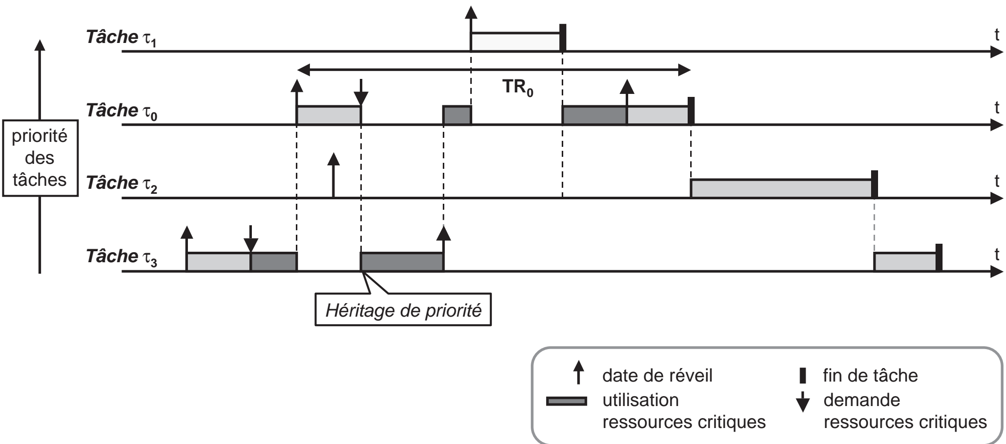


Figure 8.75 – Représentation du protocole d'héritage de priorité permettant d'éviter le phénomène d'inversion de priorité dans un ordonnancement en présence de ressources critiques.

Si la prise des ressources pour une même tâche est faite de manière emboîtée, ce protocole peut s'appliquer plusieurs fois pour une même tâche : héritage multiple. Ce processus est visualisé sur la figure 8.76.

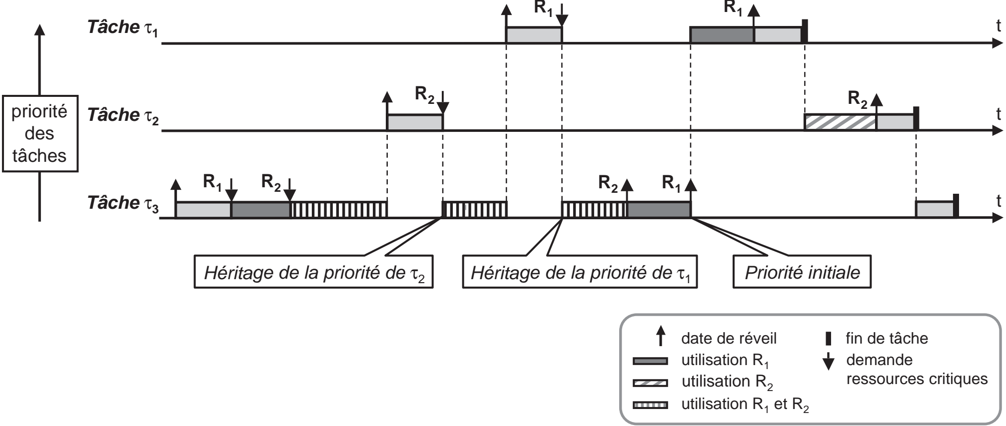


Figure 8.76 – Représentation du protocole d'héritage de priorité à héritage multiple permettant d'éviter le phénomène d'inversion de priorité dans un ordonnancement en présence de plusieurs ressources critiques.

Un exemple complet de prise en compte du partage de ressources critique avec un protocole à héritage de priorité est traité avec la configuration de trois tâches donnée dans le tableau 8.27.

Tableau 8.27 – Exemple d'une configuration de trois tâches partageant une ressource critique.

Tâche	r_i	$C_{i,\alpha}$	$C_{i,\beta}$	$C_{i,\gamma}$	C_i	D_i	T_i
τ_1	0	1	1	0	2	6	6
τ_2	0	2	0	0	2	8	8
τ_3	0	0	4	0	4	12	12

Cette configuration de tâches comprend trois tâches périodiques à échéance sur requête et à départ simultané. Le facteur d'utilisation est **U** = 0,917 et la période d'étude **H** = 24. L'affectation des priorités a été effectuée selon l'algorithme DM. Les deux tâches $\boldsymbol{\tau_1}$ et $\boldsymbol{\tau_3}$ partagent une ressource critique. L'autre tâche $\boldsymbol{\tau_2}$ est complètement indépendante des deux autres tâches. La première analyse d'ordonnancement a été faite sans utiliser le protocole à héritage de priorité et nous pouvons constater une inversion de priorité sur la figure 8.77. Cette inversion de priorité peut être facilement évitée par l'utilisation du protocole à héritage de priorité comme le montre la figure 8.78.

© Dunod – La photocopie non autorisée est un délit.

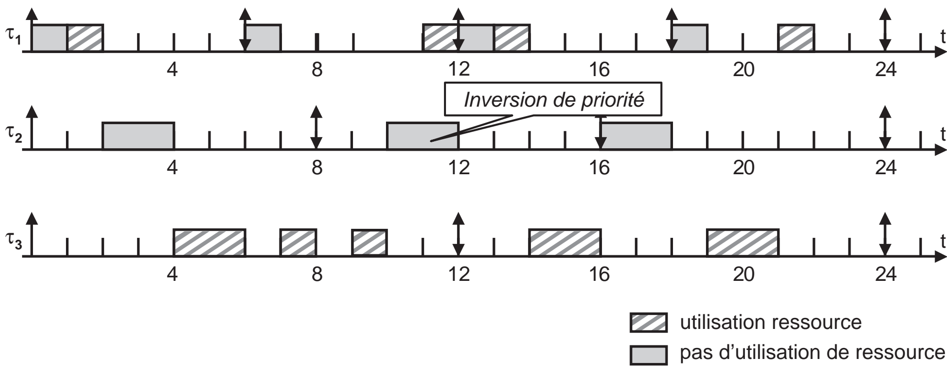


Figure 8.77 – Séquence d'exécution de la configuration décrite dans le tableau 8.27 dans un ordonnancement en présence de ressources critiques : phénomène d'inversion de priorité.

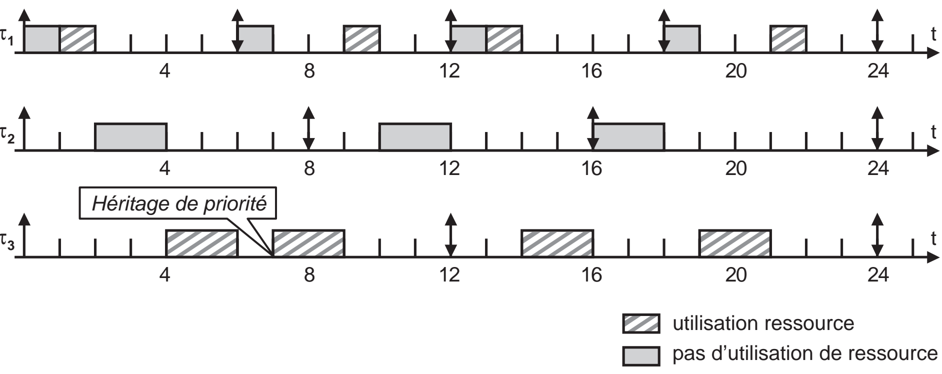


Figure 8.78 – Séquence d'exécution de la configuration décrite dans le tableau 8.27 dans un ordonnancement en présence de ressources critiques : protocole à héritage de priorité.

■ Protocole à priorité plafond pour le traitement des ressources critiques

Le protocole précédent permet donc d'éviter l'inversion de priorité et donc de pouvoir borner le temps de réponse d'une tâche. Toutefois ce temps de réponse comme le montre l'équation 8.42 peut avoir une valeur élevée si beaucoup de tâches partagent une même ressource. De même, ce protocole n'a nullement permis d'éviter le phénomène de blocage fatal. Pour répondre à ces deux inconvénients, il est possible d'utiliser le protocole dit à priorité plafond.

Une ressource possède une priorité seuil, appelée **priorité plafonnée** ou **PP**, qui est la priorité de la tâche la plus prioritaire accédant à cette ressource. Le mécanisme du protocole de gestion des ressources est alors le suivant :

– Si la ressource est libre :
  - une tâche accède à cette ressource si sa priorité est strictement supérieure aux priorités plafonnées des ressources en cours d'utilisation au niveau de toute l'application ;

- sinon la tâche est bloquée et la tâche possédant la ressource hérite de la priorité de la tâche bloquée.

– Si la ressource n'est pas libre : la tâche est bloquée et la tâche possédant la ressource hérite de la priorité de la tâche bloquée.

Notons que ce protocole se différencie du protocole à héritage de priorité simple uniquement s'il y a plus de **2** ressources critiques partagées. Prenons l'exemple de deux tâches τ_1 et τ_2 qui utilisent les deux ressources critiques $\mathbf{R_1}$ et $\mathbf{R_2}$. La tâche τ_1 ayant la priorité la plus forte, les deux ressources possèdent la même priorité plafond ($\mathbf{PP_1} = \mathbf{PP_2} = \mathbf{Prio_1}$), c'est-à-dire la priorité de τ_1. La figure 8.79 montre le principe de fonctionnement de ce protocole.

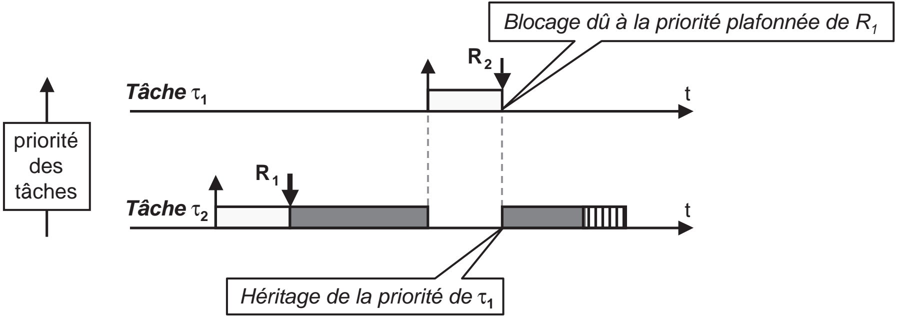


Figure 8.79 – Principe du protocole à priorité plafond dans un ordonnancement en présence de ressources critiques.

Un exemple plus complet est présenté sur la figure 8.80. Les trois tâches partagent deux ressources. La tâche τ_1, la plus prioritaire, utilise les deux ressources $\mathbf{R_1}$ et $\mathbf{R_2}$. La tâche τ_2, de priorité intermédiaire, utilise la ressource $\mathbf{R_2}$. Enfin, la tâche τ_3, possédant la priorité la plus faible, utilise la ressource $\mathbf{R_1}$. Le protocole donne aux deux ressources une priorité plafonnée identique à celle de la tâche τ_1 ($\mathbf{PP_1} = \mathbf{PP_2} = \mathbf{Prio_1}$). Lors de l'exécution, le protocole à héritage de priorité permet de limiter le temps de réponse de la tâche τ_1. En effet, la tâche τ_2 ne peut prendre la ressource $\mathbf{R_2}$ alors qu'elle est libre, puisque la priorité de la tâche τ_2 possède une priorité inférieure à la priorité plafond de la ressource utilisée ($\mathbf{R_1}$) : $\mathbf{Prio_2} < \mathbf{PP_1}$. Avec $\mathbf{Sc_{max}}$ la durée maximale de la plus grande section critique des tâches de plus faible priorité, le temps de réponse **TR** est donc :

$$TR_0 = C_0 + \sum_{i=1}^{p} \left\lceil \frac{T_i}{T_0} \right\rceil C_i + Sc_{\max} \qquad (8.44)$$

Ainsi, en plus de limiter le temps de réponse des tâches, ce protocole à priorité plafond permet de s'affranchir des situations de blocage fatal. Même si la réalisation du code n'a pas suivi la méthode des classes ordonnées, le blocage ne peut se mettre en place (figure 8.81). Ainsi, nous pouvons constater que la tâche τ_1 ne peut pas prendre la ressource $\mathbf{R_2}$ qui aurait bloqué l'application, car la priorité de la tâche τ_1

© Dunod – La photocopie non autorisée est un délit.

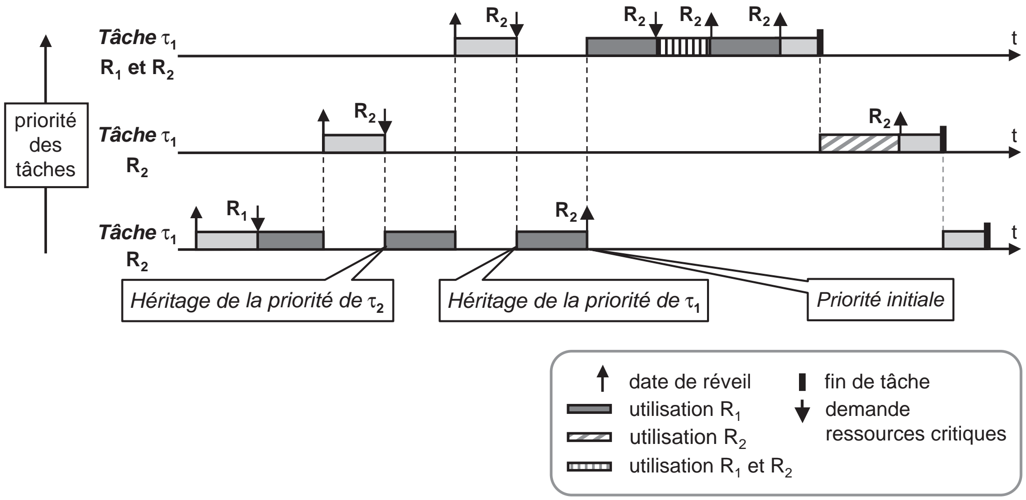


Figure 8.80 – Exemple du comportement du protocole à priorité plafond dans un ordonnancement en présence de ressources critiques.

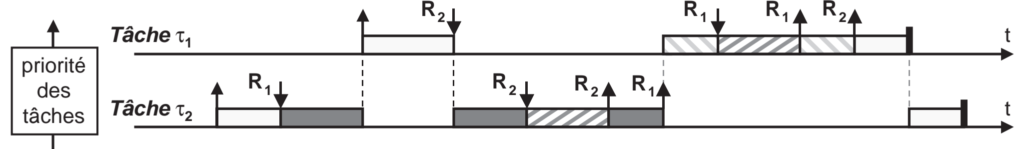


Figure 8.81 – Évitement du blocage fatal grâce au protocole à priorité plafond.

a une priorité qui n'est pas strictement supérieure à la priorité plafond de la ressource utilisée ($\mathbf{R_1}$).

Dans le cas de l'algorithme EDF, cet algorithme est complexe puisque la priorité plafonnée des ressources change au fur et à mesure de l'exécution. Le coût système est bien évidemment plus important en utilisant un algorithme d'ordonnancement de type Earliest Deadline First avec un protocole de gestion des ressources critiques de type **héritage de priorité avec priorité plafonnée dynamique**. D'autres algorithmes peuvent être utilisés comme le protocole à pile. Plusieurs exécutifs (POSIX, OSEK/VDX, Ada) mettent en œuvre une version simplifiée du protocole à héritage de priorité et priorité plafonnée.

■ Conclusion sur les protocoles utilisés pour le traitement des ressources critiques

Nous pouvons noter que, si l'algorithme d'ordonnancement est mis en place au niveau de l'ordonnanceur du noyau, les protocoles de gestion des ressources critiques sont intégrés aux primitives de protection des sections critiques (sémaphore…).

La validation temporelle de l'exécution des applications avec les algorithmes RM, DM, ED, ML est toujours possible avec des tâches partageant des ressources

critiques ; il faut intégrer aux durées des tâches les durées de blocage maximum dues aux attentes des ressources critiques (cette durée de blocage dépend du protocole de gestion des ressources). Donc chaque tâche de l'application aura une durée d'exécution exprimée sous la forme :

$$C_i^* = C_i + B_i$$

avec $\mathbf{B_i}$: durée de blocage due aux ressources critiques
Cette propriété sera utilisée dans la prochaine section pour faire une validation des applications.

8.6 Analyse d'ordonnançabilité en environnement monoprocesseur

8.6.1 Modélisation et ordonnancement des applications temps réel

Nous sommes partis d'un modèle théorique simple : la tâche périodique et indépendante. Sur la base de ce modèle théorique, nous avons étudié des algorithmes d'ordonnancement, optimaux dans leur domaine, permettant d'élaborer des conditions d'ordonnançabilité. Il est donc extrêmement intéressant de pouvoir transformer toutes les applications temps réel en un ensemble de tâches périodiques indépendantes. Or nous trouvons dans les applications temps réel, des tâches apériodiques, des tâches dépendantes et des tâches partageant des ressources critiques. La méthodologie d'analyse des applications temps réel consiste donc à transformer l'application initiale pour la rendre compatible avec l'environnement théorique de validation. Ainsi, nous avons les transformations suivantes (figure 8.82) :

– tâches périodiques indépendantes : les tâches restent identiques (modèle théorique initial) ;

– tâches apériodiques indépendantes : les tâches sont transformées en tâches périodiques à l'aide d'un des modèles de serveur (§ 8.4) ;

– tâches dépendantes : les tâches sont transformées en tâches indépendantes en transformant les paramètres temporels (§ 8.5) ;

– tâches partageant des ressources : l'utilisation des protocoles de gestion des ressources (protocoles à héritage de priorité ou à priorité plafond) permet de borner la durée de chaque tâche et ainsi de considérer les tâches comme indépendantes : méthode d'analyse RMA (§ 8.6.2).

L'application ainsi transformée est équivalente en termes d'ordonnançabilité à l'application initiale ; par conséquent, si l'application temps réel transformée est ordonnançable, l'application initiale non transformée est aussi ordonnançable. Si, dans le contexte d'un algorithme optimal pour la configuration, l'application temps réel transformée est non ordonnançable, l'application initiale non transformée ne peut pas être ordonnançable en utilisant un algorithme de la même puissance (algorithmes à priorité fixe ou algorithmes à priorité variable).

© Dunod – La photocopie non autorisée est un délit.

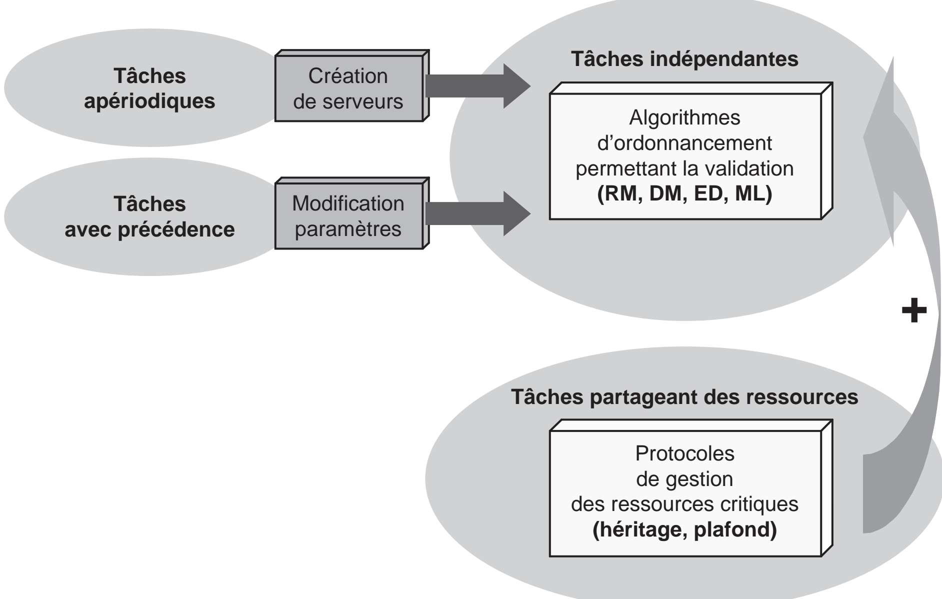


Figure 8.82 – Méthodologie d'analyse d'une application temps réel afin d'utiliser des algorithmes d'ordonnancement permettant une validation.

8.6.2 Méthode de validation RMA

Cette prise en compte des durées de blocage a permis de mettre en œuvre une méthodologie de validation temporelle d'application temps réel dont l'algorithme d'ordonnancement est fondé sur Rate Monotonic RM (affectation des priorités selon la période) et un protocole de gestion de ressources (héritage de priorité ou héritage de priorité avec priorité plafonnée). La méthodologie est appelée « Rate Monotonic Analysis » ou RMA.

Nous allons supposer d'une part que les tâches sont périodiques (utilisation d'un serveur périodique) et d'autre part que les tâches sont indépendantes au sens synchronisation (utilisation de la modification des paramètres des tâches). Ensuite, nous allons procéder au calcul du temps de blocage des tâches.

Considérons un ensemble de **n** tâches indépendantes, à échéance sur requête, classé par ordre de priorité (la tâche τ_i plus prioritaire que la tâche τ_j si $i<j$: $Prio_i > Prio_j$). Ces tâches partagent **m** ressources critiques R_k. Nous allons de plus définir les grandeurs suivantes :

– $b_{j,k}$: la durée de la section critique de la tâche τ_j utilisant la ressource R_k avec $b_{j,k} = 0$ si la tâche τ_j n'utilise pas la ressource R_k.
– PP_k : la priorité plafond de la ressource R_k, c'est-à-dire la priorité la plus haute des tâches utilisant la ressource R_k.

Ainsi, nous pouvons écrire que le temps de blocage d'une tâche τ_i, appelé B_i, s'exprime de la façon suivante :

Cas du protocole à héritage de priorité :

$$B_i = Min\left(\sum_{j=i+1}^{n} \underset{k}{Max}[b_{j,k} : \text{si } PP_k \geq \text{Prio}_i], \sum_{k=1}^{n} \underset{j>i}{Max}[b_{j,k} : \text{si } PP_k \geq \text{Prio}_i]\right) \quad (8.45)$$

Cas du protocole à priorité plafond :

$$B_i = \underset{k \in [1,m],\, j>i}{Max}[b_{j,k} : \text{si } PP_k \geq \text{Prio}_i] \quad (8.46)$$

Considérons une configuration composée de six tâches périodiques à échéances sur requête et à départ simultané. Ces tâches, dont les priorités sont affectées selon l'algorithme RM et classées par ordre de priorité selon les indices (le plus petit indice correspond à la tâche la plus prioritaire), partagent trois ressources critiques $\mathbf{R_1}$, $\mathbf{R_2}$ et $\mathbf{R_3}$. Les caractéristiques temporelles des tâches sont données dans le tableau 8.28. Le facteur d'utilisation est $\mathbf{U} = 0{,}4$ et la période d'étude $\mathbf{H} = 5000$. Les priorités plafond de ces ressources critiques sont : $\mathbf{PP_1} = \mathbf{Prio_2}$, $\mathbf{PP_2} = \mathbf{Prio_1}$ et $\mathbf{PP_3} = \mathbf{Prio_1}$. Nous pouvons déterminer la matrice $\mathbf{B_{sc}}$ des durées des sections critiques de cette application :

$$B_{sc} = \begin{matrix} b_{j,k} & R_1 & R_2 & R_3 \\ \tau_1 & 0 & 2 & 2 \\ \tau_2 & 5 & 5 & 5 \\ \tau_3 & 10 & 0 & 0 \\ \tau_4 & 10 & 0 & 0 \\ \tau_5 & 0 & 10 & 0 \\ \tau_6 & 0 & 15 & 0 \end{matrix}$$

Nous pouvons donc calculer les durées de blocage en utilisant les relations 8.45 et 8.46 pour les deux types de protocoles (tableau 8.29). Dans le cas du protocole à héritage de priorité, nous avons l'évaluation du terme $\mathbf{B_1}$:

$$B_i = Min\left(\sum_{j=2}^{6} Max[b_{j,2}, b_{j,3}], \sum_{k=2}^{3} Max[b_{2,k}, b_{3,k}, b_{4,k}, b_{5,k}, b_{6,k}]\right)$$

$$\text{car} \quad PP_1 < \text{Prio}_1$$

$$= Min\{[Max(5, 5) + Max(0, 0) + Max(0, 0) + Max(10, 0) + Max(15, 0)]$$
$$[Max(5, 0, 0, 10, 15) + Max(5, 0, 0, 0, 0)]\}$$
$$= Min([5 + 0 + 0 + 10 + 15], [15 + 5]) = Min(30, 20) = 20$$

Dans le cas du protocole à priorité plafond, nous avons l'évaluation du terme $\mathbf{B_1}$:

$$B_i = Max[b_{2,2}, b_{3,2}, b_{4,2}, b_{5,2}, b_{6,2}, b_{2,3}, b_{3,3}, b_{4,3}, b_{5,3}, b_{6,3}] \quad \text{car} \quad PP_1 < \text{Prio}_1$$
$$= Max[5, 0, 0, 10, 15, 5, 0, 0, 0] = 15$$

© Dunod – La photocopie non autorisée est un délit.

Tableau 8.28 – Exemple d'une configuration de six tâches partageant trois ressources critiques.

Ressource		R$_1$ (PP$_1$=Prio$_2$)			R$_2$ (PP$_2$=Prio$_1$)			R$_3$ (PP$_3$=Prio$_1$)					
Tâche	r_i	$C_{i,\alpha}$	$C_{i,\beta}$	$C_{i,\gamma}$	$C_{i,\alpha}$	$C_{i,\beta}$	$C_{i,\gamma}$	$C_{i,\alpha}$	$C_{i,\beta}$	$C_{i,\gamma}$	C_i	D_i	T_i
τ_1	0	0	0	0	0	2	2	2	**2**	0	4	40	40
τ_2	0	0	**5**	10	5	**5**	5	10	**5**	0	15	50	50
τ_3	0	0	**10**	0	0	0	0	0	0	0	10	500	500
τ_4	0	0	**10**	0	0	0	0	0	0	0	10	500	500
τ_5	0	0	0	0	20	**10**	20	0	0	0	50	1000	1000
τ_6	0	0	0	0	15	**15**	20	0	0	0	50	5000	5000

Les durées de blocage sont ainsi calculées et reportées dans le tableau 8.29. À partir de ces données, nous pouvons mettre en œuvre plusieurs techniques d'analyse de l'ordonnançabilité de la configuration. Celles-ci permettent de valider de plus en plus précisément l'application, c'est-à-dire que l'analyse devient de moins en moins « pire cas ».

Tableau 8.29 – Exemple d'une configuration de six tâches partageant trois ressources critiques : calcul des durées de blocage.

Tâche	**u_i**	**Prio**	**B_i (priorité héritée)**	**B_i (priorité plafond)**
τ_1	0,1	6	20	15
τ_2	0,2	5	25	15
τ_3	0,02	4	25	15
τ_4	0,02	3	15	15
τ_5	0,05	2	15	15
τ_6	0,01	1	0	0
U =	**0,4**			

■ Technique 1 : analyse RM simple

Considérons une première technique d'analyse qui est une traduction directe de la condition suffisante de l'algorithme RM (équation 8.27) en intégrant pour chacune des tâches le temps de blocage associé, soit :

$$U = \sum_{i=1}^{n} (C_i + B_i)/T_i \leq n\left(2^{\frac{1}{n}} - 1\right) \qquad (8.47)$$

Pour l'application décrite dans les tableaux 8.28 et 8.29, nous avons un facteur d'utilisation sans prendre en compte les ressources de **U** = 0,4 (tableau 8.29). La limite supérieure permettant de décider de l'ordonnançabilité de la configuration est donc satisfaite comme le montre le calcul suivant :

$$U = \sum_{i=1}^{6}\left(\frac{C_i}{T_i}\right) = \frac{4}{40} + \frac{10}{50} + \frac{10}{500} + \frac{10}{500} + \frac{50}{1000} + \frac{50}{5000}$$

$$= 0{,}4 \leq 6\left(2^{\frac{1}{6}} - 1\right) \approx 0{,}735$$

Nous pouvons maintenant faire l'évaluation du facteur d'utilisation en présence des ressources critique avec l'équation 8.47. Donc, dans les deux cas correspondant aux deux protocoles de gestion de ressources critiques, nous obtenons :

Cas du protocole à héritage de priorité :

$$U = \sum_{i=1}^{6}\left(\frac{C_i + B_i}{T_i}\right) = \left(\frac{(4+20)}{40} + \frac{(10+25)}{50} + \frac{(10+25)}{500} + \frac{(10+15)}{500} + \frac{(50+15)}{1\,000} + \frac{50}{5\,000}\right)$$

$$\approx 1{,}43 > 0{,}735$$

Cas du protocole à priorité plafond :

$$U = \sum_{i=1}^{6}\left(\frac{C_i + B_i}{T_i}\right) = \left(\frac{(4+15)}{40} + \frac{(10+15)}{50} + \frac{(10+15)}{500} + \frac{(10+15)}{500} + \frac{(50+15)}{1\,000} + \frac{50}{5\,000}\right)$$

$$\approx 1{,}14 > 0{,}735$$

Nous voyons que, dans les deux cas, la condition n'est pas respectée et que nous devons conclure à une non validité de l'ordonnancement de la configuration avec une affectation de priorité basée sur l'algorithme RM et les protocoles de gestion de ressources critiques définis. Nous pouvons remarquer que ces calculs montrent des facteurs d'utilisation supérieurs à 100 % ; cela met en exergue l'aspect « pire cas » très pessimiste de ce test.

© Dunod – La photocopie non autorisée est un délit.

■ Technique 2 : analyse RM améliorée

La technique précédente peut être améliorée en considérant que le blocage ne doit être pris en compte que pour une tâche pour une période d'étude. Ne connaissant pas la tâche qui va être bloquée par les autres, il est nécessaire de considérer le temps de blocage maximum, d'où la relation :

$$U = \sum_{i=1}^{n}\left(\frac{C_i}{T_i}\right) + \underset{i \in [1,\, n]}{Max}\left(\frac{B_i}{T_i}\right) \le n\left(2^{\frac{1}{n}} - 1\right) \qquad (8.48)$$

Appliquons cette technique à l'application que nous avons déjà testée. Donc, dans les deux cas correspondant aux deux protocoles de gestion de ressources critiques, nous obtenons :

Cas du protocole à héritage de priorité :

$$U = \frac{4}{40} + \frac{10}{50} + \frac{10}{500} + \frac{10}{500} + \frac{50}{1000} + \frac{50}{5000}$$
$$+ Max\left\{\frac{20}{40}, \frac{25}{50}, \frac{25}{500}, \frac{15}{500}, \frac{15}{1000}\right\}$$
$$= 0{,}4 + 0{,}5 = 0{,}9 < 0{,}735 \text{ NON}$$

Cas du protocole à priorité plafond :

$$U = \frac{4}{40} + \frac{10}{50} + \frac{10}{500} + \frac{10}{500} + \frac{50}{1000} + \frac{50}{5000}$$
$$+ Max\left\{\frac{15}{40}, \frac{15}{50}, \frac{15}{500}, \frac{15}{500}, \frac{15}{1000}\right\}$$
$$= 0{,}4 + 0{,}375 = 0{,}775 < 0{,}735 \text{ NON}$$

Cette condition d'ordonnançabilité est encore trop pessimiste pour permettre de conclure sur l'ordonnançabilité de cette configuration. Il est donc nécessaire de mettre en œuvre une méthode plus fine.

■ Technique 3 : analyse RM itérative

Considérons une autre technique d'analyse qui est une traduction itérative de la condition suffisante de l'algorithme RM (équation 8.27). Le test est effectué à partir de la tâche la plus prioritaire en intégrant progressivement chacune des tâches avec leur temps de blocage, soit :

$$\forall j \in [1,\, n] \; : \; U = \sum_{i=1}^{j-1} \frac{C_i}{T_i} + \frac{C_j + B_j}{T_j} \le j\left(2^{\frac{1}{j}} - 1\right) \qquad (8.49)$$

Le principe développé dans cette technique est simple. Pour une tâche $\boldsymbol{\tau_j}$, on considère d'une part les tâches qui peuvent l'interrompre (les tâches les plus prioritaires) ; cela correspond au premier terme de l'équation 8.49 (sommation du facteur d'utilisation de toutes les tâches plus prioritaires que de la tâche $\boldsymbol{\tau_j}$). Et d'autre part, il faut prendre en compte la durée de la tâche $\boldsymbol{\tau_j}$ et son terme de blocage $\mathbf{B_j}$.
Appliquons cette technique à l'application que nous avons déjà testée. Dans les deux cas correspondant aux deux protocoles de gestion de ressources critiques, nous obtenons :
Cas du protocole à héritage de priorité :

$$\text{Pour } j = 1 : \frac{(4+20)}{40} = 0{,}6 < 1$$

$$\text{Pour } j = 2 : \frac{4}{40} + \frac{(10+25)}{50} = 0{,}8 < 0{,}828$$

$$\text{Pour } j = 3 : \frac{4}{40} + \frac{10}{50} + \frac{(10+25)}{500} = 0{,}37 < 0{,}779$$

$$\text{Pour } j = 4 : \frac{4}{40} + \frac{10}{50} + \frac{10}{500} + \frac{(10+15)}{500} = 0{,}37 < 0{,}757$$

$$\text{Pour } j = 5 : \frac{4}{40} + \frac{10}{50} + \frac{10}{500} + \frac{10}{500} + \frac{(10+15)}{500} = 0{,}405 < 0{,}743$$

$$\text{Pour } j = 6 : \frac{4}{40} + \frac{10}{50} + \frac{10}{500} + \frac{10}{500} + \frac{50}{1000} + \frac{50}{5000} = 0{,}4 < 0{,}735$$

Cas du protocole à priorité plafond :

$$\text{Pour } j = 1 : \frac{(4+15)}{40} = 0{,}475 < 1$$

$$\text{Pour } j = 2 : \frac{4}{40} + \frac{(10+15)}{50} = 0{,}6 < 0{,}828$$

$$\text{Pour } j = 3 : \frac{4}{40} + \frac{10}{50} + \frac{(10+15)}{500} = 0{,}35 < 0{,}779$$

$$\text{Pour } j = 4 : \frac{4}{40} + \frac{10}{50} + \frac{10}{500} + \frac{(10+15)}{500} = 0{,}37 < 0{,}757$$

$$\text{Pour } j = 5 : \frac{4}{40} + \frac{10}{50} + \frac{10}{500} + \frac{10}{500} + \frac{(10+15)}{500} = 0{,}405 < 0{,}743$$

$$\text{Pour } j = 6 : \frac{4}{40} + \frac{10}{50} + \frac{10}{500} + \frac{10}{500} + \frac{50}{1000} + \frac{50}{5000} = 0{,}4 < 0{,}735$$

Avec cette technique l'ordonnançabilité de la configuration est prouvée. Nous pouvons ainsi déclarer que la configuration des six tâches partageant des ressources critiques est ordonnançable avec l'affectation de priorité selon l'algorithme RM. Nous pouvons remarquer toutefois que l'utilisation du protocole de gestion de res-

© Dunod – La photocopie non autorisée est un délit.

sources critiques de type à héritage de priorité donne une validation très proche de la limite (cas j = 2).

Il est intéressant de remarquer que cette technique permet d'expliquer la technique précédente. En effet, l'équation 8.48 de la technique 2 est une majoration grossière de l'équation 8.49 de la technique 3.

$$\forall j \in [1, n] : U = \sum_{i=1}^{j-1} \frac{C_i}{T_i} + \frac{C_j + B_j}{T_j} \leq j\left(2^{\frac{1}{j}} - 1\right) \quad (8.49)$$

$$\Leftarrow \forall j \in [1, n] : U = \sum_{i=1}^{j-1} \frac{C_i}{T_i} + \frac{C_j + B_j}{T_j} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

$$\Leftarrow \forall j \in [1, n] : U = \sum_{i=1}^{n} \frac{C_i}{T_i} + \frac{B_j}{T_j} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

$$\Leftarrow U = \sum_{i=1}^{n} \frac{C_i}{T_i} + \underset{i \in [1, n]}{Max}\left(\frac{B_i}{T_i}\right) \leq n\left(2^{\frac{1}{n}} - 1\right)$$

■ **Technique 4 : analyse par temps de réponse**

Cette technique calcule le pire temps de réponse d'un ensemble de tâches. Pour effectuer ce calcul, il est nécessaire de définir le temps pendant lequel le processeur est occupé à exécuter des tâches de priorité supérieure ou égale à $\mathbf{Prio_i}$ (période d'activité ou « *busy period* » de niveau **i**). Cette analyse s'effectue en plusieurs étapes :

– **Étape 1** : évaluation de la date de fin d'exécution $\mathbf{a_0}$ de la tâche $\boldsymbol{\tau_i}$ dans cette période d'activité. Le calcul de $\mathbf{a_0}$ prend en compte l'exécution d'une instance de toutes les tâches plus prioritaires que la tâche $\boldsymbol{\tau_i}$ et le temps de blocage de la tâche $\boldsymbol{\tau_i}$:

$$a_0 = B_i + \sum_{j=1}^{i} C_i \quad (8.50)$$

– **Étape 2** : en utilisant le calcul de $\mathbf{a_n}$, l'équation 8.51 détermine la prochaine itération $\mathbf{a_{n+1}}$:

$$a_{n+1} = B_i + C_i + \sum_{j=1}^{i-1} \left\lceil \frac{a_n}{T_j} \right\rceil C_j \quad (8.51)$$

– Notons que, dans cette période d'activité de niveau **i**, au plus un facteur de blocage $\mathbf{B_i}$ est pris en compte. En effet, aucune tâche de priorité inférieure à ce niveau **i** ne peut débuter et donc prendre une ressource dans cette période.

– **Étape 3** : cette étape va déterminer si cette approximation $\mathbf{a_{n+1}}$ est le temps de terminaison de $\boldsymbol{\tau_i}$. Si la valeur de $\mathbf{a_{n+1}}$ est supérieure à l'échéance de la tâche : $\mathbf{D_i}$, alors l'analyse et terminée et le test a échoué. Si $\mathbf{a_{n+1}} \neq \mathbf{a_n}$, alors reprendre l'étape 2.

– **Étape 4** : si $\mathbf{a_{n+1}} = \mathbf{a_n}$, alors nous avons obtenu le temps de terminaison de la période d'activité. Nous pouvons alors obtenir le temps de réponse $\mathbf{TR_i}$ de la tâche $\boldsymbol{\tau_i}$, soit :

$$TR_i = a_n \qquad (8.52)$$

– Nous pouvons alors conclure sur l'ordonnancement si ce temps de réponse $\mathbf{TR_i}$ de la tâche $\boldsymbol{\tau_i}$ est inférieur à son échéance $\mathbf{D_i}$.

Illustrons cette technique pour la configuration précédemment étudiée et décrite par les tableaux 8.28 et 8.29 en considérant un protocole à héritage de priorité. Nous allons nous intéresser au temps de réponse de la tâche $\boldsymbol{\tau_2}$ sachant que cela correspond à une étude classique : « analyser le temps de réponse maximum d'une tâche d'une configuration ». En suivant les différentes étapes données, nous obtenons :

– **Étape 1** : évaluation de la date de fin d'exécution $\mathbf{a_0}$ de la tâche $\boldsymbol{\tau_2}$ dans la période d'activité du niveau de priorité 3 :

$$a_0 = B_2 + \sum_{j=1}^{2} C_j = 25 + (4 + 10) = 39$$

– **Étape 2** : en utilisant le calcul de $\mathbf{a_0}$, nous obtenons $\mathbf{a_1}$:

$$a_1 = B_2 + C_2 + \sum_{j=1}^{1} \left| \frac{a_0}{T_j} \right| C_j = 25 + (10) + \left\lceil \frac{39}{40} \right\rceil 4 = 39$$

– **Étape 3** : comme $\mathbf{a_1} = \mathbf{a_0}$, alors nous allons à l'étape 4.

– **Étape 4** : nous avons obtenu le temps de terminaison de l'instance de la tâche $\boldsymbol{\tau_2}$. Nous pouvons alors obtenir le temps de réponse $\mathbf{TR_2}$ de cette tâche $\boldsymbol{\tau_2}$, soit :

$$TR_2 = a_2 = 39$$

– **Étape 5** : La période d'activité est terminée, car $\mathbf{TR_2}$ est inférieur ou égal à $\mathbf{T_2}$; l'échéance de la tâche $\boldsymbol{\tau_2}$ est donc bien respectée : $\mathbf{TR_2} = 39 < \mathbf{D_2} = \mathbf{T_2} = 50$.

Poursuivons notre analyse par la tâche $\boldsymbol{\tau_3}$. Soit les étapes suivantes :

– **Étape 1** : évaluation de la date de fin d'exécution $\mathbf{a_0}$ de la tâche $\boldsymbol{\tau_3}$ dans la période d'activité du niveau de priorité 3 :

$$a_0 = B_3 + \sum_{j=1}^{3} C_j = 25 + (4 + 10 + 10) = 49$$

© Dunod – La photocopie non autorisée est un délit.

– **Étape 2.1** : en utilisant le calcul de $\mathbf{a_0}$, nous obtenons une valeur de $\mathbf{a_1}$ qui fait apparaître le fait que la tâche $\boldsymbol{\tau_1}$ peut s'exécuter deux fois avant la date $\mathbf{a_0}$, soit :

$$a_1 = B_3 + C_3 + \sum_{j=1}^{1} \left| \frac{a_0}{T_j} \right| C_j = 25 + (10) + \left\lceil \frac{49}{40} \right\rceil 4 + \left\lceil \frac{49}{50} \right\rceil 10 = 53$$

– **Étape 3.1** : comme $\mathbf{a_1} \neq \mathbf{a_0}$ (dû à l'exécution de deux instances de la tâche $\boldsymbol{\tau_1}$), alors nous reprenons l'étape 2.

– **Étape 2.2** : en utilisant le calcul de $\mathbf{a_1}$, nous obtenons $\mathbf{a_2}$ qui fait apparaître le fait que les tâches $\boldsymbol{\tau_1}$ et $\boldsymbol{\tau_2}$ peuvent s'exécuter deux fois avant la date $\mathbf{a_1}$, soit :

$$a_2 = B_3 + C_3 + \sum_{j=1}^{2} \left| \frac{a_1}{T_j} \right| C_j = 25 + (10) + \left\lceil \frac{53}{40} \right\rceil 4 + \left\lceil \frac{53}{50} \right\rceil 10 = 63$$

– **Étape 3.2** : comme $\mathbf{a_2} \neq \mathbf{a_1}$ (dû à l'exécution des deux instances des tâches $\boldsymbol{\tau_1}$ et $\boldsymbol{\tau_2}$), alors nous reprenons à l'étape 2.

– **Étape 2.3** : en utilisant le calcul de $\mathbf{a_2}$, nous obtenons $\mathbf{a_3}$:

$$a_3 = B_3 + C_3 + \sum_{j=1}^{2} \left| \frac{a_2}{T_j} \right| C_j = 25 + (10) + \left\lceil \frac{63}{40} \right\rceil 4 + \left\lceil \frac{63}{50} \right\rceil 10 = 63$$

– **Étape 3.3** : comme $\mathbf{a_3} = \mathbf{a_2}$ (dû à la fin l'itération car, avant la date $\mathbf{a_2}$, les tâches $\boldsymbol{\tau_1}$ et $\boldsymbol{\tau_2}$ peuvent s'exécuter deux fois et la tâche $\boldsymbol{\tau_3}$ une seule fois), alors nous allons à l'étape 4 (figure 8.83).

– **Étape 4** : nous avons obtenu le temps de terminaison de la tâche $\boldsymbol{\tau_3}$. Nous pouvons alors obtenir le temps de réponse $\mathbf{TR_3}$ de cette instance de la tâche $\boldsymbol{\tau_3}$, soit :

$$TR_3 = a_3 = 63$$

– L'échéance de la tâche $\boldsymbol{\tau_3}$ est donc bien respectée : $\mathbf{TR_3} = 63 < \mathbf{D_3} = \mathbf{T_3} = 500$.

Cette analyse peut être effectuée pour l'ensemble des tâches de la configuration et conduire à la conclusion de l'ordonnançabilité de la configuration si tous les tests sont positifs.

Il est important de noter que toutes ces validations ne sont valables et applicables qu'en faisant l'hypothèse de durées de tâches fixes, connues et déterministes.

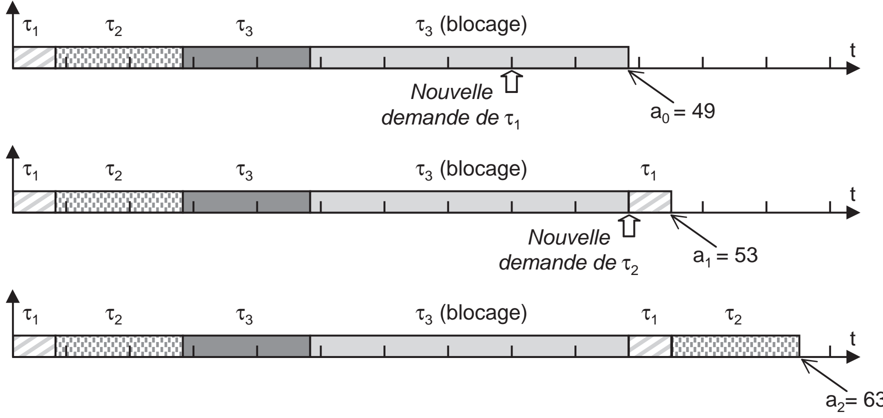


Figure 8.83 – Analyse d'ordonnançabilité selon la technique RMA basée sur l'évaluation du temps de réponse.

8.7 Ordonnancement en environnement multiprocesseur

8.7.1 Introduction générale

■ Définitions

Le système informatique de contrôle est souvent constitué de plusieurs équipements informatiques interconnectés par un réseau et ce pour les raisons suivantes :

– le procédé est, par nature, constitué d'**équipements multiples**, dotés chacun de son unité informatique : chaîne de fabrication ;

– la **sûreté de fonctionnement** conduit à multiplier les équipements de contrôle pour diminuer la défaillance de l'ensemble du procédé : systèmes embarqués (aéronautique…) ;

– les **contraintes de temps** conduisent à utiliser plusieurs systèmes informatiques pour une exécution en parallèle de certaines tâches : simulateur de vol…. Ce dernier cas peut conduire à des systèmes de type multiprocesseur.

La définition générale d'un système informatique multiprocesseur ou réparti pourrait être la suivante : « un ensemble de systèmes informatiques, appelés nœuds, reliés entre eux par un réseau de communication en vue d'échanger des informations ». Nous pouvons distinguer deux grandes catégories de systèmes informatiques répartis :

– Les **systèmes répartis à contrôle centralisé** ou **systèmes multiprocesseurs** ou encore systèmes fortement couplés qui présentent des analogies fortes avec les systèmes centralisés (monoprocesseur) en se différenciant essentiellement par la capacité à mettre en œuvre un parallélisme d'exécution:

  • base de temps commune (ordonnancement global des événements et des tâches) ;

© Dunod – La photocopie non autorisée est un délit.

- mémoire unique (vecteur de la communication entre les tâches) ;
- vue globale de l'état du système à chaque instant d'observation.

– Les **systèmes répartis ou distribués** (à contrôle décentralisé) ou systèmes faiblement couplés :
  - pas de mémoire commune et d'horloge commune
  - la synchronisation et la communication se font à l'aide d'échanges de messages à travers le réseau, ce qui amène des délais de transmission plus longs et moins déterministes.

La répartition sur plusieurs nœuds d'exécution induit de nombreuses difficultés dans les systèmes informatiques répartis :

– L'**ordonnancement** : les algorithmes optimaux classiques d'ordonnancement dans les systèmes monoprocesseurs ne s'appliquent plus dans un contexte multiprocesseur ou réparti où il existe des algorithmes d'ordonnancement spécifiques pour ces deux cas.

– **Allocation** ou **placement** ou encore **répartition des tâches** : à une certaine phase de la conception, l'application (tâches et données) doit être partitionnée à travers les différents nœuds du système pour assurer efficacement sa fonction.

– **Fiabilité** (tolérance aux fautes) : les systèmes répartis conduisent tout naturellement à une fiabilité accrue liée au fait que l'application est plus tolérante vis-à-vis des pannes processeurs. En revanche, le réseau (panne, surcharge, incohérence temporelle ou spatiale des données) amène une difficulté supplémentaire dans la réalisation d'applications à haut degré de sûreté de fonctionnement dont la fiabilité et la sécurité sont deux des éléments essentiels.

– **Autres difficultés pour les systèmes répartis** : synchronisation des processus entre les différents sites (horloge globale), validité des données (cohérence spatiale et temporelle), durées de communication, etc.

■ Synchronisation dans les systèmes distribués : horloge globale

L'ordonnancement en environnement distribué nécessite une synchronisation, c'est-à-dire une cadence ou une horloge identiques sur les différents sites. Les difficultés majeures à cette synchronisation sont les suivantes :

– les horloges des différents sites dérivent ;

– la durée de la transmission d'un message est inconnue.

La seule manière de synchroniser les sites est l'envoi de messages entre les sites. En revanche, il est nécessaire de trouver un compromis entre les deux options :

– envoyer de nombreux messages afin d'obtenir une meilleure synchronisation, mais un encombrement important du réseau ;

– envoyer un nombre restreint de messages conduisant à une synchronisation moins bonne.

De nombreuses recherches ont conduit à des résultats très intéressants. Il est possible de définir le nombre de messages à envoyer par chaque site afin d'obtenir une horloge avec une précision donnée sur l'ensemble des sites.

■ Placement des tâches dans les systèmes distribués

Le problème de placement ou d'allocation des tâches peut être résolu à partir des paramètres statiques de l'application, soit :

– **Caractéristiques statiques des tâches** : durée d'exécution, place mémoire, coûts de communications, interactions avec le procédé, importance…

– **Caractéristiques statiques de l'architecture opérationnelle** : nombre de nœuds, architecture interne des nœuds (mono ou multiprocesseur), vitesses de traitement des processeurs, caractéristiques du réseau de communication (délai de propagation, délai de transmission), etc.

La solution du placement des tâches sur les différents sites est déterminée à partir de la prise en compte de nombreux critères comme :

– minimiser le nombre de processeurs ;

– équilibrer la charge des processeurs ;

– minimiser la communication entre les nœuds ;

– prendre en compte les contraintes de résidence ;

– minimiser le temps de réponse d'une tâche ou d'un ensemble de tâches ou d'un site ;

– respecter le degré de redondance pour chaque tâche…

La solution ne peut être qu'un compromis par rapport à l'ensemble de ces critères. Une fois placées sur les différents sites, les tâches peuvent migrer entre les différents nœuds selon les besoins de l'application : surcharge d'un ou plusieurs sites, panne d'un ou plusieurs sites. Nous avons à traiter le processus de la migration dynamique qui peut être réalisée à chaque instance ou en cours d'exécution avec la migration du contexte de la tâche. Il est important de noter que la migration dynamique concerne un changement de site d'exécution car le code est installé sur plusieurs sites (figure 8.84).

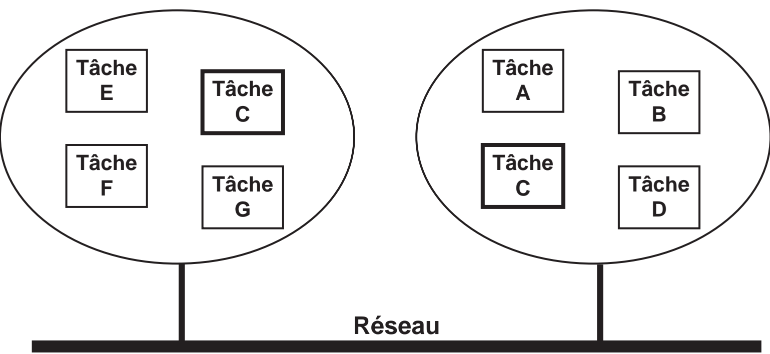


Figure 8.84 – Répartition des tâches sur deux sites : duplication de la tâche C qui peut donc s'exécuter sur les deux nœuds.

© Dunod – La photocopie non autorisée est un délit.

8.7.2 Ordonnancement dans les systèmes multiprocesseurs

Anomalie de fonctionnement

Dans le cadre de l'ordonnancement en environnement multiprocesseur, nous pouvons avoir un comportement semblable aux exécutions en environnement monoprocesseur en présence de ressources critiques, c'est-à-dire une anomalie de fonctionnement.

Prenons l'exemple d'une configuration de six tâches qui doivent s'exécuter sur deux processeurs identiques. Nous supposons que les tâches sont préemptibles et que la migration peut être effectuée uniquement en fin de tâche. Les caractéristiques temporelles sont données dans le tableau 8.30.

Tableau 8.30 – Configuration de six tâches exécutées sur deux processeurs identiques.

Tâche	r_i	C_i	D_i
τ_1	0	5	10
τ_2	0	**[2,6]**	10
τ_3	4	8	15
τ_4	0	10	20
τ_5	5	100	200
τ_6	7	2	22

Il est important de noter que, d'une part, les tâches sont à départ différé et, d'autre part, que la tâche $\boldsymbol{\tau_2}$ a une durée d'exécution qui peut varier de 2 à 6. Nous allons étudier l'exécution de cette configuration avec différentes valeurs de la durée de la tâche $\boldsymbol{\tau_2}$. Dans ce type de test, il est naturel de considérer les valeurs extrêmes du domaine de variation de la durée de la tâche $\boldsymbol{\tau_2}$. Ainsi, nous avons les séquences suivantes (figure 8.85) :

– cas I : durée d'exécution de la tâche $\boldsymbol{\tau_2}$ de 2. La séquence est valide ; mais nous pouvons constater que le phénomène d'inversion de priorité se produit lors de l'exécution de la tâche $\boldsymbol{\tau_5}$ qui s'exécute avant la tâche $\boldsymbol{\tau_4}$;

– cas II : durée d'exécution de la tâche $\boldsymbol{\tau_2}$ de 6. La séquence est aussi valide ; et nous pouvons constater que le phénomène d'inversion de priorité ne se produit pas ;

– cas III : durée d'exécution de la tâche $\boldsymbol{\tau_2}$ de 5. La séquence est valide ; et nous pouvons constater que nous obtenons les meilleurs temps de réponse pour les tâches $\boldsymbol{\tau_4}$ et $\boldsymbol{\tau_6}$.

Ces trois tests d'exécution semblent démontrer l'ordonnançabilité de la configuration. Les tests correspondent aux deux valeurs extrêmes et à une autre valeur inter-

médiaire de la durée d'exécution de la tâche τ_2 [2,6]. Un autre test est effectué avec une valeur de la durée d'exécution de la tâche τ_2 de 3, valeur faible dans l'intervalle de variation de la durée. La figure 8.86 montre que cette exécution conduit à une inversion de priorité et surtout aux dépassements des échéances pour les tâches τ_4 et τ_6. Nous pouvons ainsi constater que le comportement de l'exécution n'est pas logiquement lié aux paramètres temporels des tâches, comme leur durée.

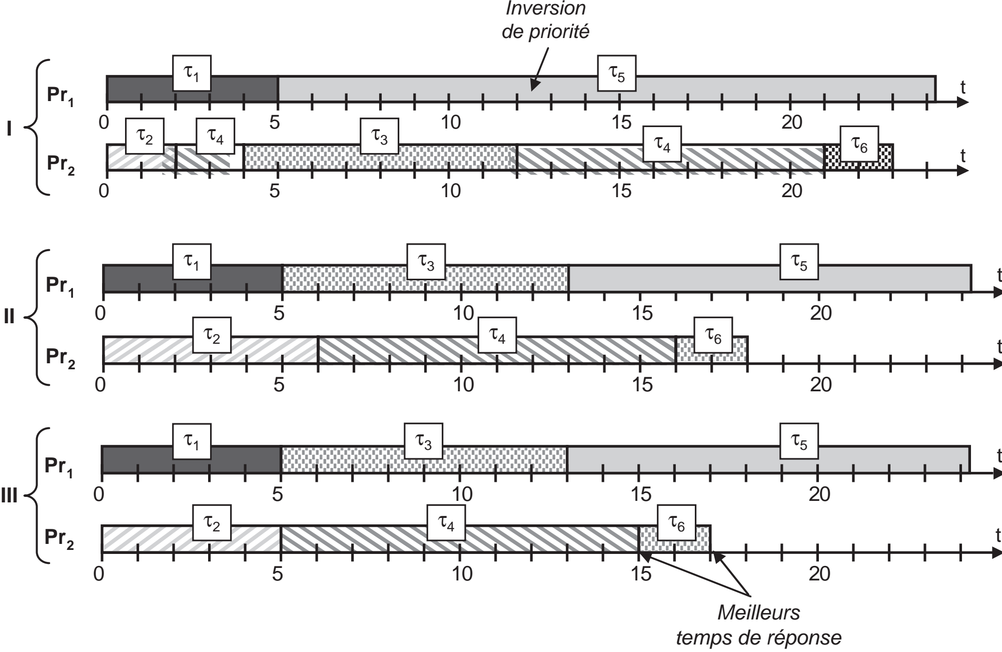


Figure 8.85 – Séquences d'exécution de la configuration, décrite dans le tableau 8.30, tracées pour trois durées différentes de la tâche τ_2 : 2 (cas I), 6 (cas II) et 5 (cas III).

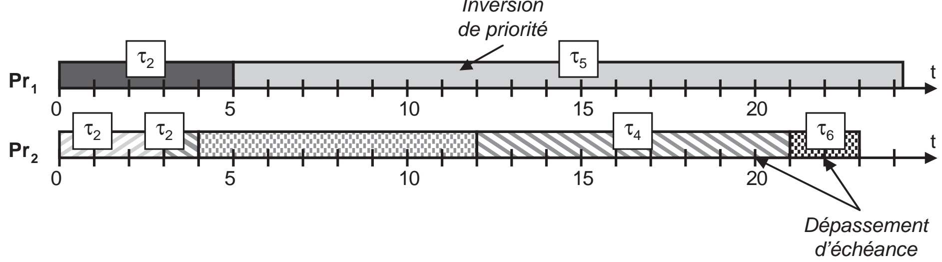


Figure 8.86 – Séquence d'exécution de la configuration, décrite dans le tableau 8.30, tracée pour une durée de la tâche τ_2 de 3.

© Dunod – La photocopie non autorisée est un délit.

Les algorithmes d'ordonnancement monoprocesseur en environnement multiprocesseur

Remarque

Il ne peut exister d'algorithme quasi en ligne qui construise une séquence valide sur une configuration matérielle comportant m ≥ 2 processeurs pour une configuration de tâches temps réel à date critique.

Cette remarque entraîne obligatoirement l'utilisation d'heuristiques ou d'utilisation de méthode d'analyse exhaustive. Ainsi, un algorithme hors ligne (résolution de problème d'optimisation de système linéaire) pourra traiter certaines configurations non résolues par un algorithme quasi en ligne ou en ligne.

Nous allons nous placer dans un environnement constitué de **m** processeurs avec une configuration de **n** tâches périodiques à exécuter, définies par les quatre paramètres temporels classiques : ($\mathbf{r_i}$,$\mathbf{C_i}$,$\mathbf{D_i}$,$\mathbf{T_i}$). Une première condition nécessaire d'ordonnançabilité de la configuration concerne le facteur d'utilisation, soit :

$$U = \sum_{i=1}^{n} C_i / T_i \leq m \qquad (8.54)$$

La propriété de cyclicité pour des tâches à départ simultané se retrouve pour un système multiprocesseur, ainsi la période d'étude est identique à celle présentée pour les systèmes monoprocesseurs :

$$H = PPCM\{T_i\}_{i \in [1,\, n]}$$

Nous pouvons effectuer une première analyse à l'aide des algorithmes optimaux et puissants étudiés en environnement monoprocesseur : EDF et ML. Considérons un exemple simple, constitué de trois tâches périodiques, décrit dans le tableau 8.31. Le facteur d'utilisation est **U** = 1,39 et la période d'étude **H** = 72. Cette configuration peut donc s'exécuter sur un environnement à deux processeurs en considérant la relation 8.54.

Nous allons supposer possible la migration en cours d'exécution. La figure 8.87 montre l'exécution de cette configuration avec l'algorithme d'ordonnancement de type EDF. La séquence n'est pas valide à cause du dépassement d'échéance de la tâche $\boldsymbol{\tau_1}$. La figure 8.88 montre la même exécution avec l'algorithme ML ; et, dans ce cas, la séquence d'exécution est valide.

Tableau 8.31 – Configuration de trois tâches à exécuter sur une plate-forme à deux processeurs.

Tâche	r_i	C_i	D_i	T_i
τ_1	0	8	9	9
τ_2	0	2	8	8
τ_3	0	2	8	8

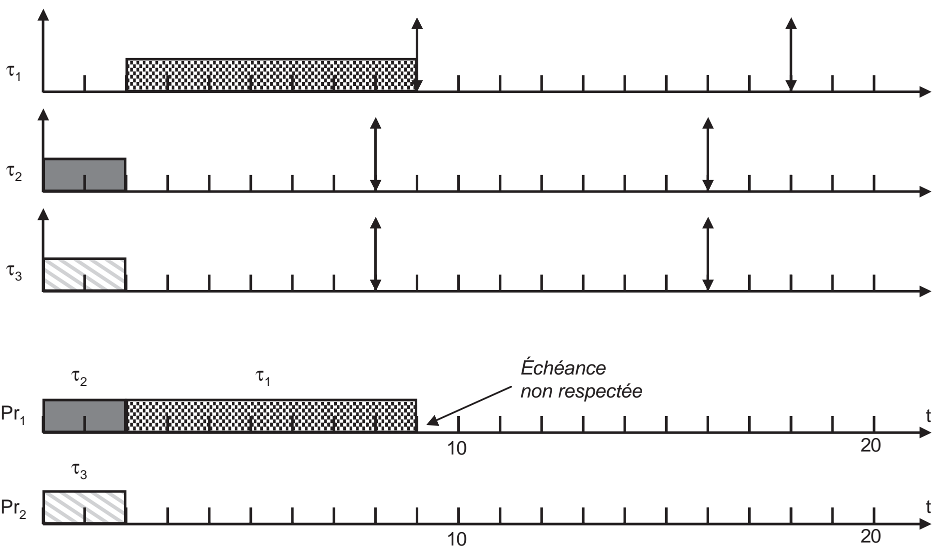


Figure 8.87 – Séquence d'exécution de la configuration décrite dans le tableau 8.31 avec l'algorithme EDF.

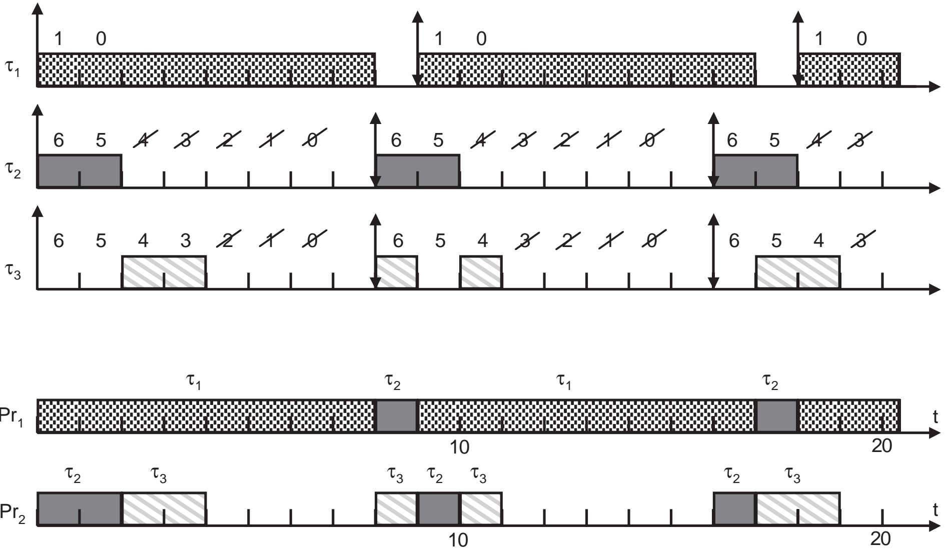


Figure 8.88 – Séquence d'exécution de la configuration décrite dans le tableau 8.31 avec l'algorithme ML.

© Dunod – La photocopie non autorisée est un délit.

À partir de cet exemple, nous pouvons faire les deux conclusions suivantes :

– Le comportement des algorithmes en environnement multiprocesseur ne peut pas être rapproché du comportement en environnement monoprocesseur, en particulier les propriétés ne sont plus vraies : optimalité…
– L'algorithme ML semble plus puissant que l'algorithme EDF dans un environnement multiprocesseur ; ils étaient identiques en environnement monoprocesseur.

■ Techniques d'ordonnancement en environnement multiprocesseur

De nombreuses techniques d'ordonnancement existent en environnement multiprocesseur. Ces techniques sont généralement utilisées pour des configurations possédant des propriétés particulières.

Nous allons seulement donner un exemple de ce type de technique basée sur le découpage temporel du temps processeur au prorata du facteur d'utilisation du processeur par chacune des tâches. Considérons une configuration de **n** tâches à échéance sur requête et à départ simultané à exécuter sur un environnement composé de **m** processeurs. Les tâches sont classées selon leur facteur d'utilisation du temps processeur, c'est-à-dire :

$$u_i \geq u_{i+1} \quad \text{pour} \quad i \in [1, n-1] \qquad (8.55)$$

Dans ce contexte, nous avons une condition nécessaire et suffisante d'ordonnançabilité qui est donnée par la relation suivante :

$$Max\left\{ Max\left\{ \frac{1}{j}\sum_{i=1}^{j} u_i \text{ pour } j \in [1, m] \right\}; \frac{1}{m}\sum_{i=1}^{n} u_i \right\} \leq 1 \qquad (8.56)$$

Cette expression un peu complexe cache une construction aisée de la séquence comme nous allons l'étudier sur un exemple précis. Étudions l'exemple de configuration de trois tâches décrite dans le tableau 8.32.

Tableau 8.32 – Configuration de trois tâches à exécuter sur une plate-forme à deux processeurs selon un algorithme basé sur la proportionnalité de l'affectation du temps processeur liée au facteur d'utilisation de chaque tâche.

Tâche	r_i	C_i	D_i	T_i	u_i
τ_1	0	2	3	3	0,66
τ_2	0	2	4	4	0,5
τ_3	0	3	6	6	0,5

Nous vérifions que cette configuration respecte bien le classement donné par la relation 8.55. D'autre part, le facteur d'utilisation globale (**U** = 1,66) indique qu'il est nécessaire de disposer d'une architecture au moins à deux processeurs pour

satisfaire la condition nécessaire 8.54. La période d'étude est **H** = 12. Enfin, nous pouvons tester que la condition nécessaire et suffisante 8.56 est bien satisfaite, soit :

$$Max\left\{Max\left\{\frac{2}{3};\frac{1}{2}\left(\frac{2}{3}+\frac{1}{2}\right)=\frac{7}{12}\right\};\frac{5}{6}\right\}=\frac{5}{6}\leq 1$$

Pour réaliser la construction de la séquence, il suffit de noter sur le diagramme temporel les instants de réveil des différentes tâches. Entre deux de ces instants, la durée processeur utile peut être divisée entre les différentes tâches au prorata des facteurs d'utilisation processeur $\mathbf{u_i}$ des différentes tâches. Ainsi, entre les deux premières dates de réveil, intervalle [0,3], la durée processeur disponible est de 3 unités de temps ; donc la répartition du temps processeur parmi les tâches est donnée dans le tableau 8.33. Il suffit ensuite d'affecter cette durée processeur sur chaque processeur comme le montre la figure 8.89. Nous trouvons ainsi un placement de l'exécution de toutes les tâches sur la période d'étude **H**. Nous pouvons remarquer que cette méthode de placement du travail du processeur par tranche conduit à mettre des temps libres tout au long de la séquence.

Tableau 8.33 – Calcul de l'affectation du temps processeur liée au facteur d'utilisation de chaque tâche pour le premier intervalle de la séquence d'exécution.

Tâche	u_i	Temps processeur dans l'intervalle [0,3]
τ_1	0,66	3*0,66=2
τ_2	0,5	3*0,5=1,5
τ_3	0,5	3*0,5=1,5

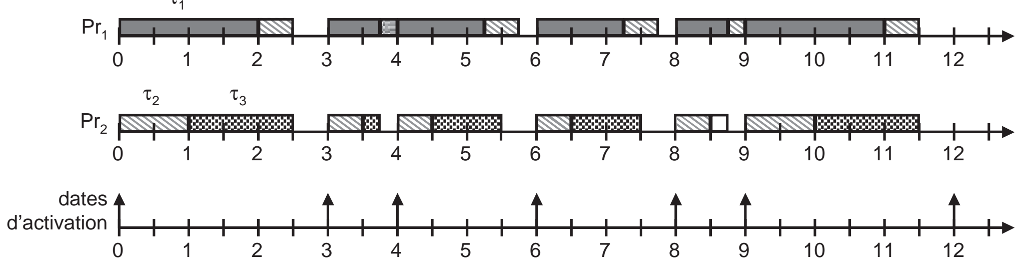


Figure 8.89 – Séquence d'exécution de la configuration décrite dans le tableau 8.32 avec un algorithme basé sur l'affectation du temps processeur liée au facteur d'utilisation.

© Dunod – La photocopie non autorisée est un délit.

8.7.3 Ordonnancement dans les systèmes distribués

Les systèmes répartis ou distribués (à contrôle décentralisé) ou systèmes faiblement couplés font deux hypothèses principales :

– pas de mémoire commune (pas de connaissance de l'état global du système) ;
– pas d'horloge commune (pas de datation globale des événements).

Dans ce contexte, deux problèmes cruciaux se posent :

– le contrôle global de l'application qui est résolu de façon efficace si le travail se fait sur la base d'une **coopération dynamique** (synchronisation et communication) entre les différents sites (et tâches) à l'aide d'échanges de messages à travers le réseau ;
– le délai de communication sur le réseau, fiabilité du réseau, pannes partielles ou totales (processeur ou réseau).

Pour répondre au problème de la coopération dynamique entre les sites, il est nécessaire de mettre en place un ordonnanceur réparti. Celui-ci doit pouvoir assurer les différentes fonctions suivantes :

– assurer un équilibrage des charges (pas de site surchargé) ;
– assurer un partage des charges (pas de site inactif) ;
– palier au dysfonctionnement (surcharge, dépassement d'échéance prévisible pour une tâche, occurrence de tâches sporadiques, panne d'un processeur, etc.).

La solution réside sur la migration de tâches après décision par communication entre les différents sites (les sites possédant des répliques des tâches). La décision de demande ou d'offre de migration va être faite en utilisant les différents services disponibles localement sur chacun des nœuds, soit :

– exploration des listes de tâches en attente ;
– utilisation de la fonction donnant les temps creux dynamiques ;
– conditions analytiques d'acceptation ;
– utilisation de la notion d'importance de la surcharge avec gestionnaire réparti.

Ainsi, au niveau de chaque site et en supplément de l'ordonnanceur local, nous trouvons un ordonnanceur réparti qui échange des messages avec les autres ordonnanceurs répartis des autres sites (figure 8.90).

Afin d'optimiser les performances de ces échanges de messages utiles uniquement à l'ordonnancement, il est nécessaire de trouver la meilleure stratégie. Nous supposons qu'un site A, dit site « demandeur », désire transférer une partie de sa charge vers d'autres sites B et C, dits sites « offreur ». Les principales politiques existantes pour répondre à cette question sont classées selon la provenance de la demande de migration, soit :

– À l'initiative du site demandeur :
  • **algorithme stable** : le site surchargé recherche site par site un appariement possible. La surcharge est mesurée selon le nombre de tâches en attente dans la liste des tâches activables ;

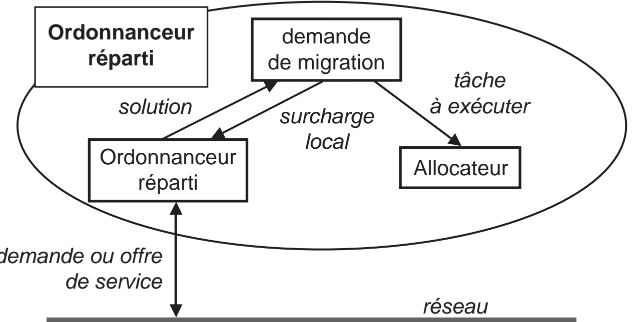


Figure 8.90 – Architecture des sites au niveau de l'agencement de l'ordonnancement.

- **algorithme aux enchères** : à la demande du site surchargé aux autres sites (demande par diffusion), les réponses concernent l'estimation de la charge des sites (notion de surplus) et sont considérées comme des enchères à la prise en compte de la demande d'aide : algorithme à 3 messages (figure 8.91).

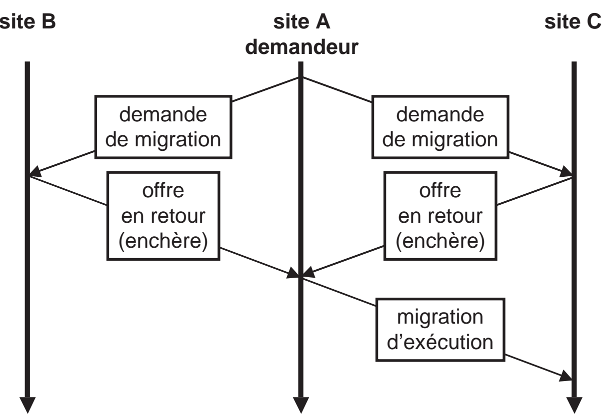


Figure 8.91 – Stratégie d'échanges de messages pour la migration des tâches à l'initiative du demandeur : algorithme aux enchères.

– À l'initiative du site offreur :

- **algorithme brouillon** : l'objectif de cette stratégie est de diminuer le nombre de phase de l'algorithme précédent « aux enchères » (politique à trois communications) à deux communications : proposition de migration et acceptation de la migration. Pour cela chaque site maintien à jour une information « surplus » qui est diffusée périodiquement de façon à ce que les tâches connaissent à un instant donné les tâches les moins chargées ;
- **algorithme du site hôte** : par rapport à l'algorithme précédent, cette stratégie considère un site hôte ou foyer qui est le site le moins chargé à la connaissance

© Dunod – La photocopie non autorisée est un délit.

du site demandeur. La demande de migration est donc envoyée préférentiellement à ce site (figure 8.92).

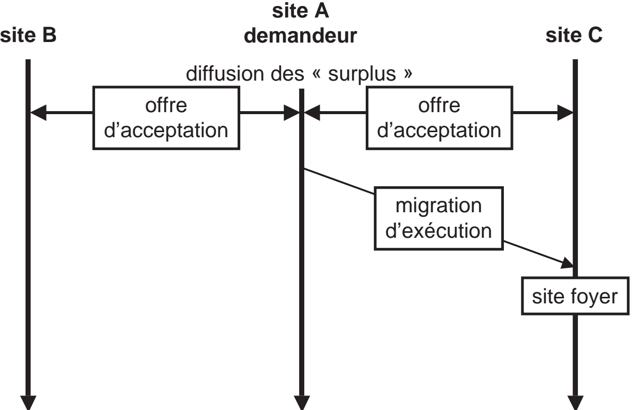


Figure 8.92 – Stratégie d'échanges de messages pour la migration des tâches à l'initiative du demandeur : algorithme du site hôte.

– À l'initiative des deux sites (demandeur et offreur) :

- **algorithme souple** : toujours dans le but de minimiser les messages échangés avant la migration effective, cette stratégie ajoute à « l'algorithme du site hôte » la diffusion simultanée de ce choix du site hôte particulier auquel doivent être renvoyées directement les enchères des autres sites (figure 8.93).

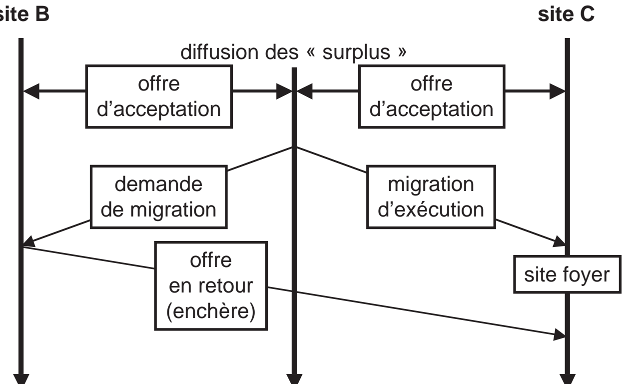


Figure 8.93 – Stratégie d'échanges de messages pour la migration des tâches à l'initiative du demandeur : algorithme souple.

Nous allons illustrer la mise en œuvre du dernier algorithme étudié (algorithme souple) avec un exemple simple. Considérons une application constituée de trois

tâches périodiques et deux tâches apériodiques à exécuter sur un environnement de trois sites. Les paramètres temporels sont donnés dans le tableau 8.34. La durée de communication entre deux sites est fixée à 2 quel que soit le message échangé.

Tableau 8.34 – Configuration de trois tâches périodiques et de deux tâches apériodiques à exécuter sur trois processeurs.

Tâche	r_i	C_i	D_i	T_i	Processeur
τ_1	0	2 ou 7	8	8	1
τ_2	0	3	8	8	2
τ_3	0	5	8	8	3
τ_4	0	2	8	–	1
τ_5	1	4	8	–	2

Nous allons tester trois cas avec cette configuration, soit :

– **Cas I** : la tâche τ_2 a une durée de 2 et la tâche τ_5 n'est prise en compte. La figure 8.94 montre les séquences obtenues sur les trois processeurs. Il n'y a aucune migration de tâche car le processeur 1 peut accepter la tâche τ_4.

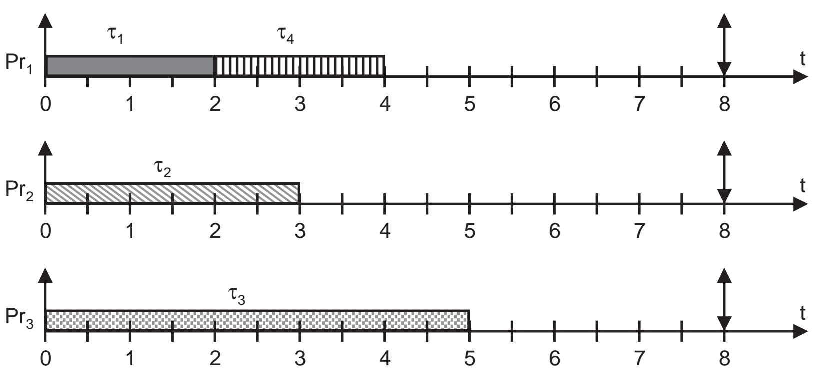


Figure 8.94 – Illustration du fonctionnement de l'algorithme souple pour la configuration décrite dans le tableau 8.34, cas I.

– **Cas II** : la tâche τ_2 a une durée de 7 et la tâche τ_5 n'est prise en compte. La figure 8.95 montre les séquences obtenues sur les trois processeurs. Le processeur 1 ne peut prendre en charge la tâche τ_4 avant son échéance. La migration s'effectue vers le processeur 2 qui est le processeur le moins chargé (site foyer). Une offre arrive au processeur 2 ; mais elle est inutile dans ce cas.

© Dunod – La photocopie non autorisée est un délit.

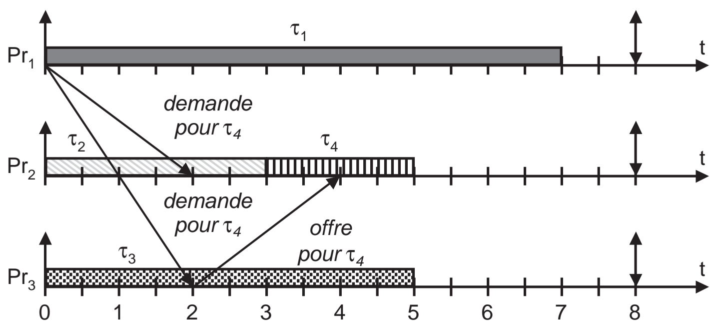


Figure 8.95 – Illustration du fonctionnement de l'algorithme souple pour la configuration décrite dans le tableau 8.34, cas II.

– **Cas III** : la tâche τ_2 a une durée de 7 et la tâche τ_5 est prise en compte. La figure 8.96 montre les séquences obtenues sur les trois processeurs. Le processeur 1 ne peut prendre en charge la tâche τ_4 avant son échéance. La migration s'effectue vers le processeur 2 qui le processeur le moins chargé au moment de la demande (site foyer). Mais le processeur 2 ne peut pas prendre en charge la tâche τ_4 à cause de la tâche τ_5. L'offre, arrivée au processeur 2 par le processeur 3, est acceptée et renvoyée vers le processeur 3 qui exécute la tâche τ_4.

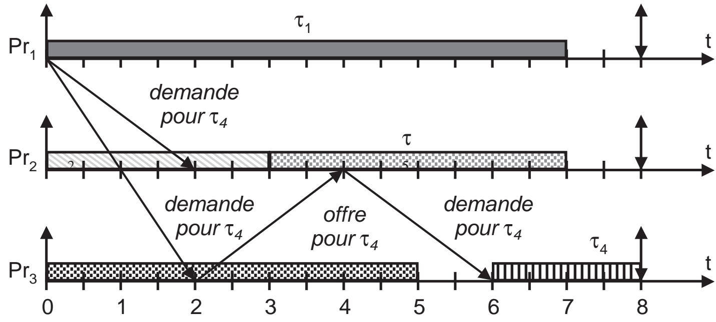


Figure 8.96 – Illustration du fonctionnement de l'algorithme souple pour la configuration décrite dans le tableau 8.34, cas III.

8.7.4 Validation des systèmes distribués

Dans la section précédente nous avons analysé des configurations dont les seuls paramètres définis étaient les sites et les tâches avec leurs paramètres temporels. L'ordonnanceur local doit traiter avec les autres sites en cas de surcharge ou de panne. Mais, de manière courante, les applications temps réel embarquées et distribuées sont complètement déterminées au niveau de la communication des messages. Nous allons nous placer dans ce contexte, où sites, tâches et messages sont connus, pour conduire l'analyse de l'application afin de la valider temporellement.

Anomalie en environnement distribué

Lorsque le système est distribué, un réseau informatique est l'unique moyen de communication entre les processeurs et constitue ainsi une ressource partagée par les tâches. La transmission des messages sur un réseau va conduire au même phénomène d'inversion de priorité puisque la transmission d'un message ne peut pas être interrompue par une autre émission, le médium de transmission étant physiquement occupé. Ainsi, pour illustrer la même anomalie que nous avons analysée en environnement monoprocesseur avec ressources critiques, considérons le cas de deux sites intégrant chacun deux tâches : site 1 (tâches $\boldsymbol{\tau_1}$ et $\boldsymbol{\tau_4}$) et site 2 (tâches $\boldsymbol{\tau_2}$ et $\boldsymbol{\tau_3}$). La communication entre les deux sites est composée de deux messages : m_1 de $\boldsymbol{\tau_1}$ vers $\boldsymbol{\tau_3}$, et m_2 de $\boldsymbol{\tau_2}$ vers $\boldsymbol{\tau_4}$ (figure 8.97). L'ordonnancement des deux sites est conduit par les échéances $\mathbf{d_i}$ des tâches ; ainsi nous avons un ordre de priorité d'ordonnancement sur le site 1 ($\boldsymbol{\tau_1}$ et ensuite $\boldsymbol{\tau_4}$) et sur le site 2 ($\boldsymbol{\tau_2}$ et ensuite $\boldsymbol{\tau_3}$).

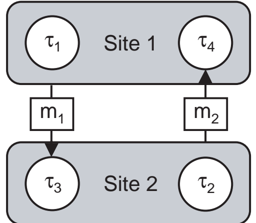


Figure 8.97 – Modélisation d'une application distribuée définie par deux sites, deux tâches sur chacun des sites et deux messages échangés.

La figure 8.98 montre deux exécutions possibles de cette configuration selon la durée de la tâche $\boldsymbol{\tau_2}$ du site 2. Ainsi, nous avons :

– Cas I (la durée de la tâche $\boldsymbol{\tau_2}$ est supérieure à celle de la tâche $\boldsymbol{\tau_1}$). La tâche $\boldsymbol{\tau_1}$ se termine donc la première et par conséquent envoie le message $\mathbf{m_1}$ sur le réseau. Ensuite, la tâche $\boldsymbol{\tau_2}$ se termine, envoie le message $\mathbf{m_2}$ lors de la libération du réseau par le message $\mathbf{m_1}$. Cela oblige la tâche $\boldsymbol{\tau_4}$ à attendre ; mais son échéance étant lointaine, les séquences d'ordonnancement sur les deux sites sont valides.

– Cas II (la durée de la tâche $\boldsymbol{\tau_2}$ est inférieure à celle de la tâche $\boldsymbol{\tau_1}$). La tâche $\boldsymbol{\tau_2}$ se termine donc la première et par conséquent envoie le message $\mathbf{m_2}$ sur le réseau. Ensuite, la tâche $\boldsymbol{\tau_1}$ se termine, envoie le message $\mathbf{m_1}$ lors de la libération du réseau par le message $\mathbf{m_2}$. Cela conduit la tâche $\boldsymbol{\tau_3}$ à attendre et à dépasser son échéance. La séquence d'ordonnancement sur le site 1 est valide mais pas la séquence sur le site 2 ; donc, d'une façon générale, l'application n'est pas valide.

En conclusion, comme pour les systèmes monoprocesseurs avec ressources critiques ou les systèmes multiprocesseur, nous pouvons ainsi constater que le comportement de l'exécution n'est pas logiquement lié aux paramètres temporels des tâches, comme leur durée. Par conséquent nous allons nous placer dans un contexte où tous les paramètres temporels des tâches et des messages sont connus et fixés *a priori*, et, avec ces hypothèses, faire une analyse du pire cas.

© Dunod – La photocopie non autorisée est un délit.

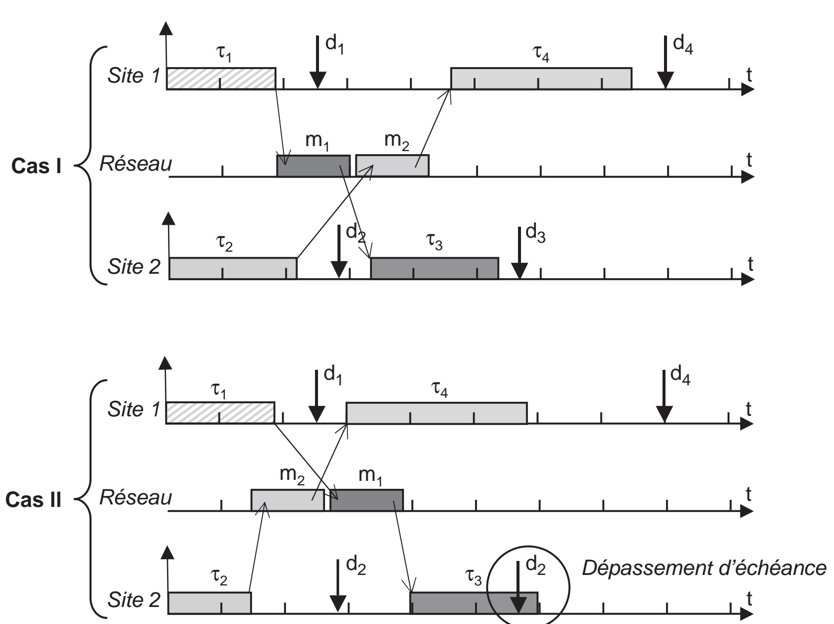


Figure 8.98 – Illustration d'une anomalie de fonctionnement dans le cas des applications distribuées.

■ Analyse holistique des systèmes distribués

Les principales hypothèses que nécessite cette analyse temporelle des systèmes distribués sont les suivantes :

– horloge globale (utilisation d'un moyen externe ou par envoi de messages) ;
– réseau fiable (pas de perte de messages) ;
– tâches à départ simultané au démarrage du système ;
– pas de migration des tâches dans le système ;
– messages lus au début de l'exécution des tâches ;
– messages émis à la fin d'exécution des tâches.

L'ordonnancement conjoint des tâches et des messages conduit à considérer que la date d'émission d'un message dépend du temps de réponse de la tâche émettrice et la date de réveil de la tâche réceptrice dépend du temps de transmission ou temps de réponse du message. Ce décalage temporel imposé par la précédence d'une tâche ou d'un message est appelé **gigue** (ne pas confondre avec la gigue comme paramètres temporels d'analyse présentée au § 8.2). Ainsi, le message émis $\mathbf{m_{1\text{-}2}}$ et la tâche réceptrice $\boldsymbol{\tau_2}$ sont obligatoirement décalés des gigues respectives $\mathbf{J_m}$ et $\mathbf{J_2}$ par rapport au début de l'exécution de la tâche émettrice $\boldsymbol{\tau_1}$ (figure 8.99).

Soit un ensemble de tâches trié par ordre de priorité sur chacun des sites (la tâche $\boldsymbol{\tau_i}$ plus prioritaire que la tâche $\boldsymbol{\tau_j}$ si $\mathbf{i<j}$: $\mathbf{Prio_i>Prio_j}$). Nous supposons que l'algorithme choisi est basé sur une affectation de priorité fixe. La technique de calcul du

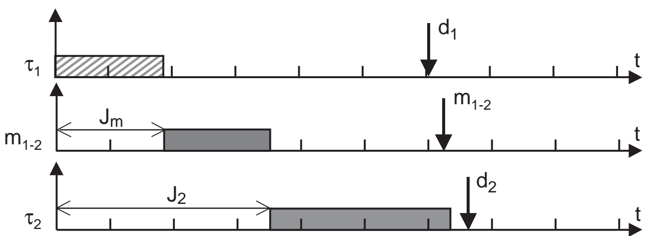


Figure 8.99 – Illustration du décalage temporel ou gigue d'un message et de la tâche réceptrice.

pire temps de réponse des tâches est basée sur le même principe que celui que nous avons utilisé dans l'analyse RMA (§ 8.6.2). Pour effectuer ce calcul, il est nécessaire de définir le temps pendant lequel le processeur est occupé à exécuter des tâches de priorité supérieure ou égale à $\mathbf{Prio_i}$ (période d'activité de niveau **i**).

Par rapport au calcul des temps de réponse dans l'analyse RMA, les équations 8.50 et 8.51 vont être augmentées de la gigue initiale que peuvent avoir les tâches étant donné le pire temps d'attente de message sur le réseau. Ainsi, les équations à résoudre sont :

$$t_0 = \frac{(q+1)C_i + B_i + \sum_{j=1}^{i-1} \frac{J_j}{T_j} C_j}{1 - \sum_{j=1}^{i-1} \frac{C_j}{T_j}} \qquad (8.57)$$

et

$$t_{n+1} = B_i + (q+1)C_i + \sum_{j=1}^{i-1} \left\lceil \frac{J_j + t_n}{T_j} \right\rceil C_j \qquad (8.58)$$

Le point fixe $\mathbf{t_{n+1} = t_n}$ est recherché par une méthode itérative. Dans une période d'activité, la charge minimale est définie par l'exécution d'une instance de chaque tâche. Sous cette forme, q est un paramètre. Chaque incrémentation de q revient à explorer une période supplémentaire de la tâche $\boldsymbol{\tau_i}$, appartenant à la période d'activité de niveau i. Le principe de l'algorithme va consister à incrémenter q tant que le test d'arrêt n'est pas vérifié : le temps de réponse de l'instance courante est inférieur ou égal à la prochaine date de réveil de la tâche. La borne du pire temps de réponse peut alors être évaluée par :

$$TR_i = Max(t + J_i - kT_i) \qquad (8.59)$$

Un exemple d'exploration de cette zone d'activité de niveau **i** est présenté sur la figure 8.100 en ne prenant pas en compte la gigue et le temps de blocage. La tâche $\boldsymbol{\tau_i}$ est exécutée selon quatre instances dans cette zone d'activité. Remarquons que, dans

© Dunod – La photocopie non autorisée est un délit.

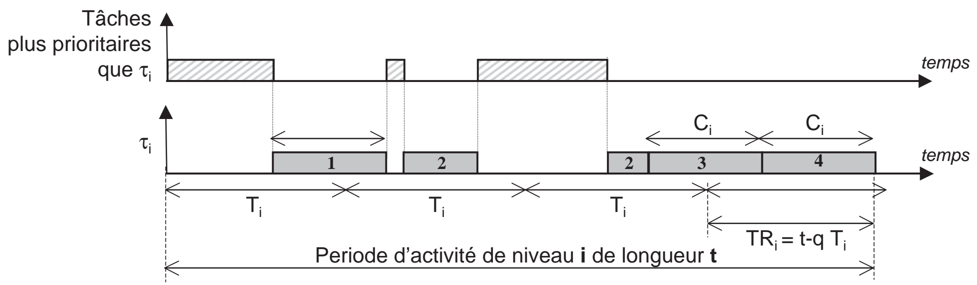


Figure 8.100 – Exemple du calcul de la longueur d'une zone d'activité de niveau **i**.

cette figure, la tâche τ_i n'est pas à échéance sur requête ; en revanche, la fin de la période d'activité se situe avant la fin de la quatrième période de la tâche τ_i.
Un calcul identique peut être réalisé dans le cas d'un algorithme à priorité dynamique (EDF) conduisant à des équations de la forme de celles obtenues précédemment.
Dans le cas qui nous intéresse où nous avons une application distribuée avec un réseau, il est nécessaire de calculer le pire temps de réponse des messages de la même manière que pour les tâches ordonnancées en environnement à priorité fixe. La différence essentielle réside dans le fait que la préemption n'est plus autorisée puisqu'un message envoyé occupe le réseau jusqu'à sa fin. Ainsi, en considérant que les messages sont classés par ordre de priorité comme les tâches (le message $\mathbf{m_i}$ de durée $\mathbf{C_i}$ plus prioritaire que la tâche $\mathbf{m_j}$ de durée $\mathbf{C_j}$ si $\mathbf{i<j}$), nous avons pour **m** messages la relation suivante :

$$t_{n+1} = (q+1)C_i + \sum_{j=1}^{i-1}\left(\left\lceil\frac{J_j + t_n}{T_j}\right\rceil + 1\right)C_j + \max_{i \le k \le m}(C_k) \qquad (8.60)$$

Et le pire temps de réponse du message $\mathbf{m_i}$:

$$TR_i = Max(t + J_i - kT_i) \qquad (8.61)$$

La durée $\mathbf{C_i}$ d'un message $\mathbf{m_i}$ est calculée selon le type de protocole avec les informations données dans le chapitre 4.
La mise en œuvre de l'ensemble de ces calculs nécessite dans la plupart des cas des calculs complexes. Prenons un exemple très simple pour illustrer cette validation des applications distribuées par évaluation des pires temps de réponse. Soit une application distribuée constituée de trois tâches périodiques indépendantes réparties sur deux processeurs avec un seul message échangé (figure 8.101 et tableau 8.35).
Le site 1 est ordonnancé avec une politique RM, c'est-à-dire que la tâche τ_1 est plus prioritaire que la tâche τ_2. Remarquons que la tâche τ_3, liée à la tâche τ_2 par le message $\mathbf{m_1}$, possède une période identique à la tâche τ_2. Nous allons calculer la gigue du message $\mathbf{m_1}$ et ensuite la gigue de la tâche τ_3. Pour un calcul aussi simple, il n'est pas nécessaire d'utiliser les équations précédentes. Ainsi, nous commençons par le site 1 ; la figure 8.102 montre que la tâche τ_2 est préemptée par la deuxième instance de la tâche τ_1. Par conséquent, la gigue du message $\mathbf{m_1}$ est donnée par :

$$J_{m_1} = C_2 + 2 \cdot C_1 = 80 + 60 = 140$$

Étant donné que dans cet exemple simple il n'y a qu'un seul message, celui-ci ne peut pas être retardé donc la gigue de la tâche $\boldsymbol{\tau_3}$ du deuxième site est :

$$J_3 = J_{m_1} + C_{m_1} = C_2 + 2 \cdot C_1 + C_{m_1}$$
$$= 80 + 60 + 20 = 160$$

Le pire temps de réponse de la tâche $\boldsymbol{\tau_3}$ sera donc inférieur à son échéance qui est égale à la période :

$$TR_3 = J_3 + C_3 = 160 + 10 = 170 < d_3 = T_3 = 200$$

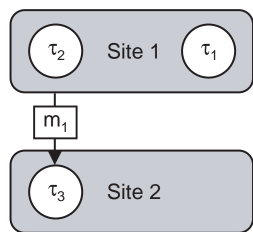


Figure 8.101 – Exemple d'une application distribuée définie sur deux sites avec trois tâches et un message échangé.

Tableau 8.35 – Configuration de trois tâches périodiques indépendantes à exécuter sur deux processeurs.

Tâche	r_i	C_i	D_i	T_i	Processeur
τ_1	0	30	100	100	1
τ_2	0	80	200	200	1
τ_3	0	10	200	200	2
m_1	0	20	–	–	–

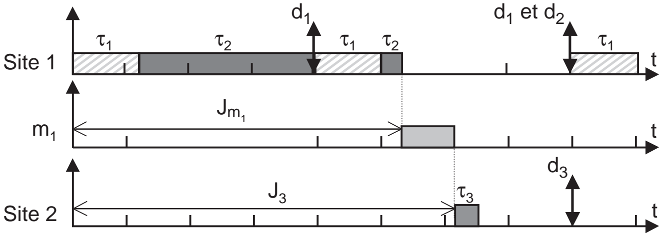


Figure 8.102 – Séquence d'exécution de l'une application décrite dans le tableau 8.35.

© Dunod – La photocopie non autorisée est un délit.

Annexes

A • REPRÉSENTATION DE L'INFORMATION

Cette annexe présente les méthodes utilisées pour la représentation des nombres entiers signés et des nombres fractionnaires à virgule fixe et à virgule flottante.

A.1 Représentation binaire des entiers signés

Pour un entier signé (*i.e.* pouvant prendre des valeurs négatives) sur n bits, une représentation naïve utiliserait un bit de signe (par exemple le bit de poids fort), et les $n-1$ bits restants pourraient permettre de représenter la valeur absolue du nombre. Cependant, le 0 aurait deux représentations (+ 0, et – 0), et les opérations arithmétiques nécessiteraient des circuits plus élaborés qu'ils ne le sont. Par exemple, il serait nécessaire dans les processeurs de prévoir l'addition et la soustraction, de tester les signes des opérandes avant de choisir l'opération à effectuer, de plus, un test à 0 nécessiterait de comparer un nombre avec deux valeurs (ou bien de restreindre la comparaison aux bits de la valeur absolue), etc. À cause de tous ces inconvénients, la représentation des entiers signés en signe et valeur absolue n'a été que très peu utilisée dans la réalité. C'est la notion de complément qui est utilisée à la place. Le complément à 1 d'un nombre binaire consiste simplement à inverser tous les bits d'un nombre donné en valeur absolue, ainsi, le nombre 9 sur 8 bits vaut b00001001. Son complément à 1 vaut b11110110 (chaque chiffre du nombre se voit remplacer par la différence entre lui et 1, soit, en binaire, remplacement des 1 par 0 et des 0 par 1). Supposons que l'on représente – 9 par le complément à 1 de 9, et additionnons 9 + – 9 : b00001001 + b11110110 = b11111111. On obtient le complément de 0, soit – 0, dans cette représentation, 0 a donc deux représentations (+ 0 et – 0). Or, la propriété d'une représentation bornée est qu'en ajoutant 1 à – 0, on obtient + 0 avec une retenue qui déborde de la représentation. L'idée, afin d'éviter le problème de la double représentation du 0, est d'ajouter 1 au complément d'un nombre pour obtenir son opposé. Cela s'appelle le complément à 2 (complément à 1 + 1). Ainsi, le complément à 2 de 9, qui sert à représenter – 9, est b11110110 + b1 = b11110111. Si l'on additionne sur 8 bits 9 et – 9 en complément à 2, on obtient alors b00001001 + b11110111 = b00000000 plus une retenue sur le 9e bit, qui est ignorée puisque débordant de la représentation. Il est primordial de considérer la taille de la représentation pour cette technique. On peut noter que le complément à 2 de 0 est 0 (représentation unique du 0). Chaque nombre a une représentation unique en complément à 2.

© Dunod – La photocopie non autorisée est un délit.

Le tableau A.1 montre les différentes représentations possibles des entiers signés sur 8 bits.

Tableau A.1 – Représentation des entiers signés (signe et valeur absolue, complément à 1, complément à 2) sur 8 bits.

Valeur décimale	Signe et valeur absolue	Complément à 1	Complément à 2
127	01111111	01111111	01111111
126	01111110	01111110	01111110
...	...	...	...
1	00000001	00000001	00000001
0	00000000	00000000	00000000
– 0	10000000	11111111	
– 1	10000001	11111110	11111111
...	...	...	...
– 126	11111110	10000001	10000010
– 127	11111111	10000000	10000001
– 128			10000000

Remarquons qu'il est simple de savoir en représentation des entiers en complément à 2 si le nombre est négatif : en effet, le bit de poids fort (le bit le plus à gauche) vaut 0 si l'entier est positif, et 1 si l'entier est négatif, comme pour la représentation en signe et valeur absolue. L'obtention de la valeur absolue d'un nombre négatif (dont le bit de poids fort vaut 1) se fait donc en retranchant 1 et en complémentant. Ainsi, le nombre b11110011 est négatif (bit de poids fort à 1), sa valeur absolue est le complément de b11110011 – b1 = b11110010, soit b00001101, c'est-à-dire 13. b11110011 est donc la représentation en complément à 2 de – 13. La seule exception concerne le plus petit nombre négatif (voir tableau A.1), puisque le domaine des entiers signés n'est pas symétrique (à cause du + 1 du complément à 2) : $[-2^{n-1}..2^{n-1}-1]$. Ainsi, le domaine est partitionné en 2, mais le 0 est représenté par + 0.
On peut remarquer que l'addition et la soustraction sont faites exactement de la même façon : effectuer $a - b$ consiste à effectuer a + *complément_à_2(b)*.
Il faut retenir que pour les entiers signés, la représentation la plus communément utilisée dans les processeurs est le complément à 2. Lors du choix d'un type de représentation pour une variable, il est primordial de savoir si l'on choisit une représen-

tation signée ou non signée. Par exemple, si en langage C on utilise un type caractère signé (char) il faut avoir conscience que son domaine de variation est [– 128..127] et non [0..255].

Afin d'illustrer la technique du complément, la figure A.1 présente son application décimale sur 8 chiffres : le complément à 10 (complément à 9+1). Notons cependant que dans ce cas, le chiffre de poids fort ne peut prendre que les valeurs 0 et 9 pour « + » et « – ».

```
Complément 9 de -352 sur 4 chiffres = 9647
Complément 10 (9+1) de -352         = 9648

 111
  0456
 +9648
 -----
  0104
```

La dernière retenue (celle provoquée par le signe) est ignorée

Figure A.1 – Addition des nombres décimaux 456 et –352 en complément à 10.

A.2 Représentation des nombres fractionnaires

Pour clore le chapitre de la représentation de l'information, il convient d'étudier succinctement le codage binaire des nombres fractionnaires. Ceci afin de démontrer la complexité des opérations sur ce type de nombres (la plupart des microprocesseurs possèdent une unité spécialisée dans ce type de calculs, alors que peu de microcontrôleurs en ont une), et avertir le lecteur sur les dangers inhérents à l'approximation de nombres réels par une représentation discrète et bornée.

Il existe deux types de représentation de nombres fractionnaires : la représentation en point fixe, et la représentation en virgule flottante.

A.2.1 Nombres à virgule fixe

Lorsque l'on connaît le domaine de variations des nombres manipulés, et la précision désirée, il est possible de définir un type **point fixe**, caractérisé par le plus petit nombre représentable appelé *delta* (généralement il s'agit d'une puissance négative de 2, comme 2^{-3} par exemple). Ce sera la puissance de 2 associée au bit de poids faible. Ensuite, comme illustré sur la figure A.2, les puissances vont croissantes comme pour les nombres binaires classiques. Généralement, les nombres en point fixe sont représentés sous la forme signe et valeur absolue. L'intervalle de représentation d'un nombre point fixe signé représenté sur n bits va de $-2^{n-1}*delta+delta$ à $+ 2^{n-1}*delta-delta$ (le n–1 est dû au fait que l'on perd le bit de signe pour la représentation de la valeur absolue), et sa précision est de *delta* (*i.e.* il existe un intervalle de *delta* entre deux nombres représentables consécutifs). On peut remarquer que la représentation point fixe en signe et valeur absolue souffre d'une double représentation du 0.

© Dunod – La photocopie non autorisée est un délit.

	delta.2^{n-2}		delta.2^2	delta.2^1	delta.2^0
signe		…			

Sur 8 bits, pour un delta valant 2^{-3}, représentation de 10,375

sgn	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
1	1	0	1	0	0	1	1

Nombres représentables : –15,875 à 15,875 avec un pas de 0,125

Figure A.2 – Représentation d'un nombre fractionnaire en point fixe.

En fait, la représentation point fixe correspond à fixer la virgule dans le nombre rationnel. En effet, le chiffre correspondant à 2^0 correspond au chiffre des unités, et les chiffres qui correspondent aux puissances négatives de 2 correspondent aux chiffres qui suivent la virgule.

Il est à noter que certaines représentations en type point fixe utilisent le complément à 2 pour représenter les nombres signés. Par conséquent, l'utilisation du type point fixe est relativement efficace, puisque l'on peut utiliser tels quels les circuits arithmétiques du processeur : deux nombres donnés en point fixe (sur le même *delta*) ont exactement les mêmes propriétés arithmétiques que les entiers.

A.2.2 Nombres à virgule flottante

Lorsque l'on n'a pas d'idée précise du domaine de variations des nombres manipulés, on utilise plutôt une représentation à virgule flottante. La représentation communément utilisée a été normalisée sous la norme IEEE 754. L'idée de base est d'utiliser des points fixes par intervalles de valeurs : si le nombre est grand, on positionne la virgule de sorte que le *delta* soit grand, si le nombre est petit, on positionne la virgule de façon à avoir un *delta* petit. Bien évidemment, la précision de la représentation change avec le *delta*.

L'idée de base est la suivante (figure A.3) : un nombre est (sauf exceptions présentées après) représenté en forme normalisée, c'est-à-dire sous la forme $signe * 1{,}mantisse * 2^{exposant}$. Or en binaire, il est inutile de représenter le bit toujours à 1 avant la virgule. Ce bit n'est donc pas représenté pour un flottant normalisé : il est appelé bit caché. L'exposant ne se représente pas à l'aide d'un signe et d'une valeur absolue, ni même d'une représentation des négatifs en complément, mais avec un biais. Le biais consiste à scinder les 2^n valeurs possibles de n bits en décalant du biais l'intervalle $[0..2^n - 1]$. Cela revient donc à considérer que les valeurs possibles de l'exposant sont comprises dans $[- biais..2^n - 1 - biais]$. Par exemple, pour un exposant donné sur 8 bits avec un biais de 127, les valeurs possibles de l'exposant se situent dans [– 127..128] (alors que la représentation de l'exposant varie évidemment dans [0..255]).

Afin d'affiner la précision autour du 0, la norme IEEE 754 propose certains aménagements récapitulés dans le tableau A.2. On pourra noter une représentation

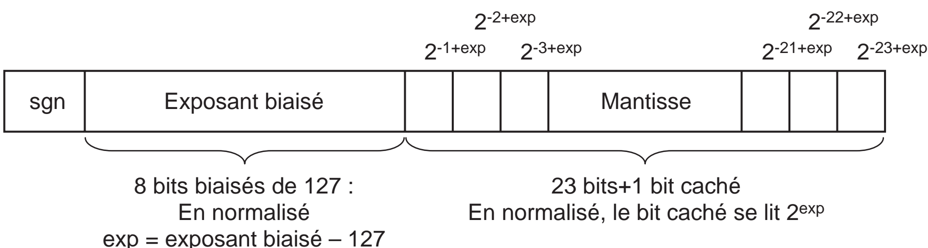


Figure A.3 – Représentation d'un nombre à virgule flottante au format IEEE 754 sur 32 bits.

dénormalisée lorsque l'exposant est égal à – *biais* (représentation à 0 de la partie « exposant biaisé » du flottant).

Tableau A.2 – Sémantiques spécifiques du format IEEE 754.

Nombre	Caractérisation	
0	exposant biaisé = 0 mantisse = 0	
•	exposant biaisé = plus grand nombre représentable (255 en 32 bits) mantisse = 0	
NaN (*Not a Number*)	exposant = plus grand nombre représentable (255 en 32 bits) mantisse ≠ 0	
Nombre dénormalisé	exposant biaisé = 0 mantisse ≠ 0	Le bit caché vaut 0, et l'exposant vaut – 126 (– biais + 1)

La figure A.4 montre que les flottants correspondent à des points fixes par partie, avec autant de parties que d'exposants possibles.

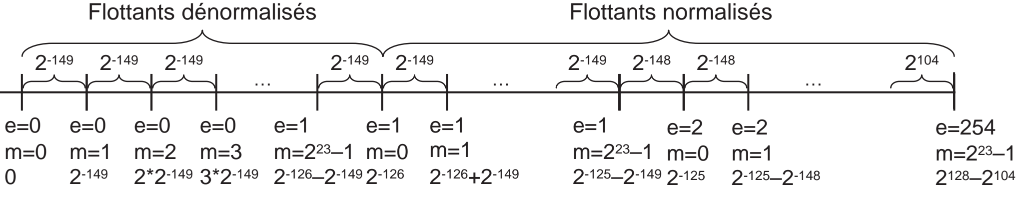


Figure A.4 – Domaines et précisions des flottants IEEE 754 sur 32 bits.

© Dunod – La photocopie non autorisée est un délit.

Afin d'illustrer la complexité des opérations arithmétiques sur les flottants, citons la prise en compte des sémantiques différentes suivant la représentation (infini, « *Not a Number* », normalisé ou dénormalisé), citons aussi la complexité, relativement aux opérations entières, d'une opération basique, telle l'addition de deux flottants. Afin d'additionner deux nombres flottants, il faut :

– mettre les deux nombres au même exposant, ce qui peut faire apparaître le bit caché dans le flottant de plus petit exposant, au détriment des bits de poids faible qui peuvent être perdus (le flottant peut être arrondi).
– L'addition des mantisses peut alors être réalisée.
– Le flottant obtenu est alors renormalisé (si l'exposant le permet).

Afin d'optimiser ces opérations, les microprocesseurs s'adjoignent souvent un coprocesseur spécialisé dans les calculs à virgule flottante. Pour les microcontrôleurs, cela est moins fréquent. Un concepteur souhaitant réaliser une régulation, par PID (Proportionnelle, Intégrale, Dérivée) par exemple, devra donc être conscient du fait que s'il manipule des nombres flottants, et qu'il ne possède pas de coprocesseur spécialisé, chaque opération arithmétique flottante nécessite de nombreuses instructions et est donc coûteuse en nombre de cycles processeur.
Enfin, il est indispensable d'avoir conscience que l'arithmétique en nombres flottants perd de nombreuses propriétés arithmétiques classiques sur les nombres réels. Par exemple, l'addition n'est pas associative (*i.e.* en fonction des ordres de grandeur de trois flottants a, b, et c, on peut observer $(a+b)+c \neq a+(b+c)$), cela est dû aux arrondis lors des mises au même exposant. Il peut alors arriver, par exemple lors de l'inversion d'une matrice de façon numérique, que les valeurs obtenues soient significativement différentes de celles que l'on aurait obtenues à l'aide d'un calcul symbolique.

B • STANDARDS POSIX

Nom IEEE	Nom	Notes
1003.0	Guide de l'environnement d'un système d'exploitation POSIX (*Guide to POSIX Operating Systems Environment*)	Abandonné en 2002.
1003.1	Interface système (*System Interface*)	Définit l'interface de programmation système d'un système d'exploitation POSIX – Dernière version 2004. Depuis 2001, intègre différents amendements donnés ci-après.
1003.1a	Services systèmes additionnels (*Additional System Services*)	Intégrée dans la 1003.1 depuis 2001.
1003.1b	Extensions temps réel (*Realtime Extensions*)	Dernière version 1993, intégrée dans 1003.1 depuis 2001.
1003.1c	Tâches (*Threads*)	Dernière version 1995, intégrée dans 1003.1 depuis 2001.
1003.1d	Extensions temps réel additionnelles (*Additional Realtime Extensions*)	Dernière version 1999.
1003.1e	Sécurité (*Security*)	Abandonné en 1998.
1003.1f	Accès transparent aux fichiers (*Transparent File Access*)	Abandonné en 1997.
1003.1g	Interfaces indépendantes des protocoles (*Protocol Independant Interfaces*)	Dernière version 2000.
1003.1h	Tolérance aux fautes (*Fault Tolerance*)	Dernière version 2000 – Devenu 1003.25.
1003.1i	Corrections aux extensions temps réel (Fixes to 1003.1b)	Dernière version 1995, intégrée dans 1003.1 depuis 2001.

© Dunod – La photocopie non autorisée est un délit.

Nom IEEE	Nom	Notes
1003.1j	Extensions temps réel avancées (*Advanced Realtime extensions*)	Dernière version 2000.
1003.1k	Interface de programmation des supports amovibles (*Removable Media API*)	Abandonné en 1997.
1003.1L	Non utilisé (pour éviter les confusions 1003.11 – 1003.1l ?)	
1003.1m	Point de contrôle/Redémarrage (*Checkpoint/Restart*)	Abandonné en 2000.
1003.1n	Corrections aux 1003.1, 1b, 1c, 1i (*Fixes to 1003.1/1b/.1i/.1c*)	Abandonné en 2001.
1003.1o	Non utilisé (pour éviter les confusions 1003.10 – 1003.1o ?)	
1003.1p	Limites de ressources (*Resource Limits*)	En développement depuis 1995.
1003.1q	Trace (*Tracing*)	Dernière version 2000. Intégré dans 1003.1 depuis 2001.
1003.1r	(*1003.1g Alignment w/ Single Unix Spec*)	Abandonné en 1997.
1003.1s	Synchronisation d'horloges (*Sync Clock*)	Choix d'abandonner en 2002.
1003.2&2a	Shell & outils, extension de portabilité (*Shell & Tools, User Portability Extensions*)	Dernière version 1992.
1003.2b	Utilitaires supplémentaires (*Additional Utilities*)	Intégrée dans la 1003.1 depuis 2001.
1003.2c	Sécurité (*Security*)	Intégrée dans 1003.1e.
1003.2d	Batch (*Batch*)	Dernière version 1994. Intégrée dans 1003.1 depuis 2001.
1003.2e	Utilitaires de supports amovibles (*Removable Media Utilities*)	Abandonné en 1997.
1003.3	Méthodes de test de conformité à POSIX (*Test Methods for Measuring Conformance to POSIX*)	Dernière version IEEE en 1991. Intégrée dans ISO/IEC 13210 :1999 en 1999.
1003.5	Interface Ada avec 1003.1 (*Ada Binding to 1003.1*)	Dernière version 1997.
1003.5a	Mise à jour Ada (*Ada Update*)	Abandonné en 1996.

Nom IEEE	Nom	Notes
1003.5b	Ada temps réel (*Ada Realtime*)	Dernière version 1996.
1003.5c	Interface Ada à 1003.1g (*Ada bindings to 1003.1g*)	Dernière version 1998.
1003.5d	(*Ada PII – Sockets*)	Abandonné en 1996.
1003.5f	Interface Ada à 1003.21 (*Ada binding to 1003.21*)	En développement depuis 1997.
1003.5g	Interfaces temps réel Ada (*Ada Binding to Real-Time Interfaces*)	En cours d'abandon en 2002.
1003.5h	Interface Ada aux horloges (*Ada binding to 1003.1s – Sync Clock*)	En cours d'abandon en 2002.
1003.9	Interface Fortran à 1003.1 (*Fortran binding to 1003.1*)	Dernière version 1996, abandonné en 2002.
1003.10	Environnement pour les applications sur super ordinateurs (*Supercomputing Application Environment Profile*)	Dernière version 1995, abandonné depuis 2001.
1003.11	(*Trans Proc Application Environment Profile*)	Abandonné en 1993.
1003.13	Profils temps réel (*Realtime Application Environment Profile*)	Dernière version 1998.
1003.13a	Profils temps réel embarqué (*Embedded Systems Application Profile Environment*)	Abandonné en 2002.
1003.13b	Profils temps réel supplémentaires (*Additional Real-Time Profiles*)	Abandonné en 2002.
1003.14	(*Multi Processing Application Environment Profile*)	Abandonné en 1998.
1003.16	Interfaces indépendantes d'un langage à 1003.1 (*LIS binding to 1003.1*)	Abandonné en 1993.
1003.17	Interface de programmation XDS (*XDS Directory API*)	En développement depuis 1993. Passé norme ISO depuis.
1003.18	Profil POSIX (*POSIX profile*)	Abandonné en 1998.
1003.19	Interface Fortran 90 à 1003.1 (*Fortran90 binding to 1003.1*)	Abandonné en 1993.

© Dunod – La photocopie non autorisée est un délit.

Nom IEEE	Nom	Notes
1003.21	Communication indépendante du langage de programmation dans les systèmes temps reel distribués (*Realtime Distrib Sys Comm (LIS)*)	En développement (version 4 en 2002).
1003.22	Guide de sécurité (*Security Framework Guide*)	En développement depuis 1995.
1003.23	Guide de développement de profils utilisateurs pour systèmes ouverts (*Guide for Developing User Organizations Open Systems Environment (OSE) Profiles*)	Dernière version 1998. Abandonné en 2004.
1003.24	Interface de programmation *X-Window* pour Ada (*Ada binding : X Window Modular Toolkit*)	Abandonné en 2001.
1003.25	Tolérance aux fautes (*Fault Tolerance*)	Nouveau nom de 1003.1h. En développement depuis 1999.
1003.26	Interface de programmation des contrôleurs de périphériques (*Device Control APIs*)	En développement depuis 2000.
2003.1	Méthodes de test pour mesurer la conformité à 1003.1 (*Test Methods for 1003.1*)	Dernière version 2000.
2003.1b	Méthodes de test pour mesurer la conformité à 1003.1b (*Test Methods for 1003.1b*)	Dernière version 2001.
2003.2	Méthodes de test pour mesurer la conformité à 1003.2 (*Test Methods for 1003.2*)	Dernière version en 1996, réaffirmé en 2002.

C • MODULE DE BOÎTES AUX LETTRES POSIX

Cette annexe présente le code source commenté d'un module de boîtes aux lettres implémenté en C, en utilisant la norme POSIX. Le module intègre les boîtes aux lettres de taille 1 présentées au paragraphe 6.2.1, p. 291, et y adjoint les boîtes aux lettres de taille *n* (avec et sans écrasement).

C.1 En-tête de module

```
/*
    BaLs.h : définitions de différents types de BaL
    Taille 1, sans ecrasement: bal
    Taille 1, avec écrasement: bal_ecr
    Taille n, sans ecrasement: bal_n
    Taille n, avec écrasement: bal_n_ecr
*/
#ifndef _BALS_H_
#define _BALS_H_
#include <pthread.h>
/*
   Boîte aux lettres de taille 1 sans écrasement (i.e. envoi et
réception bloquants)
*/
typedef struct s_bal {
      char * buf;/* Zone dans laquelle le message est stocké, mémoire
allouée DYNAMIQUEMENT par bal_init */
      unsigned char vide;/* booléen permettant de savoir si la boîte
est vide ou pleine */
      pthread_mutex_t mutex;/* Sémaphore d'exclusion mutuelle
garantissant qu'une seule tâche manipule la structure */
      pthread_cond_t pas_plein,pas_vide;/* Variables conditionnelles
utilisées pour signaler un ajout/retrait d'élément susceptible de
réveiller une tâche en attente */
      unsigned taille_element;/* taille d'un élément de boîte aux
lettres */
} *bal;

bal bal_init(const unsigned taille_element);
/* Crée et initialise une boîte aux lettres vide
pouvant contenir un élément de taille_element octets
Nécessite: taille_element>0
Renvoie: la BaL si succès, 0 sinon */
int bal_recevoir(bal b, char *buf);
/* Attend un message et reçoit un message de la BaL b
Primitive bloquante
```

© Dunod – La photocopie non autorisée est un délit.

```
Nécessite: b initialisée par bal_init
           taille(buf) >= taille_element donné lors de bal_init
Entraine: buf contient le message reçu
Renvoie: taille_element */
int bal_envoyer(bal b, const char *buf);
/* Envoie un message dans la BaL b (peut attendre que la BaL soit
vide)
Primitive bloquante
Nécessite: b initialisée par bal_init
           taille(buf) >= taille_element donné lors de bal_init
Entraine: la boîte aux lettres contient le message buf
Renvoie: taille_element */
void bal_delete(bal b);
/* Supprime la BaL b
Nécessite: b initialisée par bal_init
Entraine: b est supprimée */

/* Boîte aux lettres avec écrasement de taille 1 (i.e. réception
bloquante, envoi non bloquant) */
typedef struct s_bal_ecr {
      char * buf; /* Zone dans laquelle le message est stocké, mémoire
allouée DYNAMIQUEMENT par bal_ecr_init */
      unsigned char vide;/* booléen permettant de savoir si la boîte
est vide ou pleine */
      pthread_mutex_t mutex;/* Sémaphore d'exclusion mutuelle
garantissant qu'une seule tâche manipule la structure */
      pthread_cond_t pas_vide;/* Variable conditionnelle utilisée
pour signaler un ajout d'élément susceptible de réveiller une tâche en
attente */
      unsigned taille_element;/* taille d'un élément de boîte aux
lettres */
} *bal_ecr;

bal_ecr bal_ecr_init(const unsigned taille_element);
/* Crée et initialise une boîte aux lettres à écrasement vide
pouvant contenir un élément de taille_element octets
Nécessite: taille_element>0
Renvoie: la BaL si succès, 0 sinon */
int bal_ecr_recevoir(bal_ecr bal, char *buf);
/* Attend un message et reçoit un message de la BaL b
Primitive bloquante
Nécessite: b initialisée par bal_ecr_init
           taille(buf) >= taille_element donné lors de bal_ecr_init
Entraine: buf contient le message reçu
Renvoie: taille_element */
int bal_ecr_envoyer(bal_ecr bal, const char *buf);
/* Envoie un message dans la BaL b en écrasant si besoin le message
présent
Primitive non bloquante
Nécessite: b initialisée par bal_ecr_init
           taille(buf) >= taille_element donné lors de bal_ecr_init
Entraine: la boîte aux lettres contient le message buf
Renvoie: taille_element */
void bal_ecr_delete(bal_ecr bal);
/* Supprime la BaL b
Nécessite: b initialisée par bal_ecr_init
Entraine: b est supprimée */

/* Boîte aux lettres sans écrasement de taille n gérée en FIFO */
```

```
typedef struct s_bal_n {
      char ** buf;/* Zone dans laquelle les messages sont stockés,
mémoire allouée DYNAMIQUEMENT par bal_n_init
    Elle correspond à un tableau de taille cases, chaque case
contenant un élément de taille taille_element */
      unsigned debut, fin;/* utilisés pour gérer buf sous forme d'un
tableau circulaire */
      unsigned char vide;/* booléen permettant de discriminer les 2
cas où debut=fin (vide ou plein) */
      pthread_mutex_t mutex;/* Sémaphore d'exclusion mutuelle
garantissant qu'une seule tâche manipule la structure */
      pthread_cond_t pas_plein, pas_vide;/* Variables conditionnelles
utilisées pour signaler un ajout ou retrait d'élément susceptible de
réveiller une tâche en attente */
      unsigned taille, taille_element;/* taille de la boîte en
éléments, et taille de chaque élément */
} *bal_n;

bal_n bal_n_init(const unsigned taille, const unsigned
taille_element);
/* Crée et initialise une boîte aux lettres de taille n vide
pouvant contenir n élément de taille_element octets
Nécessite: taille_element>0 et taille>0
Renvoie: la BaL si succès, 0 sinon */
int bal_n_recevoir(bal_n bal, char *buf);
/* Attend un message et reçoit un message de la BaL bal
Primitive bloquante
Nécessite: bal initialisée par bal_n_init
           taille(buf) >= taille_element donné lors de bal_n_init
Entraine: le message reçu est le premier de la file de messages de bal
Renvoie: taille_element */
int bal_n_envoyer(bal_n bal, const char *buf);
/* Envoie un message dans la BaL bal (peut attendre que la BaL soit
vide)
Primitive bloquante
Nécessite: bal initialisée par bal_n_init
           taille(buf) >= taille_element donné lors de bal_n_init
Entraine: le message buf est ajouté à la fin de la file de messages de
bal
Renvoie: taille_element */
void bal_n_delete(bal_n bal);
/* Supprime la BaL bal
Nécessite: bal initialisée par bal_n_init
Entraine: bal est supprimée */

/* Boîte aux lettres avec écrasement de taille n */
typedef struct s_bal_n_ecr {
      char ** buf; /* Zone dans laquelle les messages sont stockés,
mémoire allouée DYNAMIQUEMENT par bal_n_ecr_init
    Elle correspond à un tableau de taille cases, chaque case
contenant un élément de taille taille_element */
      unsigned debut, fin;/* utilisés pour gérer buf sous forme d'un
tableau circulaire */
      unsigned char vide;/* booléen permettant de discriminer les 2
cas où debut=fin (vide ou plein) */
      pthread_mutex_t mutex;/* Sémaphore d'exclusion mutuelle
garantissant qu'une seule tâche manipule la structure */
```

© Dunod – La photocopie non autorisée est un délit.

```
      pthread_cond_t pas_vide;/* Variable conditionnelle utilisée
pour signaler un ajout d'élément susceptible de réveiller une tâche en
attente */
      unsigned taille, taille_element;/* taille de la boîte en
éléments, et taille de chaque élément */
} *bal_n_ecr;

bal_n_ecr bal_n_ecr_init(const unsigned taille, const unsigned
taille_element);
/* Crée et initialise une boîte aux lettres de taille n à écrasement
vide
pouvant contenir n élément de taille_element octets
Nécessite: taille_element>0 et taille>0
Renvoie: la BaL si succès, 0 sinon */
int bal_n_ecr_recevoir(bal_n_ecr bal, char *buf);
/* Attend un message et reçoit un message de la BaL bal
Primitive bloquante
Nécessite: bal initialisée par bal_n_ecr_init
           taille(buf) >= taille_element donné lors de bal_n_ecr_init
Entraine: le message reçu est le premier de la file de messages de bal
Renvoie: taille_element */
int bal_n_ecr_envoyer(bal_n_ecr bal, const char *buf);
/* Envoie un message dans la BaL bal (peut attendre que la BaL soit
vide)
Primitive non bloquante
Nécessite: bal initialisée par bal_n_ecr_init
           taille(buf) >= taille_element donné lors de bal_n_ecr_init
Entraine: le message buf est ajouté à la fin de la file de messages de
bal,
          si la file est pleine, le plus ancien message est supprimé
Renvoie: taille_element */
void bal_n_ecr_delete(bal_n_ecr bal);
/* Supprime la BaL bal
Nécessite: bal initialisée par bal_n_ecr_init
Entraine: bal est supprimée */
#endif
```

C.2 Corps de module

```
/*
    BaLs.c : définitions de différents types de BaL
    Taille 1, sans ecrasement: bal
    Taille 1, avec écrasement: bal_ecr
    Taille n, sans ecrasement: bal_n
    Taille n, avec écrasement: bal_n_ecr
*/
#include "BaLs.h"
bal bal_init(const unsigned taille_element) {
    bal b;
      if (!(b=(bal)malloc(sizeof(struct s_bal)))) return 0; /*
Allocation de la structure */
      if (!((*b).buf=(char *)malloc(taille_element))) return 0; /*
Allocation du buffer contenant un message */
      (*b).vide=1; /* Initialement la bal est vide */
      pthread_mutex_init(&((*b).mutex),0); /* Création du sémaphore
garantissant l'exclusion mutuelle des accès à la structure */
      (*b).taille_element=taille_element;
```

```
      pthread_cond_init (&((*b).pas_vide), 0); /* Variable
conditionnelle qui sera déclenchée lorsqu'un message est ajouté */
      pthread_cond_init (&((*b).pas_plein), 0); /* Variable
conditionnelle qui sera déclenchée lorsque la bal est vidée */
      return b;
}

int bal_recevoir(bal b, char *buf) {
    pthread_mutex_lock (&(*b).mutex);
    /* Exclusion mutuelle */
      while ((*b).vide) {
            pthread_cond_wait (&(*b).pas_vide, &(*b).mutex);
            /* On attend que la bal contienne un message */
      }
      memcpy(buf,(*b).buf,(*b).taille_element); /* Copie du message
dans buf */
      (*b).vide=1; /* La boîte est maintenant vide */
      pthread_mutex_unlock (&(*b).mutex); /* Fin de l'exclusion
mutuelle */
      pthread_cond_broadcast (&(*b).pas_plein); /* On réveille
d'éventuelles tâches en attente sur la boîte pleine */
      return (*b).taille_element;
}

int bal_envoyer(bal b, const char *buf) {
    pthread_mutex_lock (&(*b).mutex);
    /* Exclusion mutuelle */
      while (!(*b).vide) {
            pthread_cond_wait (&(*b).pas_plein, &(*b).mutex);
            /* On attend que la bal soit vide */
      }
      memcpy((*b).buf,buf,(*b).taille_element); /* On copie le
message dans la bal */
      (*b).vide=0; /* La boîte contient un message */
      pthread_mutex_unlock (&(*b).mutex); /* Fin de l'exclusion
mutuelle */
      pthread_cond_signal (&(*b).pas_vide); /* On réveille
l'éventuelle tâche en attente d'un message */
      return (*b).taille_element;
}

void bal_delete(bal b) {
    pthread_cond_destroy(&(*b).pas_vide);
    pthread_cond_destroy(&(*b).pas_plein);
    pthread_mutex_lock(&(*b).mutex);
    free((*b).buf);
    pthread_mutex_unlock(&(*b).mutex);
    pthread_mutex_destroy(&(*b).mutex);
    free(b);
}
bal_ecr bal_ecr_init(const unsigned taille_element) {
    bal_ecr bal;
      if (!(bal=(bal_ecr)malloc(sizeof(struct s_bal_ecr)))) return
0;/* Allocation de la structure */
      if (!((*bal).buf=(char *)malloc(taille_element))) return 0;/*
Allocation du buffer contenant un message */
      (*bal).vide=1;/* Initialement la bal est vide */
      pthread_mutex_init(&((*bal).mutex),0);/* Création du sémaphore
garantissant l'exclusion mutuelle des accès à la structure */
```

© Dunod – La photocopie non autorisée est un délit.

```
      (*bal).taille_element=taille_element;
      pthread_cond_init(&((*bal).pas_vide), 0);/* Variable
conditionnelle qui sera déclenchée lorsqu'un message est ajouté */
      return bal;
}

int bal_ecr_recevoir(bal_ecr bal, char *buf) {
    pthread_mutex_lock (&(*bal).mutex);
    /* Exclusion mutuelle */
      while ((*bal).vide) {
      /* Tant que la boîte aux lettres est vide */
            pthread_cond_wait (&(*bal).pas_vide, &(*bal).mutex);/* On
attend que la bal contienne un message */
      }
      memcpy(buf,(*bal).buf,(*bal).taille_element);/* Copie du
message dans buf */
      (*bal).vide=1;/* La boîte est maintenant vide */
      pthread_mutex_unlock (&(*bal).mutex);/* Fin de l'exclusion
mutuelle */
      return (*bal).taille_element;
}

int bal_ecr_envoyer(bal_ecr bal, const char *buf) {
    pthread_mutex_lock (&(*bal).mutex);
    /* Exclusion mutuelle */
      memcpy((*bal).buf,buf,(*bal).taille_element);/* Le message
écrase un éventuel message présent dans la bal */
      (*bal).vide=0;/* La boîte contient un message */
      pthread_mutex_unlock (&(*bal).mutex);/* Fin de l'exclusion
mutuelle */
      pthread_cond_signal (&(*bal).pas_vide);/* On réveille
l'éventuel thread en attente d'un message */
      return (*bal).taille_element;
}

void bal_ecr_delete(bal_ecr bal) {
    pthread_cond_destroy(&(*bal).pas_vide);
    pthread_mutex_lock(&(*bal).mutex);
    free((*bal).buf);
    pthread_mutex_unlock(&(*bal).mutex);
    pthread_mutex_destroy(&(*bal).mutex);
    free(bal);
}
bal_n bal_n_init(const unsigned taille, const unsigned taille_element)
{
    bal_n bal;
      if (!(bal=(bal_n)malloc(sizeof(struct s_bal_n)))) return 0;/*
Allocation de la structure */
      if (!((*bal).buf=(char **)calloc(taille,taille_element)))
return 0; /* Allocation du buffer contenant taille messages */
      (*bal).vide=1; /* La bal est initialement vide */
      pthread_mutex_init(&((*bal).mutex),0); /* Création du sémaphore
garantissant l'exclusion mutuelle des accès à la structure */
      (*bal).debut=0; /* indice de début de la file dans le tableau
circulaire */
      /* Invariant: non vide => debut est l'indice du plus vieux
message non lu
              vide => debut=fin*/
      (*bal).fin=0;
```

```
      /* Invariant: vide ou debut!=fin => fin est l'indice de la
première case disponible du tableau
              non vide et debut=fin => fin est l'indice du message le
plus ancien */
      (*bal).taille_element=taille_element;
      (*bal).taille=taille;
      pthread_cond_init (&((*bal).pas_plein), 0);
      /* Variable conditionnelle qui sera déclenchée lorsqu'un message
est retiré (i.e.
       quand il y a au moins une place disponible */
      pthread_cond_init (&((*bal).pas_vide), 0);
      /* Variable conditionnelle qui sera déclenchée lorsqu'un message
est ajouté (i.e.
       quand il y a au moins un message à lire */
      return bal;
}

int bal_n_recevoir(bal_n bal, char *buf) {
    pthread_mutex_lock (&(*bal).mutex);
    /* Exclusion mutuelle */
      while ((*bal).vide) {
            pthread_cond_wait (&(*bal).pas_vide, &(*bal).mutex);
            /* On attend que la bal contienne un message */
      }
      /* Il existe au moins un message, le plus ancien a l'indice
debut */
      memcpy(buf,&(*bal).buf[(*bal).debut],(*bal).taille_element); /*
Copie du message le plus ancien dans buf */
      (*bal).debut=((*bal).debut+1)%(*bal).taille;
      if ((*bal).debut==(*bal).fin) (*bal).vide=1; /* Respect de
l'invariant */
      pthread_mutex_unlock (&(*bal).mutex); /* Fin de l'exclusion
mutuelle */
      pthread_cond_broadcast (&(*bal).pas_plein); /* Réveil d'un
éventuel thread en attente de place dans la bal */
      return (*bal).taille_element;
}

int bal_n_envoyer(bal_n bal, const char *buf) {
    pthread_mutex_lock (&(*bal).mutex);
    /* Exclusion mutuelle */
while ((!(*bal).vide) && ((*bal).debut==(*bal).fin)) {
        /* La bal est pleine */
            pthread_cond_wait (&(*bal).pas_plein, &(*bal).mutex);
            /* On attend qu'il y ait de la place dans la bal */
      }
      memcpy(&(*bal).buf[(*bal).fin],buf,(*bal).taille_element); /*
Copie du message à la fin de la file (indice fin d'après l'invariant)
*/
      (*bal).fin=((*bal).fin+1)%(*bal).taille;
      (*bal).vide=0; /* Respect de l'invariant */
      pthread_mutex_unlock (&(*bal).mutex);
      /* Fin de l'exclusion mutuelle */
      pthread_cond_signal (&(*bal).pas_vide); /* Réveil de l'éventuel
thread en attente de message */
      return (*bal).taille_element;
}

void bal_n_delete(bal_n bal) {
```

© Dunod – La photocopie non autorisée est un délit.

```
    pthread_cond_destroy(&(*bal).pas_plein);
    pthread_cond_destroy(&(*bal).pas_vide);
    pthread_mutex_lock(&(*bal).mutex);
    free((*bal).buf);
    pthread_mutex_unlock(&(*bal).mutex);
    pthread_mutex_destroy(&(*bal).mutex);
    free(bal);
}
bal_n_ecr bal_n_ecr_init(const unsigned taille, const unsigned
taille_element) {
    bal_n_ecr bal;
      if (!(bal=(bal_n_ecr)malloc(sizeof(struct s_bal_n_ecr))))
return 0; /* Allocation de la structure */
      if (!((*bal).buf=(char **)calloc(taille,taille_element)))
return 0;/* Allocation du buffer contenant taille messages */
      (*bal).vide=1; /* La bal est initialement vide */
      pthread_mutex_init(&((*bal).mutex),0); /* Création du sémaphore
garantissant l'exclusion mutuelle des accès à la structure */
(*bal).debut=0;
      /* Invariant: non vide => debut est l'indice du plus vieux
message non lu
              vide => debut=fin*/
      (*bal).fin=0;
      /* Invariant: vide ou debut!=fin => fin est l'indice de la
première case disponible du tableau
              non vide et debut=fin => fin est l'indice du message le
plus ancien */
      (*bal).taille_element=taille_element;
      (*bal).taille=taille;
      pthread_cond_init (&((*bal).pas_vide), 0);
      /* Variable conditionnelle qui sera déclenchée lorsqu'un message
est ajouté (i.e.
         quand il y a au moins un message à lire */
      return bal;
}

int bal_n_ecr_recevoir(bal_n_ecr bal, char *buf) {
    pthread_mutex_lock (&(*bal).mutex);
    /* Exclusion mutuelle */
      while ((*bal).vide) {
            pthread_cond_wait (&(*bal).pas_vide, &(*bal).mutex);
            /* On attend que la bal contienne un message */
      }
      /* Il existe au moins un message, le plus ancien a l'indice
debut */
      memcpy(buf,&(*bal).buf[(*bal).debut],(*bal).taille_element); /*
Copie du message le plus ancien dans buf */
      (*bal).debut=((*bal).debut+1)%(*bal).taille;
      if ((*bal).debut==(*bal).fin) (*bal).vide=1; /* Respect de
l'invariant */
      pthread_mutex_unlock (&(*bal).mutex);
      /* Fin de l'exclusion mutuelle */
      return (*bal).taille_element;
}

int bal_n_ecr_envoyer(bal_n_ecr bal, const char *buf) {
    pthread_mutex_lock (&(*bal).mutex);
    /* Exclusion mutuelle */
      memcpy(&(*bal).buf[(*bal).fin],buf,(*bal).taille_element);
```

```
      /* Copie du message à la fin de la file (indice fin d'après
l'invariant) avec
       écrasement éventuel du plus ancien message */
      if ((*bal).fin==(*bal).debut && !(*bal).vide) {
      /* La boîte était déjà pleine, il y a eu écrasement => debut est
repoussé sur le plus ancien message restant */
        (*bal).debut=((*bal).debut+1)%(*bal).taille;
      }
      (*bal).fin=((*bal).fin+1)%(*bal).taille;
      (*bal).vide=0; /* Respect de l'invariant */
      pthread_mutex_unlock (&(*bal).mutex);
      /* Fin de l'exclusion mutuelle */
      pthread_cond_signal (&(*bal).pas_vide);/* Réveil de l'éventuel
thread en attente de message */
      return (*bal).taille_element;
}

void bal_n_ecr_delete(bal_n_ecr bal) {
    pthread_cond_destroy(&(*bal).pas_vide);
    pthread_mutex_lock(&(*bal).mutex);
    free((*bal).buf);
    pthread_mutex_unlock(&(*bal).mutex);
    pthread_mutex_destroy(&(*bal).mutex);
    free(bal);
}
```

D • MODULE DE COMMUNICATION ADA

Cette annexe présente le code source commenté d'un module Ada générique définissant les modules de données, synchronisations et boîtes aux lettres.

D.1 Spécification de module

D.1.1 Gestion de files bornées

Les boîtes aux lettres définies ci-après nécessitent l'emploi de files de messages bornées. Le module présenté dans cette section définit des files bornées implémentées sous la forme de tableaux circulaires.

■ Spécification de module

```
-- FileBornee est une file bornée:
-- lorsqu'on enfile un élément dans une file pleine, l'élément le plus
-- ancien est écrasé
-- Notations: si F est une file bornée:
-- tête est la tête de F
-- [F] est le contenu (ordonné) de F
-- [F\tête] est le contenu de F sans la tête
-- e[F] est la file F à laquelle on ajoute l'élément e en queue
-- 'F représente F avant une primitive
-- F' représente F après une primitive
generic
type element is private;
package File_Bornee is
   type File_Bornee(taille: positive) is private;
   function Vide(F: File_Bornee) return boolean;
   -- Retourne: vrai ssi cardinal([F])=0
   function Pleine(F: File_Bornee) return boolean;
   -- Retourne: vrai ssi cardinal([F])=taille
   procedure Enfiler(F: in out File_Bornee; e: element);
   -- Ajoute un élément en queue de la file
   -- (si la file est pleine, le plus ancien élément est supprimé)
   -- Entraine: non Pleine(F) => F'= e['F]
   --           Pleine(F)     => F'= e['F\tête]
   procedure Defiler(F: in out File_Bornee; e: out element);
   -- Défile un élément de F
   -- Nécessite: non Vide(F)
   -- Exception: Vide(F) => Constraint_Error
   -- Entraine: e = tête et F'=['F\tête]
```

© Dunod – La photocopie non autorisée est un délit.

```
private
   -- Implémentation par tableau géré sous forme de liste circulaire
   type Tableau_Elements is array (positive range <>) of element;
   -- Tableau non contraint d'éléments
   type File_Bornee(taille: positive) is record
      T: Tableau_Elements(1..taille);
      Debut, Fin: natural :=1;
      Vide : boolean :=true;
   end record;
end;
```

Corps de module

```
-- File_Bornee est une file bornée:
-- lorsqu'on enfile un élément dans une file pleine, l'élément le plus
-- ancien est écrasé
-- Notations: si F est une file bornée:
-- tête est la tête de F
-- [F] est le contenu (ordonné) de F
-- [F\tête] est le contenu de F sans la tête
-- e[F] est la file F à laquelle on ajoute l'élément e en queue
-- 'F représente F avant une primitive
-- F' représente F après une primitive
package body File_Bornee is

   procedure Defiler (F: in out File_Bornee; e: out element) is
   -- Défile un élément de F
   -- Nécessite: non Vide(F)
   -- Exception: Vide(F) => Constraint_Error
   -- Entraine: e = tête et F'=['F\tête]
   begin
      if F.Vide then raise Constraint_Error; end if;
      E := F.T(F.Debut);
      F.debut:= (F.debut mod F.T'length)+1;
      if F.debut = F.fin then
         F.Vide := true;
      end if;
   end Defiler;

   procedure Enfiler (F: in out File_Bornee; e: element) is
   -- Ajoute un élément en queue de la file
   -- (si la file est pleine, le plus ancien élément est supprimé)
   -- Entraine: non Pleine(F) => F'= e['F]
   --           Pleine(F)     => F'= e['F\tête]
   begin
      F.T(F.Fin) := E;
      if not F.Vide and F.Fin=F.Debut then
         -- Il y a eu écrasement
         F.Debut:=(F.Debut mod F.T'Length)+1;
      end if;
      F.Fin:=(F.Fin mod F.T'Length)+1;
      F.Vide := false;
   end Enfiler;

   function Pleine(F: File_Bornee) return boolean is
   -- Retourne: vrai ssi cardinal([F])=taille
   begin
      return F.debut = F.fin and not F.Vide;
   end;
```

```
   function Vide (F: File_Bornee) return boolean is
   -- Retourne: vrai ssi F est vide
   begin
      return F.Vide;
   end Vide;

end File_Bornee;
```

D.1.2 Spécification du module de communication

```
-- Paquetage Communications
-- Définit des outils génériques de communication DARTS
-- Modules de données, synchronisation, boîtes aux lettres
-- avec ou sans écrasement, de taille unitaire ou bornée

with System; use System; -- Pour les priorités
with File_Bornee; -- Définit une file bornée d'objets gérée sous forme
de tableau circulaire
generic
   type element is private;
package Communications is
   package File_Bornee_N_Elements is new
File_Bornee(Element=>Element);
   -- Instanciation d'une file bornée sur le type element
   use File_Bornee_N_Elements;

   type Pt_Element is access all element; -- Pointeur sur element
------------------------
-- Modules de données
------------------------
   protected type MDD(priorité: natural := priority'last) is
   -- module de données sans valeur initiale
   -- Note: la valeur par défaut sur la priorité (utilisée si le
   -- paramètre est omis)
   -- est le paramètre par défaut choisi par Ada lors de la création
   -- d'un module de données
      pragma Priority(priorité);
      procedure Ecrire(e: element);
      -- modifie le contenu
      function Lire return element;
      -- retourne le contenu
   private
      val: element;
   end;

  protected type Mddi(Initial: Pt_Element:=null; Priorité: Natural :=
Priority'Last) is
   -- module de données avec valeur initiale
   -- le passage par pointeur est dû au fait qu'un discriminant
   -- doit être de type discret ou pointeur
   -- Note: la valeur par défaut sur la priorité (utilisée si le
   -- paramètre est omis)
   -- est le paramètre par défaut choisi par Ada lors de la création
   -- d'un module de données
      pragma Priority(priorité);
      procedure Ecrire(e: element);
      -- modifie le contenu
      function Lire return element;
      -- retourne le contenu
```

© Dunod – La photocopie non autorisée est un délit.

```
   private
      val: element := initial.all;
   end;
------------------------
-- Synchronisation
------------------------
   protected type Synchro_C(priorité: natural := priority'last) is
   -- Synchronisation à compte (les déclenchements successifs
   -- non pris en compte sont accumulés)
      pragma Priority(priorité);
      procedure Signal;
      entry Wait; -- when Nombre_Signalés > 0
   private
      Nombre_Signalés: natural := 0;
   end;

   protected type Synchro_B(priorité: natural := priority'last) is
   -- Synchronisation binaire (les déclenchements successifs
   -- non pris en compte ne sont pas accumulés)
      pragma Priority(priorité);
      procedure Signal;
      entry Wait; -- when Signalé
   private
      Signalé: boolean := false;
   end;
------------------------
-- Boîtes aux lettres
------------------------
  protected type BaL_1_Ecrasement(priorité: natural := priority'last)
is
   -- Boîte aux lettres de taille 1 avec écrasement
      pragma Priority(priorité);
      procedure Envoyer(e: element);
      entry Recevoir(e: out element); -- when Pleine
   private
      Pleine: boolean := false;
      val: element;
   end;

   protected type BaL_1(priorité: natural := priority'last) is
   -- Boîte aux lettres de taille 1 sans écrasement
      pragma Priority(priorité);
      entry Envoyer(e: element); -- when not Pleine
      -- l'envoi peut être bloquant
      entry Recevoir(e: out element); -- when Pleine
   private
      Pleine: boolean := false;
      val: element;
   end;
  protected type BaL_n_Ecrasement(n:positive:=1; priorité: natural :=
priority'last) is
   -- Boîte aux lettres de taille n avec écrasement
      pragma Priority(priorité);
      procedure Envoyer(e: element);
      entry Recevoir(e: out element); -- when not Vide(F)
   private
      F: File_Bornee_N_Elements.File_Bornee(n); -- File bornée de
taille n
   end;
```

```
   protected type BaL_n(n:positive:=1; priorité: natural :=
priority'last) is
   -- Boîte aux lettres de taille n sans écrasement
      pragma Priority(priorité);
      entry Envoyer(e: element); -- when not Pleine(F)
      -- l'envoi peut être bloquant
      entry Recevoir(e: out element); -- when not Vide(F)
   private
      F: File_Bornee_N_Elements.File_Bornee(n); -- File bornée de
taille n
   end;
end;
```

D.2 Corps de module

```
-- Paquetage Communications
-- Définit des outils génériques de communication DARTS
-- Modules de données, synchronisation, boîtes aux lettres
-- avec ou sans écrasement, de taille unitaire ou bornée
package body Communications is
   protected body BaL_1 is
      entry Envoyer (e: element) when not Pleine is
      begin
         val := e;
         Pleine := true;
      end Envoyer;
      entry Recevoir (e: out element) when Pleine is
      begin
         e := val;
         Pleine := false;
      end Recevoir;
   end BaL_1;

   protected body BaL_1_Ecrasement is
      procedure Envoyer (e: element) is
      begin
         val := e;
         Pleine := true;
      end Envoyer;
      entry Recevoir (e: out element) when Pleine is
      begin
         e := val;
         Pleine := false;
      end Recevoir;
   end BaL_1_Ecrasement;

   protected body BaL_n is
      entry Envoyer (e: element) when not Pleine(F) is
      begin
         Enfiler(F,e);
      end Envoyer;
      entry Recevoir (e: out element) when not Vide(F) is
      begin
         Defiler(F,e);
      end Recevoir;
   end BaL_n;
```

© Dunod – La photocopie non autorisée est un délit.

```
   protected body BaL_n_Ecrasement is
      procedure Envoyer (e: element) is
      begin
         Enfiler(F,e);
      end Envoyer;
      entry Recevoir (e: out element) when not Vide(F) is
      begin
         Defiler(F,e);
      end Recevoir;
   end BaL_n_Ecrasement;

   protected body MDD is
      procedure Ecrire (e: element) is
      begin
         val := e;
      end Ecrire;
      function Lire return element is
      begin
         return Val;
      end Lire;
   end MDD;

   protected body MDDi is
      procedure Ecrire (e: element) is
      begin
         val := e;
      end Ecrire;
      function Lire return element is
      begin
         return Val;
      end Lire;
   end MDDi;

   protected body Synchro_B is
      procedure Signal is
      begin
         Signalé := true;
      end Signal;
      entry Wait when Signalé is
      begin
         Signalé := false;
      end Wait;
   end Synchro_B;

   protected body Synchro_C is
      procedure Signal is
      begin
         Nombre_Signalés := Nombre_Signalés+1;
      end Signal;
      entry Wait when Nombre_Signalés>0 is
      begin
         Nombre_Signalés := Nombre_Signalés-1;
      end Wait;
   end Synchro_C;

end Communications;
```

BIBLIOGRAPHIE

G. ASCH, et coll., *Les capteurs en instrumentation industrielle*, Dunod (1991).

J.G.P. BARNES, *Programmer en Ada 95*, Addison-Wesley (2000).

A. BURNS, A. WELLINGS, *Concurrency in Ada*, Cambridge University Press (1998).

G. BOOCH, *Ingénierie du logiciel ADA*, InterEditions (1988).

L. BRIAND, *Systèmes temps réel en ADA,* Masson (1991).

F. COTTET, *LabVIEW : Programmation et applications*, Dunod (2001).

F. COTTET, *Traitement des signaux et Acquisition de données*, Dunod (2002).

F. COTTET, J. DELACROIX, C. KAISER, Z. MAMMERI, *Ordonnancement temps réel*, Hermes (2000).

A. DORSEUIL, P. PILLOT, *Le temps réel en milieu industriel*, Dunod (1991).

P. FICHEUX, *Linux embarqué*, Eyrolles (2002).

S. GOLDSMITH, *Real-time Systems Developments,* Prentice Hall (1993).

H. GOMAA, *Software Design Methods for Concurrents and Real-Time Systems*, Addison-Wesley (1993).

D. HATLEY, I. PIRBHAI, *Stategies for Real-Time System Specification*, Hermes (1988).

B.W. KERNIGHAN, D.M. RITCHIE, *Le langage C, norme ANSI*, Dunod (1997).

M.H. KLEIN, T. RALYA, B. POLLAK, R. OBENZA, M.G. HARBOUR, *A Practitioner's Handbook for Real-Time Analysis*, Kluwer Academic Publishers (1993).

H. KOPETZ, *Real-Time Systems*, Kluwer Academic Publishers (1997).

A.M. LISTER, *Principes fondamentaux des systèmes d'exploitation*, Eyrolles (1983).

M. MALAGARDIS, *Projet Sceptre : proposition de standard de noyau d'exécutif temps réel*, Bureau d'Orientation de la Normalisation en Informatique (Rapp. BNI 26/2).

N. NISSANKE, *Realtime Systems*, Prentice Hall, series in Computer Science (1997).

H. NUSSBAUMER, *Informatique industrielle*, Presses Polytechniques Romandes (1986).

P. RICHARD, M. RICHARD, F. COTTET, « *Analyse holistique des systèmes temps réel distribués : principes et algorithmes* », chapitre 7 de « Ordonnancement pour l'informatique parallèle », Hermes (2003).

A. TANENBAUM, *Architecture de l'ordinateur, cours et exercices*, Dunod (2001).

© Dunod – La photocopie non autorisée est un délit.

A. Tanenbaum, *Réseaux, cours et exercices*, Dunod (1997).
J. Toux, *Capteurs*, Techniques de l'Ingénieur, traité Mesure et Contrôle.
D. Tschirhart, *Commande en Temps réel*, Dunod, Informatique Industrielle (1990).

Sites internet

Le protocole CAN (en anglais) : http://www.can-cia.de/can/
La norme POSIX (en anglais) : http://www.opengroup.org/
La norme OSEK/VDX (en anglais) : http://www.osek-vdx.org/
Le profil *Ravenscar* (en anglais) : http://polaris.dit.upm.es/~str/proyectos/ork/

LEXIQUE ANGLAIS – FRANÇAIS

Anglais	Français
Absolute deadline	Échéance
Access Point	Point d'Accès
Actuator	Actionneur
Arrival time (release time or ready time)	Date de réveil
Availability	Disponibilité (d'un système)
Best effort strategy	Stratégie de meilleur effort
Binding	Interface entre deux langages
Board Support Package	Module spécifique à une architecture matérielle
Broadcast communication	Communication par diffusion
Broadcast communication	Communication par diffusion
Buffer	Zone de mémoire tampon
Busy period	Période d'activité
Byte	Octet
Coding (Implementation)	Codage (implémentation)
Context switching	Changement de contexte
Critical (or exclusive access) resources	Ressources critiques
Critical resource	Ressource critique
Deadline	Échéance ou délai critique
Deadline Monotonic Scheduling	Ordonnancement basé sur délai critique
Deadlock	Interblocage
Dedicated real-time system	Système temps réel dédié

© Dunod – La photocopie non autorisée est un délit.

Anglais	Français
Deferrable server	Serveur ajournable
Deferred Service Routine	Routine de traitement d'interruption différée
Dependability	Sûreté de fonctionnement
Design	Conception
Dispatcher	Séquenceur
Distributed applications	Applications distribuées
Distributed real-time system	Système temps réel distribué
Earliest Deadline Scheduling	Ordonnancement basé sur l'échéance la plus proche
Embedded real-time system	Système temps réel embarqué
Failure	Défaillance
Feasible schedule	Séquence valide
Firmware	Logiciel enregistré en EPROM
First In First Out	Premier arrivé premier servi
Flags	Registre d'état
Full-Duplex	Dans les deux sens
Garbage collector	Ramasse miettes
Hard real-time system	Système temps réel à contraintes temporelles strictes
Heap	Segment de données, tas
Hook	Crochet, fonction pouvant être attachée à un événement particulier du système
Hub	Répéteur
Independent or dependant tasks	Tâches indépendantes ou dépendantes
Interoperability	Interopérabilité
Interrupt latency	Temps de masquage des interruptions
Interrupt Service Routine	Routine de traitement d'interruption
Interrupt Request	Requête d'interruption
Local area network	Réseau local
Log	Trace d'exécution
Mailbox	Boîte aux lettres
Message queue	Boîte aux lettres (file de messages)

Anglais	Français
Minimum Laxity Scheduling	Ordonnancement basé sur la laxité
Multi-Level Feedback	Ordonnancement à tourniquets par priorité dynamique
Multi-user resource	Ressource partageable
Mutex	Sémaphore d'exclusion mutuelle
Mutual exclusion	Exclusion mutuelle
Offset	Décalage
On-line or off-line scheduling	Ordonnancement en ligne ou hors ligne
One shot	Horloge déclenchée une fois
Open source	Logiciel libre
Operating system	Système d'exploitation
Overhead	Surcoût processeur dû au système d'exploitation
Overload	Surcharge
Patch	Modification
Pattern matching	Recherche de motif
Periodic ou aperiodic task	Tâche périodique ou apériodique
Pipe	Tube
Polled Loop	Boucle de scrutation
Polling	Scrutation, attente active
Polling server	Serveur à scrutation
Polling task	Tâche de scrutation
Precedence constraints	Contraintes de précédence
Preemptive latency	Temps de retard de l'ordonnanceur
Priority inheritance	Héritage de priorité
Probing	Test
Process Control	Contrôle de procédé
Processor laxity	Laxité du processeur
Processor load factor	Facteur de charge du processeur
Processor slack time	Temps creux du processeur
Processor utilization factor	Facteur d'utilisation du processeur

© Dunod – La photocopie non autorisée est un délit.

Anglais	Français
Pthread	Tâche POSIX
Quality of service (QoS)	Qualité de service
Rate Monotonic Scheduling	Ordonnancement basé sur la période
Real-time	Temps réel
Real-time kernel	Noyau temps réel
Real-time languages	Langages temps réel
Real-time network	Réseau temps réel
Real-time system	Système temps réel
Reliability	Fiabilité
Requirements	Besoins (spécifications)
Resource constraints	Contraintes de ressources
Resources	Ressources
Response time	Temps de réponse
Round Robin	A tourniquet
Safety	Sécurité (innocuité)
Sensor	Capteur
Shell	Invite de commande
Scheduling	Ordonnancement
Scheduling algorithm	Algorithme d'ordonnancement
Scheduling period (or schedule length)	Période d'étude
Security	Sécurité (confidentialité)
Sensor	Capteur
Shared memory	Mémoire partagée
Shortest Remaining Time First	Ordonnancement basé sur le temps de calcul restant
Socket	Connexion réseau
Soft real-time system	Système temps réel à contraintes temporelles relatives
Software life cycle	Cycle de vie d'un logiciel
Specification languages	Langages de spécification
Spinlock	Verrou

Anglais	Français
Stack	Pile
Structured Analysis Real Time	Analyse structurée temps réel
Timeout	Expiration d'un délai
Swap	Échange
Switch	Commutateur
Task	Tâche
Task criticality (task importance)	Importance de la tâche
Task period	Période de la tâche
Task priority	Priorité de la tâche
Task set	Ensemble de tâches
Thread	Tâche gérée directement par le système d'exploitation ou l'exécutif (processus léger)
Time sharing	Travail en temps partagé
Timer	Horloge
Timing failure	Faute temporelle
Token	Jeton
Trigger	Réveil
Watchdog	Chien de garde
War-Driving	Piratage de réseaux sans fil
Worst-case computation time	Durée maximale d'exécution

SIGLES

Sigle	Désignation complète
ADARTS	Ada Based Design Approach for Real-Time Systems (GOMAA, 1987)
AFNOR	Association Française de Normalisation
AGP	Advanced Graphic Port
ANSI	American National Standards Institute
ARP	Address Resolution Protocol
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
ASR	Asynchronous Service Routine
ATA	Advanced Technology Attachment
BaL	Boîte aux Lettres
BCC	Basic Conformance Class
BCP	Bloc de Contrôle de Processus
B.N.F.	Backus-Naur Form
BSP	Board Support Package
CAN	Controller Area Network (1994) – Convertisseur Analogique Numérique
CCC	Com Conformance Class
CCITT	Comité Consultatif International Téléphonique et Télégraphique
CIN	Code Interface Node
CNA	Convertisseur Numérique Analogique
CODARTS	Concurrent Design Approach for Real-Time Systems (GOMAA, 1987)
CODOP	Code Opération

Sigle	Désignation complète
CRC	Cyclic Redundancy Check
CSMA/CA	Carrier Sense Multiple Access/Collision Avoidance (1990)
CSMA/CD	Carrier Sense Multiple Access/Collision Detection – Ethernet (1985)
CSMA/DCR	Carrier Sense Multiple Access/Deterministic Collision Resolution (1990)
DARTS	Design Approach for Real-Time Systems (GOMAA, 1984)
DFD	Diagramme Flot de Données
DHCP	Dynamic Host Configuration Protocol
DM	Deadline Monotonic
DSR	Deferred Service Routine
E/S	Entrées/Sorties
ECC	Extended Conformance Class
EDF	Earliset Deadline First
EDL	Earliset Deadline Last
FAT	File Allocation Table
FDDI	Fibber Distributed Data Interface (1990)
FIFO	First In First Out
FIP	Factory Instrumentation Protocol (1990)
FPGA	Field Programmable Gate Array
FPLA	Field Programmable Logic Array
FTP	File Transfer Protocol
GCC	Gnu Compiler Collection
GNAT	GNU Ada Translator
GNU	GNU's Not Unix
GPL	General Public License
GRAFCET	Graphe Fonctionnel de Commande Étape Transition (IEC 1988)
HTR	Horloge Temps Réel
HOOD	Hiearchical Object Oriented Design (CRI-Cisi Ingénierie-Matra, 1987)
HTTP	Hyper Text Transfer Protocol
I-PDU	Interaction layer Processor Data Unit
IBSS	Independent Basic Service Set

© Dunod – La photocopie non autorisée est un délit.

Sigle	Désignation complète
ICANN	Internet Corporation for Assigned Names and Numbers
IDE	Integrated Drive Electronics
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IMAP	Internat Message Access Protocol
IMFS	In Memory File System
IP	Internet Protocol
IPC	InterProcess Communication
IPTES	Incremental Prototyping Technology for Embedded Real-Time Systems
IRQ	Interrupt ReQuest
ISA	Industry Standard Architecture
ISO	International Organization for Standardization
ISR	Interrupt Service Routine
ITU	International Communication Union
JSD	Jackson System Design (Michaël Jackson, 1981)
LL	Least Laxity
LLC	Logical Link Protocol
MAC	Medium Access Control
MDD	Module De Données
ML	Minimum Laxity
MLF	Multi-Level Feedback
MMU	Memory Management Unit
MSMC	Modélisation Simulation des machines Cybernétiques (Brenier, 2001)
MTU	Maximal Transfer Unit
MUX	Multiplexeur
NFS	Network File System
NTFS	New Technology File System
OIL	OSEK Implementation Language
OMG	Object Management Group
OMT	Object Modeling Technique

Sigle	Désignation complète
OS	Operating System
OSEK	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
OSI	Open Systems Interconnection
PC	Personal Computer
PCF	Point Coordination Function
PCI	Peripheral Componant Interconnect
PCMCIA	Personnal Computer Memory Card International Association
PDU	Protocol Data Unit
POP	Post Office Protocol
POSIX	Portable Operating System Interface
PSE	Profil d'environnement générique POSIX
PXI	PCI Extensions for Instrumentation
RAM	Random Access Memory
RIPE	Réseaux IP Européens
RM	Rate Monotonic
RMA	Rate Monotonic Analysis
RR	Round Robin
RT FIFO	File temps réel de messages
RTEMS	Real-Time Executive for Multiprocessor Systems
SA	Structured Analysis
SA_DT	Structured Analysis Design Technics
SA_RT	Structured Analysis Real Time (Ward-Mellor, 1984 ; Pirbhaï-Hatley, 1986)
SCSI	Small Computer System Interface
SD	Structured Design (E. Yourdon, L.L. Constantine, G. Meters, 1979)
SDL	Specification and Description Language (CCITT, 1988)
SE	Système d'Exploitation
SMTP	Simple Mail Transfer Protocol
SRPT	Shortest Remaining Time First
TDMA	Time Division Multiple Access (1990)
TCP	Transmission Control Protocol

© Dunod – La photocopie non autorisée est un délit.

Sigle	Désignation complète
TFTP	Trivial File Transfer Protocol
TOR	Tout Ou Rien
UAL	Unité Arithmétique et Logique
UDP	User Datagram Protocol
UML	Unified Modeling Language (OMG, 1995)
USB	Universal Serial Bus
VAN	Vehicle Area Network
VDX	Vehicle Distributed eXecutive
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VI	Virtual Instrument
VME	Versatile Module Eurocard
VXI	VME Extensions for Instrumentation
Wi-Fi	Wireless Fidelity
XSI	X/Open System Interface

INDEX

Généralités

.dll 394

A

Access Point 169
accumulateur 111
accusé de réception 173
actionneur 2
Ada body 273
Ada specification 273
Ad-Hoc 169
adresses IP privées 173
AFNOR 164
algèbre booléenne 115
algorithme
  à priorités 141
  d'ordonnancement 431
  du tourniquet (*round robin*) 141
  hors-ligne 407
allocation 492
analyse structurée SA 28
anneau à jeton 166, 169
anomalie
  de comportement 465, 474
  de fonctionnement 494
  environnement distribué 505
arbitrage de bus 160
architecture multitâche 8
ARINC 629 166
ARP 171
ASCII 125
  étendu 125
ASR (*Asynchronous Service Routine*) 192
assembleur 112
asservissement 391
associativité 118
atomique 146
attente active 126
attributs Ada 250

B

BCC1 229
BCC2 229
big endian 388
binaire 120
binary digit 110
binding 400
  Ada 284
bit 110
Bloc de Contrôle de Processus (BCP) 138
blocage fatal 472
bloqué 138
boîte aux lettres 88, 194
  nommée 212
boîtier de conditionnement 131
BSP (*Board Support Package*) 184
bus
  à jeton 169
  AGP (*Advanced Graphic Port*) 129
  ATA (*Advanced Technology Attachment*) 129
  d'entrées/sorties 128
  externe 128
  FireWire 129
  IDE (*Integrated Drive Electronics*) 129
  ISA (*Industry Standard Architecture*) 129

© Dunod – La photocopie non autorisée est un délit.

mémoire 111
parallèle 128
PCI (*Peripheral Component Interconnect*) 129
PCMCIA (*Personal Computer Memory Card International Association*) 129
SCSI (*Small Computer System Interface*) 129
série 128
USB 128

C

cadre 157
CALL 157
CAN (*Controller Area Network*) 166, 177
capteur 2
carte d'acquisition 131, 388
CCC0 230
CCC1 230
CCCA 230
CCCB 230
CCITT 164
centrale
d'acquisition 388
de conditionnement 388
chaînes de caractères 256, 258
chien de garde 208, 472
cible 23
classes
d'adresses IP 170
de conformité 229
client 173
client/serveur 173
code
intermédiaire 269
machine 112
CodeWarrior 239
CODOP (code opération) 113
coercion 368
cohérence
du contrôle 99
spatiale 180
temporelle 180
cohésion 99
fonctionnelle 100
séquentielle 99
temporelle 99
commande 251
d'un moteur à combustion 65
communication 9, 87, 463
asynchrone 194
par diffusion 160
par messages 193
point à point 160
commutateur 167
commutativité 118
compilation 270
complément 116
à 1 513
à 2 513
compteur 134
ordinal 111
conception préliminaire 19
conditionnement de signaux 136
conjonction 116
connecteurs 116, 268
connexion 175
contexte 139
contraintes
de bout en bout 409
de régularité 410
de temps 409
temporelles 3
Convertisseur Analogique Numérique (CAN) 135
Convertisseur Numérique Analogique (CNA) 136
corps de module 270
CRC 179
crochet (*hook*) 229
CSMA/CA 169
CSMA/CD 166-167
CSMA/DCR 166, 168
CSMA/MA 166
cycle 110
de développement en W 183, 351

D

datagramme 171, 174
date
d'activation 6
de réveil 6, 411
Deadline Monotonic 436
deadlock 149

défaut de page 158
délai
  critique 412
    dynamique 418
  de latence 416
delta 515
déréférencement 256
dérive des horloges 209, 297
désarmer une interruption 128
développement
  croisé 183
  incrémental 231
DHCP 173
diagramme 247
  de contexte 36
  de décomposition 39
  état-transition 45
  préliminaire 38
dictionnaire de données 31
direction des lignes 132
dispatcher 139
distribué 4
distributivité 118
division euclidienne 120
DQDB 166
driver 279
DSR (*Deffered Service Routine*) 207
durée d'exécution 6, 410

E

Earliest Deadline First 440
ECC1 230
ECC2 230
échangeur de chaleur 341
échéance 6, 412
échelle 404
écroulement du réseau 168
éditeur de liens 270
élection 139
élément
  absorbant 118
  neutre 118
embarqué 3
empreinte mémoire 183
en ligne 11
enregistrement 252
en-tête de module 270
entier signé 513
entrées/sorties 109
  analogiques 135
  numériques 132
énumération 254
environnement multiprocesseur 491
équité 142
équivalence 116
espace
  de collision 168
  de noms 275
et logique 116
Ethernet 166
  commuté 168
événement 191
exception 276
exclusion mutuelle 91, 145
exécuté 138
exécutif 182
  temps réel 181
exécution
  asynchrone 11
  synchrone 10

F

face avant 247
facteur
  d'utilisation 424-425
  de charge 424
FDDI 166, 169
fenêtre glissante 175
fichier
  d'échange 158
  exécutable 269
  temps réel 214
FIFO (*First In First Out*) 141
file temps réel 244
FIP 166, 180
flot de données 30, 251, 331
flottant dénormalisé 517
fonction 265
fréquence d'horloge 110
front montant ou descendant 134
FTP 176

G

gain 135
gamme 135
garde 186

© Dunod – La photocopie non autorisée est un délit.

GCC 246
gestion
  d'erreur en langage LabVIEW 277
  de la sécurité d'une mine 56
  des erreurs en langage C 275
gigue 410, 418
  temporelle 6
GNAT 246, 363
groupe de tâches 225

H

heap 155
hexadécimal 120
horloge
  globale 492
  temps réel (HTR) 139
hors ligne 11
hôte 23
HTTP 176
hub 167

I

I16 251
I32 251
I8 251
IBSS 169
ICANN 170
idempotence 118
IEC 617 119
IEEE 1394 129
IEEE 754 516
IEEE 802.11 168
IEEE 802.3 166
IEEE 802.3D 168
IEEE 802.4 169
IEEE 802.5 169
IEEE 802.6 169
IEEE 209
implication 116
IN 266
IN OUT 267
indicateur 251
infrastructure 169
instant critique 430
instrument virtuel 251
interblocage 149
interruption 126
  logicielle 206
  matérielle 206
intranet 173
Inverse Deadline 436
inversion de priorité 154, 471
IP 170-171
IPC 211
I-PDU (*Interaction layer Processor Data Unit*) 226
IRQ (*Interrupt ReQuest*) 206
ISO 164
ISO/IEC-9945 209
ISR 126
ITU 164

L

LabVIEW 251
langage
  Ada 246
  C 245
  LabVIEW 246
  macro 271
langages
  flots de données 246
  réactifs 16
latence due au noyau 182
laxité
  dynamique 418
  L 416
Least Laxity 443
ligne 132
little endian 388
LLC (*Logical Link Control*) 165
lois de De Morgan 119

M

MAC 165
main 266
masque 132
masquer une interruption 128
Measurement and Automation eXplorer 402
Medium Access Control 165
mémoire
  cache 115
  centrale 110
  virtuelle 158
message
  périodique 226

queue 212
méthode 27
« ESML++ » 70
DARTS 81
de conception 81
SA-RT 27
Metrowerks 239
microcontrôleur 110
micronoyau 183
microprocesseur 109
migration de tâches 500
MIL STD 806 119
Minimum Laxity 443
MLF (*Multi-Level Feedback*) 143
MMU (*Memory Management Unit*) 158
MODBUS 394
modèle OSI 164
module 269
de données 91
de traitement 86
modulo 120
moniteur 150
à la Ada 186
de Hoare 186
mot
machine 111
mémoire 110
MTU 175
multitâche coopératif 333
mutex 147, 185

N

nombre fractionnaire 515
non logique 116
normalisé 516
noyau 181
dirigé par le temps 208
dirigé par les événements 208
monolithique 181
temps réel 11, 183

O

objet 269
protégé 317
octal 120
octet 110
OIL (*OSEK Implementation Language*) 222
one shot 217
opérandes 113
Operating System (*OS*) 138
optimal 432
optimalité 409
ordonnançabilité 409
ordonnancement 10, 140
en ligne 407
global 215
local 214
mixte 215
systèmes distribués 500
systèmes multiprocesseurs 494
OSEK/VDX 221
OSEKturbo 239
ou logique 116
exclusif 116
overhead 139, 309

P

package 273
page 158
pagination 157
paquetage 267, 273
partage
de ressources 9
de ressources critiques 423
passerelle 171
pattern matching 134
période
d'étude 425
d'exécution 411
périodicité 6
périphérique d'entrées/sorties 278
PID 394
pilotage d'un four à verre 61
pilotes de périphérique 278
pipe 196
pipeline 115
point
d'entrée 111
fixe 515
pointeur 255, 258
polling 126
port 176
numérique 132
série 388
portage 396

© Dunod – La photocopie non autorisée est un délit.

POSIX 209
  1003.1b 211
précédence 420, 463
préchargement des instructions 114
préempté 138
préemptible 11, 228
  non ~ 11, 228
prefetch 114
préprocesseur 271
prêt 138
primitives 151
principe de localité 115
priorité 287
  des opérateurs 118
problème
  de la concurrence 144
  du lecteur/écrivain 148
  du producteur/consommateur 149
procédure 265
processeur 110
processus 138, 214
  de contrôle 34
  fonctionnel 30
  primitif 39, 49
profils
  de communication 230
  temps réel 220
programmation 19
programme principal 266
protocole 162
  à arbitre centralisé 169
  à héritage de priorité 475
  à priorité plafond 290, 478
    immédiat 318
PSE 220
pthread 218, 285

Q

quantum 139

R

ramasse miettes (*garbage collector*) 258
Rate Monotonic 431
  Analysis 482
Ravenscar 319
récursivité 269
réentrance 151
référencement 256
registre 111
  à décalage 262
  d'état (*flags*) 112
relais électriques 388
rendement 143
rendez-vous 198, 201
  synchronisé 204
réparti 4
représentation des entiers 123
réseau
  internet 161
  local 161, 171
réseaux
  de Petri 72
  de terrain 176
résolution 135
ressource critique 9, 145, 471
RIPE 171
RMA 482
Round Robin 437
routeur 161, 170-171, 174
routine de traitement d'interruption
  (ISR) 126, 206
RS-232 128, 388
RT FIFO 195
RTAI 243
RTEID/ORKID 240
RTEMS 240
RTLinux 241

S

scrutation 126
section critique 145
segment 174
  de code 154
  de données 154
  de pile 154
sémaphore 146, 185
  à compte 147, 185
  binaire 146-147, 185, 238
  binaire (*mutex*) 219
  compteur 238
  d'exclusion mutuelle 238
  en lecture/écriture 219
  nommé 212
séquence saturée 429
séquenceur 11, 407
  de commandes 111

serveur 173
  à scrutation 450
  ajournable 454
  périodique 447
  sporadique 457
    dynamique 459
shared memory 213
shell 235
Shortest Remaining Time First 143
signal 187, 191
  à compte 193
  apériodique 83
  asynchrone 191
  fugace 192
  mémorisé 193
  périodique 83
  synchrone 191
SMTP 176
socket 197, 363
sortie d'erreur 277
sous-programme 265
spécification globale 19
spinlocks 220
SRPT 143
stack 154
structure séquence 265
surcoût processeur 139
suspendu 138
suspensif 128
swap 158
switch 167
synchronisation 9, 87, 201, 463
  à diffusion 204
  de processus 149
système
  d'exploitation (SE) 138
  de contrôle-commande 1
  de freinage automobile 36
  multiprocesseur 491
  réparti 492

T

table
  de vérité 116
  des pages 159
tableau 254, 257
  noir 200
tâche 86, 183-184, 214
  anonyme 316
  apériodique 414, 447
  atomique 408
  basique 222
  communicante 98
  créée 184
  émettrice 194
  endormie 184
  étendue 222
  exécutée 184
  existante 184
  logicielle 83
  matérielle 83
  périodique 297
  préemptée 184
  prête 184
  réceptrice 194
  réveillée 184
  supprimée 184
  suspendue 184
tas 154
TCP 164, 173
TCP/IP 164
TDMA 166, 170
temps
  d'accès 110
  d'attente 142
  de réponse 6, 142, 417
  libre processeur 428
Test and Set Lock 220
thermocouple 388
thread 214
tick 208, 235
timer 217
  périodique 217
TOR 132
Tornado® 314
tourniquet 437
tranche canal 167
trigger 134
tube 196
type
  enregistrement 250
  énuméré 249-250

U

U16 251
U32 251

© Dunod – La photocopie non autorisée est un délit.

U8 251
UDP 173
Unicode 125
unité arithmétique et logique (UAL) 111
USB (*Universal Serial Bus*) 128

V

validation systèmes distribués 504
variable conditionnelle 189
vecteur d'interruption 206
virgule flottante 516
virtual instrument ou *vi* 251
  vi non réentrant 335
voie virtuelle 404
VxWorks 231

W

wait 187
War-Driving 169
Wi-Fi 168
Win32 239
WindView 239
WorldFIP 180

X

X.25 164
xor 116
XSI 212

Langage informatique

#define 271
#endif 272
#ifndef 272
#include 271

A

access 258
ActivateTask 224
Ada.Real_Time 326
aliased 258
and then 262
Any_Priority 317
array 257
attendre 334
attendre un multiple de 334

B

boolean 250

C

calloc 256
case 260
case 1 260
Ceiling_Locking 319
ChainTask 224
char 248
character 250
ClearEvent 225
CLOCK_CPU_TIME 217
clock_getres 218
CLOCK_MONOTONIC 217
clock_nanosleep 218
CLOCK_REALTIME 216
CLOCK_THREAD_CPU-TIME_ID 217
cluster 252

D

default 260
Default_Priority 317
delay 325
delay until 325
delta 250
DisableAllInterrupts 227
do 260
double 248

E

else 260
elsif 260
EnableAllInterrupts 227
entry 317
enum 249
errno 275
ErrorHook 229

F

float 248, 250
for 260
free 256
function 267, 318

G

GetResource 225

I

if 260
int 248
intConnect 238
integer 250
Interrupt_Priority 317

K

kill 213

L

Locking_Policy 319
long 248
long double 248
long long 248
loop 260

M

malloc 256
memcmp 255
memcpy 255
mlock 217
mlockall 217
mmap 213, 217
mq_receive 212
mq_send 212
MSG_Q_ID 311
msgQCreate 238, 308
msgQReceive 238, 307
msgQSend 238

N

nanosleep 218
natural 250

O

or else 262
others 257

P

positive 250
pragma 319
pragma Attach_Handler 327
priority 317
Priority_Queuing 319
procedure 267, 317
protected type 317
pthread_attr_destroy 287
pthread_attr_init 287
pthread_attr_setinheritsched 288
pthread_attr_setschedpolicy 288
pthread_attr_setscope 287
pthread_barrier 220
pthread_barrier_init 220
pthread_barrier_wait 220
pthread_cond 219
pthread_cond_broadcast 220, 293
pthread_cond_destroy 292
pthread_cond_init 219, 292
pthread_cond_signal 219, 292
pthread_cond_t 291
pthread_cond_timedwait 297
pthread_cond_wait 292
pthread_create 218, 285, 287
pthread_join 287
pthread_make_periodic_np 297
pthread_mutex_lock 289
pthread_mutex_t 289
pthread_mutex_unlock 289
pthread_mutexattr_init 290
pthread_mutexattr_setprioceiling 290
pthread_mutexattr_setprotocol 290
pthread_mutexattr_t 290
pthread_rwlock_rdlock 219
pthread_rwlock_wrlock 219
pthread_self 219
pthread_spin_init 220
pthread_spin_lock 220

Q

Queuing_Policy 319

R

raise 213
record 250
ReleaseResource 225
ResumeAllInterrupts 227
rwlocks 219

© Dunod – La photocopie non autorisée est un délit.

S

SCHED_FIFO 215
SCHED_OTHER 216
sched_param 288
SCHED_RR 216
SCHED_SPORADIC 216
Schedule 228
SEM_ID 310
sem_open 212
sem_post 212
sem_timedwait 299
sem_wait 212
semBCreate 238
semCCreate 238, 308
semGive 238, 308
semMCreate 238, 307
semTake 238, 308
SendDynamicMessage 226
SendMessage 226
SendZeroMessage 226
Set_Priority 316
SetEvent 225
shm_open 213
short 248
sigaction 213
sigemptyset 213
sigwait 213
sizeof 247
strcmp 257
strcpy 257
strlen 257
struct 247
SuspendAllInterrupts 227
switch 260
sysClkRateGet 309
sysClkRateSet 309

T

task type 314
Task_Dispatching_Policy 319
taskActivate 235
taskDelay 238, 305, 309
taskDelete 235
taskInit 235, 305
taskLock 237
taskPrioritySet 237
taskResume 305
taskSafe 237
taskSpawn 235, 305
taskUnlock 237
taskUnsafe 237
TerminateTask 224
tExcTask 238
Time_Span 326
timer_create 218
timer_settime 218
typedef struct 249

U

union 249
unsigned 248
unsigned char 248
unsigned long 248
unsigned long long 248
unsigned short 248
use 275
usleep 287

W

WaitEvent 225
while 260
with 275

TECHNIQUE ET INGÉNIERIE
Série EEA

GESTION INDUSTRIELLE
CONCEPTION
FROID ET GÉNIE CLIMATIQUE
MÉCANIQUE ET MATÉRIAUX
CHIMIE
ENVIRONNEMENT ET SÉCURITÉ
EEA

Francis Cottet • Emmanuel Grolleau

SYSTÈMES TEMPS RÉEL DE CONTRÔLE-COMMANDE

Conception et implémentation

Cet ouvrage présente une méthodologie complète et opérationnelle de développement des systèmes temps réel de contrôle-commande. Il permet au lecteur de :

- connaître et mettre en œuvre les méthodes de spécification et de conception ;
- définir et paramétrer l'environnement d'exécution des systèmes ;
- réaliser l'implémentation multitâche basée sur un noyau temps réel ;
- développer l'application en C, Ada ou LabVIEW™.

L'ouvrage fait également le point sur les dernières avancées dans le domaine des systèmes temps réel multitâches. De nombreux exemples industriels sont traités, permettant de comprendre puis de mettre en œuvre les principes de cette méthodologie de développement.

Ce livre s'adresse à tous les ingénieurs ou techniciens concepteurs d'applications temps réel de contrôle-commande de procédés industriels. Il est également destiné aux étudiants en informatique industrielle.

FRANCIS COTTET
est professeur à l'ENSMA (École Nationale Supérieure de Mécanique et d'Aérotechnique).

EMMANUEL GROLLEAU
est maître de conférences à l'ENSMA.

Tous deux sont chercheurs au Laboratoire d'informatique scientifique et industrielle (LISI), dans l'équipe « Système temps réel », coordonnée par Francis Cottet lui-même.

ISBN 2 10 007893 3

www.dunod.com