

cette approche opérationnelle, cela pèse lourdement sur le parallélisme potentiel des threads.

Prenons maintenant les signaux. Certains sont logiquement spécifiques aux threads, tandis que d'autres ne le sont pas. Par exemple, si un thread invoque alarm, il est logique que le signal qui en résulte soit transmis au thread original de l'appel. Or, lorsque les threads sont implémentés entièrement au sein de l'espace utilisateur, le noyau ignore leur existence ; il peut donc difficilement diriger le signal vers le bon thread. Une complication supplémentaire surgit si un processus ne peut avoir qu'une alarme en cours à un instant donné alors que plusieurs threads peuvent invoquer alarm indépendamment.

D'autres signaux, comme les interruptions clavier, ne sont pas spécifiques aux threads. Qui doit les intercepter ? Un thread désigné ? Tous les threads ? Un nouveau thread spontané qui vient d'être créé ? Par ailleurs, que se passe-t-il si un thread modifie les handlers de signal sans en informer les autres threads ? Et quid d'un thread qui souhaite intercepter un signal particulier (par exemple, l'utilisateur a appuyé sur Ctrl+C) alors qu'un autre thread, lui, veut que ce signal arrête le processus. De telles situations peuvent se produire si un ou plusieurs threads exécutent des procédures de bibliothèque standard tandis que d'autres sont développées par le programmeur. Il est clair que cela pose des problèmes de compatibilité. En général, les signaux sont suffisamment difficiles à gérer dans un environnement monothread. Le passage en multi-thread n'en facilite pas la prise en charge.

Le dernier problème est celui de la gestion de la pile. Dans de nombreux systèmes, lorsqu'une pile de processus déborde, le noyau lui alloue automatiquement de l'espace supplémentaire. Lorsqu'un processus possède plusieurs threads, il doit également avoir plusieurs piles. Si le noyau n'est pas conscient de la présence de ces piles, il ne peut pas les « agrandir » automatiquement lorsque survient une erreur de pile. En fait, il risque même de ne pas réaliser que l'erreur mémoire est due à l'augmentation de l'espace nécessaire à une pile.

Ce ne sont certes pas des problèmes insurmontables, mais ils montrent que le simple fait d'introduire des threads dans un système existant sans passer par une reconception substantielle du système est une démarche qui ne peut pas fonctionner. La sémantique des appels système doit être redéfinie, et les bibliothèques doivent être réécrites, à tout le moins. Et tout cela doit être fait de façon à assurer la compatibilité descendante avec les programmes existants pour le cas limité d'un processus ne possédant qu'un seul thread.

2.3 La communication interprocessus

Il arrive souvent que les processus aient besoin de communiquer entre eux. Par exemple, dans un pipeline du shell, la sortie du premier processus doit être passée au deuxième processus, et ainsi de suite. Il existe donc un besoin de communication entre les processus, de préférence de façon structurée et en évitant les interruptions.

Dans les sections suivantes, nous aborderons quelques-uns des problèmes liés à cette communication interprocessus (*IPC, InterProcess Communication*).

On rencontre essentiellement trois problèmes. Le premier coule de source : comment un processus fait-il pour passer des informations à un autre processus ? Le deuxième repose sur la nécessité de s'assurer que deux processus, ou plus, ne produisent pas de conflits lorsqu'ils s'engagent dans des activités critiques (deux processus tentent de récupérer le dernier Mo de mémoire). Le troisième concerne le séquencage en présence de dépendances : si le processus A produit des données et que le processus B les imprime, B doit attendre que A ait terminé pour pouvoir remplir sa tâche. Nous verrons ces trois problématiques l'une après l'autre à partir de la section suivante.

Il est important de mentionner que deux de ces problèmes s'appliquent également aux threads. Pour ce qui est du passage de l'information, cela est aisément dans la mesure où ceux-ci partagent un espace d'adressage commun (les threads sont situés dans des espaces d'adressage différents et qui ont besoin de communiquer entre eux dans le cadre de la communication interprocessus). Cependant, les deux autres problèmes, à savoir le fait qu'ils s'évitent les uns les autres et que leur séquencage soit approprié, s'appliquent bel et bien aux threads. Aux mêmes problèmes, les mêmes solutions. Nous parlerons donc de ces problèmes dans le contexte des processus, mais n'oubliez pas que les points abordés sont également applicables aux threads.

2.3.1 Les conditions de concurrence

Dans certains systèmes d'exploitation, les processus qui travaillent ensemble peuvent partager un espace de stockage commun dans lequel chacun peut lire et écrire. Cet espace de stockage partagé peut se trouver dans la mémoire principale (éventuellement dans une structure de données du noyau) ou il peut s'agir d'un fichier partagé. L'emplacement de la mémoire partagée ne modifie pas la nature de la communication, ni celle des problèmes qui peuvent survenir. Pour voir comment fonctionne la communication interprocessus dans la pratique, prenons un exemple simple, mais courant : le spouleur d'impression. Lorsqu'un processus veut imprimer un fichier, il entre son nom dans un **répertoire de spoule** spécial. Un autre processus, le **démon d'impression**, regarde périodiquement s'il y a des fichiers à imprimer ; si c'est le cas, il les imprime et supprime leurs noms du répertoire. Supposons que notre répertoire de spoule possède un grand nombre d'entrées, numérotées 0, 1, 2, et ainsi de suite, chacune pouvant accueillir un nom de fichier. Disons qu'il existe deux variables partagées : out, qui pointe vers le prochain fichier à imprimer, et in, qui pointe vers la prochaine entrée libre du répertoire. Ces deux variables peuvent très bien être conservées dans un fichier de deux mots, disponible pour tous les processus. À un instant donné, les entrées 0 à 3 sont vides (les fichiers ont été imprimés) et les entrées 4 à 6 sont pleines (avec les noms des fichiers en file d'attente pour l'impression). Alors, plus ou moins simultanément, les processus A et B décident de mettre un fichier dans la file d'attente d'impression. La figure 2.18 illustre cette situation.

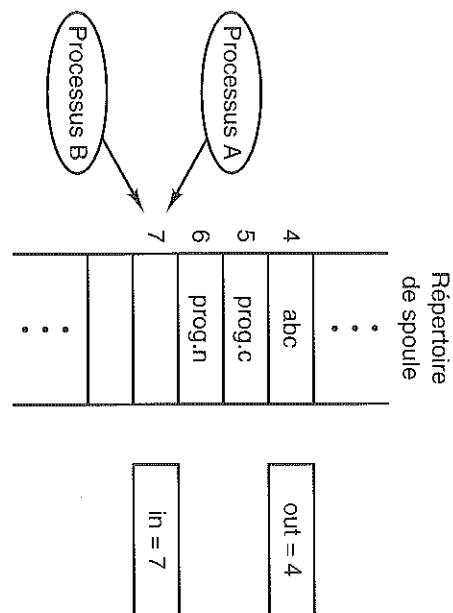


Figure 2.18 • Deux processus veulent accéder à la mémoire partagée en même temps.

Voici ce qui peut se produire dans les cas de figures où la loi de Murphy (« si quelque chose peut mal tourner, alors ça tournera mal ») s'applique. Le processus A lit la valeur de `in` et en stocke la valeur, 7, dans une variable locale appelée `next_free_slot`. À ce moment-là, une interruption se produit. La CPU, jugeant que le processus A a bénéficié de suffisamment de temps d'exécution, bascule vers le processus B. Le processus B lit la valeur de `in` et récupère également un 7. Il stocke cette valeur dans sa variable locale appelée `next_free_slot`. À ce point, les deux processus pensent que la prochaine entrée disponible est la 7.

Le processus B continue de s'exécuter. Il stocke son nom de fichier dans l'entrée 7 et actualise `in`, qui prend la valeur 8.

Finalement, le processus A s'exécute à nouveau, reprenant les choses où il les avait laissées. Il examine `next_free_slot`, y trouve un 7, et écrit son nom de fichier dans le connecteur 7, écrasant le nom que le processus B venait juste d'y placer. Ensuite, il calcule `next_free_slot + 1`, ce qui donne 8, et positionne `in` à 8. En interne, le répertoire de spoule est cohérent, ce qui fait que le démon d'impression ne remarque pas que quelque chose ne tourne pas rond. Mais le processus B ne recevra jamais de sortie. L'utilisateur B va attendre longtemps une impression qui ne viendra jamais. De telles situations — où deux processus ou plus lisent ou écrivent des données partagées et où le résultat final dépend de quel élément s'exécute à un instant donné — sont nommées **conditions de concurrence (race conditions)**.

Il n'est pas follement amusant de déboguer des programmes contenant des conditions de concurrence. Les résultats des tests sont souvent satisfaisants à première vue, mais de temps en temps il se produit des situations curieuses et inexpliquées.

2.3.2 Les

Comment utiliser les primitives de concurrence
 — impliquant la
 — écrivent dans la
 — méthodes partagées
 — s'est produite
 — gées avant la
 — primitives d'impression
 — de conception
 — dans tous les systèmes
 — Le problème n'est pas
 — façon abstraite
 — internes et dans les
 — Cependant, les
 — activités critiques
 — programmation
 — tique, ou synchronisation
 — simultanément
 — Si cette imprécision
 — tre que des programmes
 — ment les données
 — nos fins :

1. Deux processus critiques

2. Il n'est pas mis en œuvre
3. Aucun programme

4. Aucun programme
- d'autres programmes

Du point de vue du programmeur, celui illustré par la figure 2.18 illustre la condition T_1 . Un peu plus que, mais non pas tout à fait. Par conséquent, l'instant T_1 est l'instant où l'entrée in entre dans la section critique T_4 et nous sortons de la section critique T_4 .

2.3.2 Les sections critiques

Comment éviter les conditions de concurrence ? Pour éviter les problèmes de ce type — impliquant la mémoire et les fichiers partagés, ainsi que tout autre élément partagé —, il faut trouver une solution pour interdire que plusieurs processus lisent et écrivent des données partagées simultanément. **L'exclusion mutuelle** est une méthode qui permet de s'assurer que si un processus utilise une variable ou un fichier partagés, les autres processus seront exclus de la même activité. Le problème décrit s'est produit parce que le processus *B* a commencé à utiliser l'une des variables partagées avant que le processus *A* n'en ait terminé avec celle-ci. Le choix des opérations primitives appropriées pour mettre en œuvre l'exclusion mutuelle est une question de conception majeure pour tout système d'exploitation. Nous examinerons ce sujet dans tous les détails au cours des sections suivantes.

Le problème posé par les conditions de concurrence peut également être formulé de façon abstraite. De temps en temps, un processus est occupé à effectuer des traitements internes, et d'autres activités qui ne sont pas génératrices de conditions de concurrence. Cependant, les processus ont besoin d'accéder à la mémoire ou à des fichiers partagés, activités critiques susceptibles de produire des conditions de concurrence. La partie du programme à partir de laquelle on accède à la mémoire partagée se nomme **région critique**, ou **section critique**. Si l'on pouvait empêcher que deux processus se trouvent simultanément dans leurs sections critiques, on éviterait les conditions de concurrence.

Si cette implémentation permet d'agir sur ces conditions, elle ne suffit pas à permettre que des processus parallèles coopèrent de la façon appropriée et utilisent efficacement les données partagées. Quatre conditions doivent être réunies pour arriver à nos fins :

1. Deux processus ne doivent pas se trouver simultanément dans leurs sections critiques.
 2. Il ne faut pas faire de suppositions quant à la vitesse ou au nombre de processeurs mis en œuvre.
 3. Aucun processus s'exécutant à l'extérieur de sa section critique ne doit bloquer d'autres processus.
 4. Aucun processus ne doit attendre indéfiniment pour pouvoir entrer dans sa section critique.
- Du point de vue abstrait, le comportement que nous voulons mettre en œuvre est celui illustré à la figure 2.19. Le processus *A* entre dans sa section critique à l'instant T_1 . Un peu plus tard, à l'instant T_2 , le processus *B* tente d'entrer dans sa section critique, mais ne le peut pas car un autre processus se trouve déjà dans sa section critique. Par conséquent, le processus *B* est provisoirement suspendu, c'est-à-dire jusqu'à l'instant T_3 , où le processus *A* quittera sa section critique, ce qui permettra à *B* d'y entrer immédiatement. Enfin, le processus *B* va quitter sa section critique à l'instant T_4 et nous retrouverons la situation d'origine où aucun processus n'était dans sa section critique.

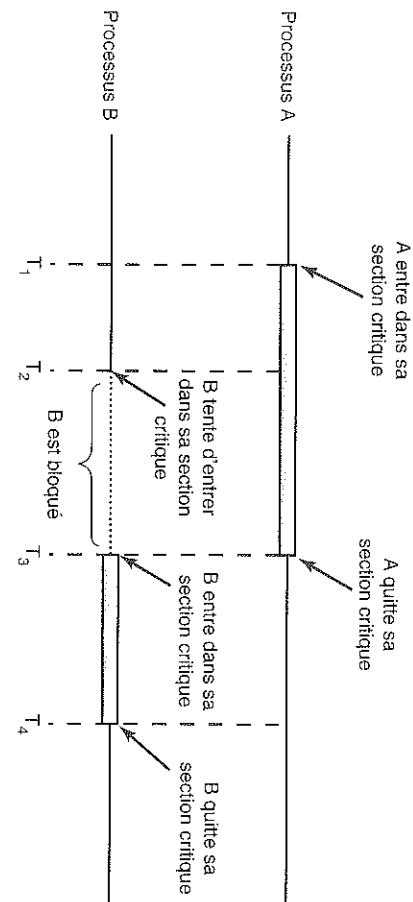


Figure 2.19 • Exclusion mutuelle à l'aide des sections critiques.

2.3.3 L'exclusion mutuelle avec attente active

Dans cette section, nous examinerons diverses propositions permettant d'obtenir une exclusion mutuelle de façon que, pendant qu'un processus est occupé à l'actualisation de la mémoire partagée dans sa section critique, aucun autre processus ne puisse entrer dans sa section critique pour y semer le désordre.

Désactivation des interruptions

La solution la plus simple est que chaque processus désactive toutes les interruptions juste après son entrée dans la section critique, et qu'il les réactive juste après l'avoir quittée. Si les interruptions sont désactivées, l'horloge ne peut envoyer d'interruptions. Le processeur ne peut basculer d'un processus à un autre que s'il reçoit des interruptions d'horloge ou autres. Mais si les interruptions sont désactivées, il ne peut plus basculer dans un autre processus. Ainsi, une fois qu'un processus a désactivé les interruptions, il peut examiner et actualiser la mémoire partagée sans craindre l'intervention d'un autre processus.

Une telle approche est rarement intéressante. En effet, il n'est pas très judicieux de donner aux processus utilisateurs le pouvoir de désactiver les interruptions. Qu'adviendrait-il si l'on ne les réactivait jamais ? Cela sonnerait le glas pour tout système. En outre, dans le cas d'un système multiprocesseur, la désactivation des interruptions n'affecte que le processeur qui a exécuté l'instruction disable. Les autres continuent de s'exécuter et peuvent accéder à la mémoire partagée.

D'autre part, il est souvent pratique pour le noyau lui-même de désactiver les interruptions, le temps d'exécuter quelques instructions pendant qu'il actualise des variables ou des listes. Si une interruption se produit pendant que la liste des processus prêts, par exemple, se trouve dans un état incohérent, des conditions de concurrence

vont se produire ressante au sein en tant que n

Alternance

La figure 2.20 illustre une autre approche programmée. Ce choix est fait en C (ou pas) est puissant. Des systèmes tels que à un instant donné, le plus inopportun de nettoyage. À la figure 2.20, des processus mémoire partagée en section critique. La petite boucle qu'elle va passer valeur attendue telle démarre

vont se produire. En conclusion, la désactivation des interruptions est souvent intéressante au sein du système d'exploitation lui-même, mais elle n'est pas appropriée en tant que mécanisme d'exclusion mutuelle pour les processus utilisateur.

Variables de verrou

Notre deuxième tentative va se tourner vers une solution logicielle. Supposons une variable unique, partagée (un verrou ou *lock*) dont la valeur initiale est de \emptyset . Lorsqu'un processus tente d'entrer dans sa section critique, il commence par tester le verrou. Si celui-ci est à \emptyset , le processus le positionne à 1 et entre en section critique. Si le verrou est déjà à 1, le processus attend qu'il passe à \emptyset . Ainsi, \emptyset signifie qu'aucun processus ne se trouve dans la section critique, et un 1 implique la présence d'un processus dans la section critique.

Malheureusement, cette solution présente le même inconvénient majeur que celui que nous avons vu au niveau du répertoire de spoule. Supposons qu'un processus lise le verrou et constate qu'il est à \emptyset . Avant qu'il n'ait pu le positionner à 1, un autre processus planifié s'exécute et le fait à sa place. Lorsque le premier processus reprend son exécution, il positionne également le verrou à 1, et les deux processus entrent dans leurs sections critiques simultanément.

Vous pensez peut-être qu'il est possible de contourner ce problème en commençant par lire la valeur du verrou, puis en la revérifiant juste avant d'y stocker quelque chose. Cela n'est pas d'une grande utilité. La concurrence intervient alors si le second processus modifie la valeur du verrou juste après que le premier processus a terminé sa seconde vérification.

Alternance stricte

La figure 2.20 illustre une troisième approche de l'exclusion mutuelle. Cet extrait de programme — à l'instar de la quasi-totalité des exemples de ce livre — est écrit en C. Ce choix est dû au fait que pratiquement tous les systèmes d'exploitation sont écrits en C (ou parfois en C++) mais rarement en Java, Modula 3 ou Pascal. Le langage C est puissant, efficace et prévisible, caractéristiques fondamentales pour développer des systèmes d'exploitation. Par exemple, Java peut manquer de capacité de stockage à un instant critique et avoir besoin d'invoquer le nettoyeur de mémoire au moment le plus inopportun. Une telle situation ne peut pas se produire en C, car il n'existe pas de nettoyage mémoire avec ce langage.

À la figure 2.20, la variable entière *turn*, initialement positionnée à \emptyset , effectue le suivи des processus dont c'est le tour d'entrer en section critique et de lire ou actualiser la mémoire partagée.

Initialement, le processus 0 inspecte la valeur de *turn*, constate qu'elle est de \emptyset et entre en section critique. Le processus 1 fait la même constatation. Il entre donc dans une petite boucle qui lui fait tester en continu la valeur de *turn* pour voir à quel moment elle va passer à 1. Le test d'une variable se poursuivant ainsi jusqu'à l'apparition d'une valeur attendue se nomme **attente active** (*busy waiting*). Il est préférable d'éviter une telle démarche dans la mesure où elle est forte consommatrice de temps processeur.

Figure 2.20 • Proposition de solutions au problème de la section critique. (a) Processus 0.(b) Processus 1. Dans les deux cas, n'oubliez pas les points-virgules à la fin des instructions while.

```

while (TRUE) {                                while (TRUE) {
    while (turn != 0) /* loop */;           while (turn != 1) /* loop */;
    critical_region();                      critical_region();
    turn = 1;                             turn = 0;
    noncritical_region();                  noncritical_region();
}
}

```

(a) (b)

Réservez l'attente active pour les situations dans lesquelles vous pouvez raisonnablement prévoir qu'elle sera brève. Un verrou assorti d'une boucle d'attente active se nomme un **spin lock**.

Lorsque le processus 0 quitte la section critique, il positionne turn à 1, pour permettre au processus 1 d'entrer en section critique. Supposons que le processus 1 en ait terminé rapidement avec sa section critique, ce qui fait que l'on retrouve nos deux processus en section non critique alors que turn est à 0. Le processus 0 exécute rapidement sa boucle ; il quitte sa section critique et positionne turn à 1. À ce stade, turn est à 1 et les deux processus s'exécutent dans leurs sections non critiques.

Ensuite, le processus 0 quitte sa section non critique et retourne en début de boucle. Malheureusement, il n'est plus autorisé à entrer dans sa section critique, car turn est à 1 et le processus 1 est occupé dans sa section non critique. Le processus 0 se retrouve alors bloqué dans sa boucle while jusqu'à ce que le processus 1 définisse turn à 0. En d'autres termes, cette situation d'attente est inenvisageable lorsque l'un des deux processus est beaucoup plus lent que l'autre.

Cette démarche entre en conflit avec la troisième condition mentionnée plus haut : le processus 0 est bloqué par un processus, hors de sa section critique. Pour revenir à notre répertoire de spoule, si l'on établissait un parallèle entre la section critique et la lecture/écriture du répertoire, le processus 0 ne serait pas autorisé à imprimer un autre fichier, car le processus 1 serait occupé à une autre tâche.

Dans les faits, cette solution implique que les deux processus alternent strictement leurs entrées en section critique, comme lors de la mise de fichiers en spoule. Aucun processus ne pourrait donc mettre en file d'attente deux fichiers à la suite. Si cette implémentation évite toutes les conditions de concurrence, elle ne peut être considérée comme une solution sérieuse dans la mesure où elle ne respecte pas la troisième condition.

Solution de Peterson

Combinant les notions d'intervention tour à tour des processus à celles des variables verrou et des variables d'avertissement (*warning variables*), un mathématicien néerlandais, T. Dekker, fut le premier à mettre au point une solution logicielle à l'exclusion mutuelle, ne faisant pas appel à l'alternance stricte. En 1981, G. L. Peterson découvrit une méthode plus simple pour obtenir le même résultat, rendant obsolète la solution de Dekker. L'algorithme de Peterson est illustré à la figure 2.21.

Il se compose de prototypes de fonctions utilisées. Cependant, il existe plusieurs types dans cet exemple.

Figure 2.21 • Solution de Peterson

```

#define FALSE
#define TRUE
#define N
int interest;
int turn;
void enter_region();
void leave_region();
int other = 0;
int interes = 0;
turn = 0;
while (1)
{
    if (turn == 0 && interes == 0)
    {
        interest = 1;
        turn = 1;
        enter_region();
    }
    else if (turn == 1 && interes == 0)
    {
        interest = 1;
        turn = 0;
        leave_region();
    }
    else if (turn == 0 && interes == 1)
    {
        interest = 0;
        turn = 1;
        enter_region();
    }
    else if (turn == 1 && interes == 1)
    {
        interest = 0;
        turn = 0;
        leave_region();
    }
}

```

Avant d'utiliser ce programme, nous devons déclarer les prototypes de fonctions. Voyons comment il fonctionne. Le processus 0 entre dans sa section critique. Il met son état de variable interest à 1 et positionne son état de variable turn à 1. Étant donné qu'il est le seul à exécuter l'instruction interested [processus 0 au début],

Prenons maintenant l'exemple du processus 1. Il effectue l'opération d'entrée placée et percevante. Il vérifie si turn est à 1. [] ne l'exécute] si l'état de la variable turn est à 1, il entre dans la section critique.

Il se compose de deux procédures développées en C ANSI, ce qui signifie que les prototypes de fonctions doivent être fournis pour toutes les fonctions définies et utilisées. Cependant, pour des raisons de place, nous ne reproduirons pas les prototypes dans cet exemple et les suivants.

Figure 2.21 • Solution de Peterson pour l'exclusion mutuelle.

```
#define FALSE 0
#define TRUE 1
#define N 2
int turn;
int interested[N];

void enter_region(int process); /* le processus est 0 ou 1 */
{
    int other;
    other = 1 - process;
    /* nombre de processus */
    /* à qui le tour ? */
    /* toutes valeurs initialement
       0 (FALSE) */

    interested[process] = TRUE;
    /* montre que vous êtes intéressé */
    turn = process;
    /* définit indicateur */
    while (turn == process && interested[other] == TRUE)
        /* instruction null */;

}

void leave_region(int process) /* processus : qui quitte */
{
    interested[process] = FALSE;
    /* indique départ de la
       section critique */
}
```

Avant d'utiliser les variables partagées (à savoir d'entrer en section critique), chaque processus appelle `enter_region` avec son propre numéro de processus, 0 ou 1, en tant que paramètre. Cet appel le met en attente, si nécessaire, jusqu'à ce qu'il puisse entrer. Une fois qu'il en a terminé avec les variables partagées, le processus appelle `leave_region` pour autoriser un autre processus à entrer.

Voyons comment fonctionne cette solution. Au départ, aucun des processus ne se trouve dans sa section critique. C'est alors que le processus 0 appelle `enter_region`. Il montre son intérêt en définissant son élément de tableau et en positionnant `turn` à 0. Étant donné que le processus 1 n'est pas intéressé, `enter_region` retourne immédiatement. Si le processus 1 appelle maintenant `enter_region`, il attend jusqu'à ce que `interested[0]` passe à FALSE, événement qui ne peut se produire qu'une fois que le processus 0 aura appelé `leave_region` pour quitter la section critique.

Prenons maintenant le cas de figure où les deux processus appellent `enter_region` simultanément. Tous deux vont stocker leur numéro de processus dans `turn`. C'est l'opération de stockage la plus tardive qui est prise en compte ; la première est remplacée et perdue. Supposons que le processus 1 soit stocké en dernier, ce qui fait que `turn` est à 1. Lorsque les deux processus arrivent à l'instruction `while`, le processus 0 ne l'exécute pas et entre en section critique. Le processus 1 boucle et n'entre pas en section critique tant que le processus 0 n'a pas quitté la sienne.

Instruction TSL

Étudions maintenant une proposition qui sollicite un peu d'aide de la part du matériel. De nombreux ordinateurs, et notamment ceux qui ont été conçus pour accueillir plusieurs processeurs, prennent en charge l'instruction :

```
TSL RX,LOCK
```

(*Test and Set Lock*, tester et définir le verrou) qui fonctionne de la manière suivante. Elle lit le contenu du mot mémoire lock dans le registre RX, puis stocke une valeur différente de 0 à l'adresse mémoire lock. Les opérations qui consistent à lire le mot et à y stocker une valeur sont absolument indivisibles ; aucun autre processeur ne peut accéder au mot mémoire tant que l'instruction n'est pas terminée. Le processeur exécutant l'instruction TSL verrouille le bus mémoire pour interdire aux autres processeurs d'accéder à la mémoire tant qu'il n'a pas terminé. Pour exploiter l'instruction TSL, on fait appel à une variable partagée, lock, afin de coordonner l'accès à la mémoire partagée. Lorsque lock est à 0, n'importe quel processus peut la positionner à 1 via l'instruction TST, puis lire ou écrire dans la mémoire partagée. Cela fait, le processus répositionne lock à 0 à l'aide d'une instruction move ordinaire.

Comment utiliser cette instruction pour empêcher deux processus d'entrer simultanément dans leurs sections critiques ? La solution est illustrée à la figure 2.22 qui montre une sous-routine en langage assembleur fictif (mais type), composée de quatre instructions. La première copie l'ancienne valeur de lock dans le registre, puis la positionne à 1. Ensuite, l'ancienne valeur est comparée à 0. Si elle est différente de 0, cela signifie que lock était déjà positionnée ; le programme retourne simplement au début et refait le test. lock finira bien par être à 0, lorsque le processus actuellement en section critique en aura terminé. Et la sous-routine sera retournée avec lock défini. Il est très simple de supprimer le verrou. Le programme stocke un 0 dans la variable lock, et aucune instruction spéciale n'est nécessaire.

Figure 2.22 • Entrer et sortir de la section critique à l'aide de l'instruction TSL.

```
enter_region:
    TSL REGISTER,LOCK | copie lock dans le registre et la définit à 1
    CMP REGISTER,#0   | lock était-elle à 0 ?
    JNE enter_region  | si elle n'était pas à 0, boucle
    RET               | retourne à l'appelant ; entre en section critique

leave_region:
    MOVE LOCK,#0      | stocke un 0 dans lock
    RET               | retourne à l'appelant
```

L'une des solutions au problème des sections critiques devient maintenant plus palpable. Avant d'entrer dans sa section critique, le processus appelle `enter_region`, qui se met en position d'attente active jusqu'à ce que le verrou soit libre. Ensuite, il récupère le verrou et retourne. Une fois qu'il a terminé, le processus invoque `leave_region` qui stocke un 0 dans le verrou. Comme pour toutes les solutions reposant sur les sections critiques, les processus doivent appeler `enter_region` et `leave_region` aux bons moments pour que la méthode fonctionne. Si un processus « triche », l'exclusion mutuelle échoue.

2.3.4

Tant la production
de produits
qu'en scène
peut prendre
de quatre à
six mois.

Cette ap-
plication
également
H, de pro-
font que
alors que
a terminé
l'appelant
wake up
sleep et
employé
problème

Examinons
bloquera
l'autorité
avec la pre-
l'appelant
wake up
sleep et
employé
problème

consomma-
buffet
produces
récupére
produces
en scène
Les prob-
dans le pro-
d'entrer et
ou plusieu-
récupére
sommant
qui va le
Cette ap-
concurrent
du nombre
que nous

2.3.4 Le sommeil et l'activation

Tant la solution de Peterson que celle de l'instruction TSL sont bonnes, mais toutes deux ont pour inconvénient de faire appel à l'attente active. Voici ce qu'elles font essentiellement : lorsqu'un processus souhaite entrer dans sa section critique, il vérifie si la « porte » est ouverte. Si ce n'est pas le cas, le processus s'installe dans une petite boucle en attendant l'ouverture.

Cette approche est naturellement consommatrice de temps processeur, mais elle peut également avoir des effets indésirables. Prenons un ordinateur avec deux processus : H , de priorité supérieure, et L , de priorité inférieure. Les règles d'ordonnancement font que H s'exécute chaque fois qu'il se trouve en état prêt. À un moment donné, alors que L est dans sa section critique, H se trouve prêt à l'exécution (par exemple, il a terminé une opération d'E/S). H entre en attente active. Mais, étant donné que L ne peut pas être ordonné tant que H est en cours d'exécution, L n'a jamais l'occasion de quitter sa section critique et H boucle sans fin. C'est ce que l'on appelle parfois le problème de l'inversion des priorités.

Examinons maintenant quelques primitives de communication interprocessus qui bloquent au lieu de consommer du temps processeur lorsqu'elles n'obtiennent pas l'autorisation d'entrer dans leur section critique. L'une des plus simples fonctionne avec la paire sleep et wakeup. sleep est un appel système qui provoque le blocage de l'appelant. Celui-ci est suspendu jusqu'à ce qu'un autre processus le réveille. L'appel wakeup prend un paramètre, désignant le processus à réveiller. Il arrive également que sleep et wakeup prennent tous deux un paramètre qui est une adresse mémoire employée pour relier les sleep et les wakeup entre eux.

Problème du producteur-consommateur

Pour illustrer l'emploi de ces primitives, étudions le problème du **producteur-consommateur** (également connu sous le nom de **tampon délimité**, ou *bounded-buffer*). Deux processus partagent un tampon commun de taille fixe. L'un d'eux, le producteur, place des informations dans le tampon ; l'autre, le consommateur, les récupère. Il serait possible de généraliser le problème de façon à tenir compte de m producteurs et n consommateurs, mais nous nous contenterons de l'exemple mettant en scène deux éléments pour simplifier la compréhension de la solution.

Les problèmes se produisent lorsque le producteur souhaite placer un nouvel élément dans le tampon alors que ce dernier est déjà plein. La solution pour le producteur est d'entrer en sommeil, pour être réveillé lorsque le consommateur aura supprimé un ou plusieurs éléments du tampon. De la même façon, si le consommateur souhaite récupérer un élément dans le tampon et qu'il constate que celui-ci est vide, il entre en sommeil jusqu'à ce que le producteur ait placé quelque chose dans le tampon, opération qui va le réveiller.

Cette approche semble assez simple, mais elle conduit au même type de conditions de concurrence que celles rencontrées avec le répertoire de spoule. Pour effectuer le suivi du nombre d'éléments présents dans le tampon, nous avons besoin d'une variable, que nous appellerons count. Si le nombre maximal d'éléments que le tampon peut

contenir est N, le code du producteur commence par effectuer un test pour vérifier que la valeur de count est N. Dans l'affirmative, le producteur entre en sommeil ; dans la négative, il ajoute un élément au tampon et incrémente count.

Le code du consommateur est comparable : il consiste d'abord à tester count pour voir si elle est à 0. Si c'est le cas, le consommateur entre en sommeil. Si count est différente de 0, le code récupère un élément et décrémente le compteur. Chacun des processus effectue également un test pour déterminer s'il faut réveiller l'autre. La figure 2.23 montre les codes du producteur et du consommateur.

Figure 2.23 • Le problème du producteur-consommateur, avec une condition de concurrence fatale.

```
#define N 100
int count = 0;
void producer(void)
{
    int item;
    while (TRUE) {
        item = produce_item();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
    void consumer(void)
    {
        int item;
        while (TRUE) {
            /* se répète en continu */
            /* si le tampon est vide,
               entre en sommeil */
            item = remove_item();
            count = count - 1;
            if (count == N - 1) wakeup(producer);
            consume_item(item);
        }
    }
}
```

Pour exprimer des appels système tels que `sleep` et `wakeup` en C, nous allons vous montrer comme des appels à des routines de bibliothèque. Celles-ci n'appartiennent pas à la bibliothèque C standard, mais elles sont probablement disponibles sur n'importe quel système reconnaissant ces appels système. Les procédures

`insert_item`
des éléments

Retournons
décompte n'
à 0. À cet in-

consommateu-

ment dans l'é-

appelle `wakeu`

Malheureuse-
perdu. Lorsq

cédemment

remplira le t

pour toujour

Le problème

sommeil est

La solution c

Lorsqu'un wa

tionné. Plus t

d'éveil est la

mise dans un

des exemples

ne sera pas s

d'attente d'é

principe le p

2.3.5 Les
telle était la s
type de variab
gistrés pour r
variable un s
wakeu n'étais
cours.
Dijkstra prop
sleep et wake
si sa valeur es
stocké) et pou
sans que down
tion et éventuel
action atomiqu
démarré, aucu

`insert_item` et `remove_item`, qui n'apparaissent pas, prennent en charge le placement des éléments dans le tampon et leur retrait.

Retournons à la condition de concurrence. Elle peut se produire parce que l'accès décompte n'est pas constraint. La situation suivante pourrait ainsi se faire jour. Le tampon est vide ; le consommateur vient de lire le décompte pour constater qu'il est à 0. À cet instant, l'ordonnanceur décide d'arrêter provisoirement l'exécution du consommateur et de commencer à exécuter le producteur. Ce dernier insère un élément dans le tampon, incrémenté le décompte et constate qu'il est alors à 1. Partie du principe qu'il était à 0 et que le consommateur était en sommeil, le producteur appelle `wakeup` pour réveiller le consommateur.

Malheureusement, le consommateur n'est pas en sommeil et le signal `wakeup` perdu. Lorsque le consommateur s'exécute enfin, il teste la valeur du décompte précédemment lue ; puisqu'elle est de 0, il se met en sommeil. Tôt ou tard, le producteur remplira le tampon et se mettra également en sommeil. Et les deux « dormiront » pour toujours...

Le problème réside dans le fait qu'un `wakeup` envoyé à un processus non encore en sommeil est perdu. S'il n'était pas perdu, l'ensemble fonctionnerait comme attendu. La solution consiste à ajouter au tableau un bit d'attente d'éveil (*wakeup waiting bit*). Lorsqu'un `wakeup` est envoyé à un processus non encore en sommeil, ce bit est positionné. Plus tard, lorsque le processus tente de se mettre en sommeil, si le bit est positionné, il est remis à 0, mais le processus reste en éveil. Le bit d'attente d'éveil est la « tirelire » des signaux d'éveil. Alors que cette méthode nous sauve mise dans un exemple simple comme celui-ci, il n'est guère compliqué de construire des exemples impliquant trois processus ou plus, dans lesquels un bit d'attente d'éveil ne sera pas suffisant. Il faudra alors faire appel à un correctif pour ajouter un bit d'attente d'éveil supplémentaire, voire 8 ou 32 bits supplémentaires, mais dans ce principe le problème reste entier.

2.3.5 Les sémaphores

Telle était la situation, en 1965, lorsque E. W. Dijkstra suggéra d'utiliser un nouveau type de variable entière qui devait permettre de décompter le nombre de `wakeup` enregistrés pour un usage ultérieur. Dans sa proposition, il appelait ce nouveau type variable un **sémaphore**. Celui-ci pouvait avoir une valeur nulle, indiquant qu'aucun `wakeup` n'était enregistré ; ou une valeur positive, si un ou plusieurs `wakeup` étaient en cours.

Dijkstra proposait d'exploiter deux opérations, `down` et `up` (des généralisations de `sleep` et `wakeup`). En voici le principe. L'opération `down` sur un sémaphore détermine si sa valeur est supérieure à 0. Si c'est le cas, elle la décrémente (en utilisant un `wake` stocké) et poursuit son activité. Si la valeur est de 0, le processus est placé en sommeil sans que `down` ne se termine pour l'instant. Les activités de vérification, de modification et éventuellement de mise en sommeil sont toutes effectuées dans le cadre d'un

terminée ou bloquée. Cette atomicité est essentielle à la résolution des problèmes de synchronisation, ainsi que pour éviter les conditions de concurrence.

L'opération up incrémente la valeur du sémaphore concerné. Si un ou plusieurs processus se trouvaient en sommeil sur ce sémaphore, incapables de terminer une opération down antérieure, l'un d'eux est choisi (aléatoirement) par le système et est autorisé à terminer son down. Ainsi, une fois un up accompli sur un sémaphore contenant des processus en sommeil, le sémaphore sera toujours à 0, mais il contiendra un processus en sommeil en moins. L'opération d'incrémantation du sémaphore et d'éveil d'un processus est également indivisible. Aucun processus ne se bloque jamais en faisant un up (et aucun processus ne se bloque jamais en faisant un wakeup, comme précédemment décrit).

Dans l'article original de Dijkstra, les noms *p* et *v* étaient employés au lieu de *down* et *up*, respectivement, mais ces deux lettres n'ont pas de signification mnémotechnique pour les personnes ne parlant pas le néerlandais. C'est pourquoi nous en resterons à *down* et *up*. Ces termes ont été introduits pour la première fois en Algol 68.

Résolution du problème du producteur-consommateur avec des sémaphores

Les sémaphores résolvent le problème des wakeup perdus, comme le montre la figure 2.24. Il est essentiel qu'ils soient implémentés de façon indivisible. La méthode habituelle consiste à implémenter *up* et *down* en tant qu'appels système. Le système d'exploitation désactive toutes les interruptions pendant un très court laps de temps, durant lequel il teste le sémaphore, l'actualise et place si nécessaire le processus concerné en sommeil. Étant donné que l'ensemble de ces actions ne prend que quelques instructions, aucun dommage ne découle de la désactivation des interruptions. Si plusieurs processeurs sont utilisés, chaque sémaphore doit être protégé par une variable verrou, avec l'instruction *TSL*, pour s'assurer que seul un processus a la fois sonde la valeur du sémaphore. Comprenez bien que si *TSL* intervient pour empêcher plusieurs processeurs d'accéder au sémaphore en même temps, cette méthode est très différente de l'attente active où le producteur (ou le consommateur) doit attendre que l'autre ait rempli ou vidé le tampon. Avec les sémaphores, l'opération ne prendra que quelques microsecondes, alors qu'autrement le délai risque d'être nettement plus long.

Cette solution utilise trois sémaphores : *full*, pour compter le nombre d'emplacements occupés ; *empty*, pour compter le nombre d'emplacements vides ; *mutex*, pour s'assurer que le producteur et le consommateur n'accèdent pas simultanément au tampon. Au départ, *full* est à 0, *empty* est égal au nombre de connecteurs dans le tampon, et *mutex* est égal à 1. Les sémaphores qui sont initialisés à 1 et qui sont utilisés par deux processus ou plus pour faire en sorte que seul l'un d'entre eux pourra entrer en section critique à un instant donné, sont appelés **sémaphores binaires**. Si chaque processus effectue un *down* juste avant d'entrer en section critique et un *up* juste avant de la quitter, l'exclusion mutuelle est garantie.

Maintenant que nous disposons d'une bonne primitive de communication interprocessus, retournons à notre séquence d'interruptions de la figure 2.5. Dans un système utilisant les sémaphores, la manière la plus naturelle de masquer les interruptions consiste à avoir un sémaphore, initialement défini à 0, associé à chaque périphérique d'E/S. Juste après avoir démarré un périphérique d'E/S, le processus effectue un down sur le sémaphore associé et se bloque immédiatement. Lorsque l'interruption survient, le handler d'interruptions lance un up sur le sémaphore associé ; le processus est alors à nouveau prêt à s'exécuter. Selon ce modèle, l'étape 5 de la figure 2.5 consiste à effectuer un up sur le sémaphore du périphérique ; ainsi, lors de l'étape 6, l'ordonnanceur sera en mesure d'exécuter le gestionnaire du périphérique. Naturellement, si plusieurs processus sont prêts, l'ordonnanceur peut choisir d'exécuter un processus plus important. Nous verrons quelques algorithmes d'ordonnancement plus loin dans ce chapitre.

Dans l'exemple de la figure 2.24, nous avons en réalité exploité les sémaphores de deux manières différentes. Cette différence est suffisamment importante pour mériter quelques explications. Le sémaphore mutex est employé pour l'exclusion mutuelle. Il est conçu afin de garantir qu'un seul processus à la fois accédera en lecture et en écriture au tampon et aux variables associées. Cette exclusion mutuelle est indispensable pour éviter tout désordre. À la section suivante, nous étudierons l'exclusion mutuelle et verrons comment aller plus loin dans ce domaine.

Figure 2.24 • Le problème du producteur-consommateur et les sémaphores.

```
#define N 100          /* nombre d'emplacements dans le tampon */
typedef int semaphore; /* les sémaphores sont un type de variable int
                           spécial */
semaphore mutex = 1;   /* contrôle l'accès à la section critique */
semaphore empty = N;   /* compte les emplacements vides dans le tampon */
semaphore full = 0;    /* compte les emplacements pleins */
void producer(void)
{
    int item;
    while (TRUE) {           /* TRUE est la constante 1 */
        item = produce_item(); /* génère quelque chose à placer
                                   dans le tampon */
        down(&empty);         /* décrémente le décompte des
                                   emplacements vides */
        down(&mutex);          /* entre en section critique */
        insert_item(item);    /* place un nouvel élément dans le
                                   tampon */
        /* quitte la section critique */
        up(&mutex);            /* incrémente le décompte des
                                   emplacements pleins */
    }
}
void consumer(void)
{
    int item;
    while (TRUE) {           /* boucle sans fin */
        /* ... */
    }
}
```

```

down(&full);           /* décrémente le décompte des
                      emplacements pleins */
down(&mutex);          /* entre en section critique */
item = remove_item(); /* prélève un élément dans le tampon */
up(&mutex);            /* quitte la section critique */
up(&empty);            /* incrémente le décompte des
                      emplacements vides */
consume_item(item);  /* fait quelque chose avec l'élément */
}

```

Les sémaphores servent également à faire de la **synchronisation**. Les sémaphores full et empty sont nécessaires pour garantir que certaines séquences d'événements se produisent ou ne se produisent pas. Dans ce cas de figure, ils font en sorte que le producteur cesse de s'exécuter lorsque le tampon est plein, et que le consommateur cesse de s'exécuter quand il est vide. Il s'agit d'un usage différent de celui de l'exclusion mutuelle.

2.3.6 Les mutex

Quand des décomptes ne sont pas nécessaires, on utilise parfois une version simplifiée des sémaphores, que l'on appelle des mutex. Ceux-ci ne servent qu'à prendre en charge l'exclusion mutuelle pour certaines ressources partagées ou certaines portions de code. Ils sont faciles à mettre en œuvre et efficaces. Cela les rend particulièrement intéressants dans les paquetages de threads entièrement implémentés dans l'espace utilisateur.

Un mutex est une variable qui peut prendre deux états : déverrouillé ou verrouillé. Par conséquent, un seul bit est nécessaire pour le représenter, même si dans la pratique on utilise souvent un entier, où 0 signifie déverrouillé et toutes les autres valeurs signifient verrouillé. Deux procédures interviennent avec les mutex. Lorsqu'un thread (ou un processus) a besoin d'accéder à sa section critique, il invoque mutex_lock. Si le mutex est déverrouillé (la section critique est donc disponible), l'appel réussit et le thread appelant est libre d'entrer en section critique.

Parmi eux, si le mutex est déjà verrouillé, le thread appelant est bloqué jusqu'à ce que le thread en section critique en ait terminé et appelle mutex_unlock. Si plusieurs threads sont bloqués sur le mutex, l'un d'eux est choisi aléatoirement pour prendre possession du verrou.

Les mutex sont si simples qu'il est aisé de les implémenter dans l'espace utilisateur si une instruction TSL est présente. La figure 2.25 montre le code de mutex_lock et mutex_unlock pour une utilisation avec un package de threads dans l'espace utilisateur.

Figure 2.25 • Implémentation de mutex_lock et mutex_unlock.

mutex_lock:	
TSL REGISTER,MUTEX	copie mutex pour enregistrer et définir mutex à 1
CMP REGISTER,#0	mutex était-il à 0 ?
JZE ok	si oui, il n'était pas verrouillé, donc retourne

Le code de mutex_lock commence par une instruction CALL qui appelle la fonction mutex_lock. Mais il existe aussi une autre manière d'utiliser un mutex, qui n'utilise pas de fonction mais qui se présente comme une instruction JMP. C'est ici que nous avons une autre instruction JMP qui va au label OK. Mais il existe une autre manière d'utiliser un mutex, qui se présente comme une instruction RET. C'est ici que nous avons une autre instruction RET qui va au label OK. Avec les threads, il existe une autre manière d'utiliser un mutex, qui se présente comme une instruction CALL qui appelle la fonction mutex_lock. Mais il existe aussi une autre manière d'utiliser un mutex, qui se présente comme une instruction JMP qui va au label OK. Et avec les threads, il existe une autre manière d'utiliser un mutex, qui se présente comme une instruction RET qui va au label OK.

C'est ici que nous avons une autre manière d'utiliser un mutex, qui se présente comme une instruction CALL qui appelle la fonction mutex_lock. Mais il existe aussi une autre manière d'utiliser un mutex, qui se présente comme une instruction JMP qui va au label OK. Et avec les threads, il existe une autre manière d'utiliser un mutex, qui se présente comme une instruction RET qui va au label OK.

Le système d'exploitation fondé sur les threads offre de nombreux avantages. Par exemple, il est possible d'avoir plusieurs threads exécutant simultanément sur un seul processeur à plusieurs coeurs. Avec les threads, il est possible d'avoir plusieurs threads exécutant simultanément sur un seul processeur à plusieurs coeurs. Et avec les threads, il est possible d'avoir plusieurs threads exécutant simultanément sur un seul processeur à plusieurs coeurs.

Et avec les threads, il est possible d'avoir plusieurs threads exécutant simultanément sur un seul processeur à plusieurs coeurs. Avec les threads, il est possible d'avoir plusieurs threads exécutant simultanément sur un seul processeur à plusieurs coeurs.

Et avec les threads, il est possible d'avoir plusieurs threads exécutant simultanément sur un seul processeur à plusieurs coeurs. Avec les threads, il est possible d'avoir plusieurs threads exécutant simultanément sur un seul processeur à plusieurs coeurs.

Et avec les threads, il est possible d'avoir plusieurs threads exécutant simultanément sur un seul processeur à plusieurs coeurs. Avec les threads, il est possible d'avoir plusieurs threads exécutant simultanément sur un seul processeur à plusieurs coeurs.

Et avec les threads, il est possible d'avoir plusieurs threads exécutant simultanément sur un seul processeur à plusieurs coeurs. Avec les threads, il est possible d'avoir plusieurs threads exécutant simultanément sur un seul processeur à plusieurs coeurs.

```

CALL thread_yield | mutex est occupé ; planifie un autre thread
JMP mutex_lock | réessaye plus tard
ok:
RET

mutex_unlock:
MOVE MUTEX,#0 | stocké un 0 dans mutex
RET | retourne à l'appelant

```

Le code de `mutex_lock` est comparable à celui de `enter_region` (voir figure 2.22), mais il existe une différence cruciale entre les deux. Lorsque `enter_region` ne parvient pas à entrer dans la section critique, il continue de tester inlassablement le verrou (attente active). Tôt ou tard, un autre processus est ordonné pour exécution, et le processus détenant le verrou finit par s'exécuter et par libérer ce dernier.

Avec les threads, la situation est différente dans la mesure où il n'existe pas d'horloge chargée d'arrêter les threads qui s'exécutent depuis trop longtemps. Par conséquent, un thread qui tente d'acquérir un verrou selon la méthode de l'attente active bouclera indéfiniment et n'obtiendra jamais le verrou ; en effet, il ne permet jamais à un autre thread de s'exécuter pour libérer le verrou.

C'est ici qu'intervient la différence entre `enter_region` et `mutex_lock`. Lorsque le second n'arrive pas à obtenir un verrou, il invoque `thread_yield` pour céder le processeur à un autre thread. Il n'y a donc pas d'attente active. La prochaine fois que le thread s'exécutera, il testera le verrou une nouvelle fois.

Etant donné que `thread_yield` est juste un appel à l'ordonnanceur de threads dans l'espace utilisateur, il est très rapide. Par conséquent, ni `mutex_lock` ni `mutex_unlock` n'ont besoin d'effectuer des appels au noyau. En les exploitant, les threads utilisateurs peuvent se synchroniser entièrement dans l'espace utilisateur grâce à des procédures qui se satisfont de quelques instructions.

Le système mutex que nous venons de décrire est un ensemble d'appels « bruts de fondrie ». Quel que soit le logiciel développé, la demande en fonctionnalités est toujours plus importante, et les primitives de synchronisation ne font pas exception. Par exemple, il arrive qu'un paquetage de threads inclue un appel `mutex_tryLock` qui, soit obtient le verrou, soit retourne un code d'échec, mais qui ne se bloque pas. Un tel appel donne au thread la possibilité de décider de sa prochaine activité en présence d'une alternative à la simple attente.

Jusqu'à présent, nous n'avons fait qu'effleurer un problème qui mérite pourtant des explications claires. Avec un paquetage de threads dans l'espace utilisateur, il n'y a pas d'inconvénient à ce que plusieurs threads accèdent au même mutex dans la mesure où tous les threads interviennent dans un espace d'adressage commun. Il reste que, avec la plupart des solutions déjà traitées — comme l'algorithme de Peterson et les sémaphores —, on part du principe (non dit) que plusieurs processus ont accès à au moins un espace de mémoire partagée, éventuellement à un seul mot... mais à quelle chose. Or, si les processus ont des espaces d'adressage disjoints, comment vont-ils se partager la variable `turn` de l'algorithme de Peterson, ou des sémaphores, ou encore un tampon commun ?

Il y a deux réponses. D'une part, certaines structures de données partagées, comme les sémaphores, peuvent être stockées dans le noyau ; on ne peut alors y accéder que via des appels système. Cette approche élimine le problème. D'autre part, la plupart des systèmes d'exploitation modernes (y compris UNIX et Windows) proposent une méthode permettant aux processus de partager une portion de leur espace d'adresse avec d'autres processus. De cette façon, les tampons et autres structures de données peuvent être partagés. Dans le pire des cas, si aucune autre solution n'est opérationnelle, on peut faire appel à un fichier partagé.

Si deux processus ou plus partagent tout ou partie de leur espace d'adressage, la distinction entre les processus et les threads devient floue. Mais elle existe toujours. Deux processus partageant un espace d'adressage commun continuent d'avoir leurs propres fichiers ouverts, timers d'alerte et autres propriétés. En revanche, les threads d'un même processus se partagent tout cela. Et il reste vrai que plusieurs processus partageant un espace d'adressage commun n'auront jamais l'efficacité des threads utilisateur dans la mesure où le noyau est fortement impliqué dans leur gestion.

2.3.7 Les moniteurs

La communication interprocessus est-elle devenue une sinécure grâce aux sémaphores ? Ne rêvons pas, et examinons attentivement l'ordre des down avant d'insérer ou de supprimer des éléments dans le tampon de la figure 2.24. Supposons que les deux down du code du producteur soient intervenus dans l'ordre inverse, mutex ayant été décrémenté avant empty. Si le tampon est totalement plein, le producteur va se bloquer si mutex est à 0. Par conséquent, la prochaine fois que le consommateur essaiera d'accéder au tampon, il fera un down sur mutex (à 0) et se bloquera également. Les deux processus pourraient ainsi rester bloqués indéfiniment et aucune autre tâche ne pourrait prendre le relais. Cette situation malheureuse se nomme un inter-bloge. Nous y reviendrons en détail au chapitre 3.

Nous soulignons ce problème pour vous montrer à quel point vous devez vous montrer attentif dans l'emploi des sémaphores. La moindre erreur serait fatale à l'exécution de votre programme. C'est un peu comme programmer en langage d'assemblage, mais en pire, car les erreurs sont des conditions de concurrence, des interblocages ou d'autres formes de comportements imprévisibles ou impossibles à reproduire.

Pour faciliter le développement de programmes qui fonctionnent, Hoare et Brinch Hansen ont mis au point une primitive de synchronisation de haut niveau, appelée moniteur. Vous le verrez, les deux propositions sont légèrement différentes. Un moniteur est une collection de procédures, de variables et de structures de données qui sont toutes regroupées dans un module spécial. Les processus peuvent appeler les procédures d'un moniteur chaque fois qu'ils le souhaitent, mais ils ne peuvent pas accéder directement aux structures de données internes du moniteur à partir de procédures déclarées à l'extérieur du moniteur. La figure 2.26 illustre un moniteur écrit dans un langage imaginaire, appelé Sabir Pascal.

Figure 2.26 • Un moniteur.

```

monitor exemple
    integer i;
    condition c;
procedure producteur( );
    .
    .
    .
end monitor;

procedure consommateur( );
    .
    .
    .
end;

```

Les moniteurs ont une propriété importante qui les rend intéressants pour faire de l'exclusion mutuelle : à un instant donné, un seul processus peut être actif dans le moniteur. Les moniteurs sont des constructions du langage de programmation, ce qui fait que le compilateur sait qu'ils sont spécifiques : il gère les appels aux procédures de moniteur différemment des autres appels. Généralement, lorsqu'un processus appelle une procédure de moniteur, les premières instructions déterminent si un autre processus est actuellement actif au sein du moniteur. Si c'est le cas, le processus appelant est suspendu jusqu'à ce que l'autre processus ait quitté le moniteur. Sinon, il peut y entrer.

C'est au compilateur qu'il revient d'implémenter l'exclusion mutuelle sur les entrées dans le moniteur, mais il est courant d'employer un mutex ou un sémaphore binaire. Étant donné que c'est le compilateur — et non le programmeur — qui prend en charge l'exclusion mutuelle, il y a moins de risques que les choses se passent mal. Quoi qu'il arrive, le développeur du moniteur n'a pas à se soucier de la manière dont le compilateur organise les choses à ce niveau. Il lui suffit de savoir qu'une fois toutes les sections critiques converties en procédures de moniteur, deux processus ne pourront jamais exécuter leurs sections critiques en même temps.

Malgré tout leur intérêt quant à l'exclusion mutuelle, les moniteurs ne suffisent pas. Il nous faut également faire en sorte que les processus se bloquent lorsqu'ils ne sont pas en mesure de poursuivre. Dans le contexte du producteur-consommateur, il est relativement aisés de placer tous les tests d'état du tampon (plein, vide, etc.) dans des procédures de moniteur, mais comment bloquer le producteur s'il trouve un tampon plein ?

La solution consiste à introduire des variables conditionnelles prenant en charge deux opérations : wait et signal. Lorsqu'une procédure de moniteur découvre qu'elle ne peut se poursuivre (le producteur apprend que le tampon est plein), elle effectue un wait sur une variable conditionnelle, que nous appellerons full. Cette action provoque le blocage du processus appelant. Elle permet à un autre processus, qui s'était vu auparavant interdire l'entrée dans le moniteur, d'y entrer maintenant.

Cet autre processus, par exemple le consommateur, peut réveiller son partenaire en sommeil en envoyant un signal sur la variable conditionnelle que son partenaire attend.

Pour éviter que deux processus actifs ne se trouvent dans le moniteur simultanément, nous avons besoin d'une règle indiquant ce qui se passe à l'issue d'un signal. Hoare a proposé de laisser s'exécuter le processus le plus récemment éveillé et de suspendre l'autre. Brinch Hansen a proposé une ruse : exiger du processus émettant le signal qu'il quitte le moniteur immédiatement. En d'autres termes, l'instruction signal pourrait apparaître comme la déclaration finale d'une procédure de moniteur. Nous retiendrons cette approche, car elle est conceptuellement plus simple, mais aussi d'une implémentation plus aisée. Si un signal est adressé à une variable conditionnelle attendue par plusieurs processus, un seul d'entre eux, déterminé par l'ordonnanceur système, sera réactif.

Il existe également une troisième solution, que ni Hoare, ni Brinch Hansen n'ont proposée. Elle consiste à laisser l'émetteur du signal continuer de s'exécuter et à autoriser le processus en attente à commencer son exécution dès que l'émetteur du signal a quitté le moniteur.

Les variables conditionnelles ne sont pas des compteurs. Elles n'accumulent pas les signaux pour une utilisation ultérieure, à la manière des sémaphores. Ainsi, si une variable conditionnelle est signalée alors que personne ne l'attend, le signal est perdu pour toujours. En d'autres termes, wait doit intervenir avant signal. Cette règle simplifie considérablement l'implémentation. Dans la pratique, ce n'est pas un problème dans la mesure où il est aisément de suivre l'état de chaque processus à l'aide de variables si nécessaire. Un processus qui, autrement, devrait émettre un signal, pourra ainsi constater que l'opération n'est pas nécessaire en consultant le contenu de ces variables.

La figure 2.27 donne l'articulation du code pour illustrer le problème du producteur-consommateur avec les moniteurs. Encore une fois, ce code est écrit dans un langage imaginaire appelé Sabir Pascal. L'avantage de ce langage imaginaire est qu'il est extrêmement simple et qu'il suit à la lettre le modèle développé par Hoare et Brinch Hansen.

Figure 2.27 Résoudre le problème du producteur-consommateur avec les moniteurs. Une seule procédure de moniteur est active à un moment donné. Le tampon possède N emplacements.

```

monitor ProducteurConsommateur
    condition full, empty;
    integer count;
procedure insert(item: integer);
begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
end;
function remove: integer;
begin
    if count = 0 then wait(empty);
    remove = remove_item;
end;

```

que wait n'est étant non état

Notre Sabir

tion existant

forme imagi-

langage on

rise le regar-

syncronizati-

a commencé

méthode su-

La figure 2.27

consommate-

quatre classi-

et c. Les de-

producteur

moniteur. Il

```

count := count - 1;
if count = N - 1 then signal(full)
end;
count := 0;
end monitor;
procedure producteur;
begin
while true do
begin
item = produce.item;
ProducteurConsommateur.insert(item)
end
end;
procedure consommateur;
begin
while true do
begin
item = ProducteurConsommateur.remove;
consume.item(item)
end
end;

```

Vous pensez peut-être que les opérations `wait` et `signal` ressemblent à `sleep` et `wakeup`, dont nous avons vu précédemment qu'elles intégraient des conditions de concurrence fatales. Et elles se ressemblent en effet. Mais à une différence essentielle près : `sleep` et `wakeup` échouaient. En effet, pendant qu'un processus tentait de dormir, l'autre essaierait de le réveiller. Or, cela ne peut pas se produire avec les moniteurs. L'exclusion mutuelle des procédures de moniteur garantit que, si le producteur se trouvant à l'intérieur d'une procédure de moniteur découvre que le tampon est plein, il sera capable de terminer l'opération `wait` sans avoir à se soucier de l'éventualité où l'ordonnanceur activerait le consommateur juste avant la fin de `wait`. Le consommateur ne sera même pas autorisé à entrer dans le moniteur tant que `wait` ne sera pas terminée et que le producteur n'aura pas été marqué comme étant non exécutable.

Notre Sabir Pascal est un langage imaginaire, mais certains langages de programmation existants prennent en charge les moniteurs, même si ce n'est pas toujours sous la forme imaginée par Hoare et Brinch Hansen. L'un de ces langages est Java. Java est un langage orienté objet qui prend en charge les threads au niveau utilisateur et qui autorise le regroupement de méthodes (procédures) en classes. En ajoutant le mot clé `synchronized` à une déclaration de méthode, Java garantit que, une fois qu'un thread a commencé à exécuter cette méthode, aucun autre thread ne peut lancer une autre méthode `synchronized` de cette classe.

La figure 2.28 montre comment apporter une solution au problème du producteur-consommateur avec des moniteurs, en langage Java cette fois. La solution repose sur quatre classes. La classe extérieure, `ProducerConsumer`, crée et démarre deux threads, `p` et `c`. Les deuxième et troisième classes, `producer` et `consumer`, contiennent le code du producteur et celui du consommateur. Enfin, la classe `our_monitor` représente le moniteur. Elle contient deux threads synchronisés qui servent à insérer des éléments

dans le tampon partagé, et à les récupérer. À la différence des exemples précédents, nous reprenons ici la totalité du code de `insert` et `remove`.

Les threads du producteur et du consommateur sont fonctionnellement identiques à leurs contreparties des exemples précédents. Le producteur inclut une boucle sans fin générant des données pour les placer dans le tampon commun. En outre, le consommateur possède une boucle sans fin qui récupère les données dans le tampon commun et les traite. La partie intéressante du programme repose sur la classe `our_monitor` : elle contient le tampon, les variables d'administration et deux méthodes synchronisées. Lorsque le producteur est actif dans `insert`, il sait que le consommateur ne peut pas l'être dans `remove`, ce qui lui permet d'actualiser les variables et le tampon sans craindre de conditions de concurrence. La variable `count` s'occupe du suivi du nombre d'éléments présents dans le tampon. Elle peut prendre n'importe quelle valeur comprise entre `0` et `N-1`. La variable `lo` représente l'index de l'emplacement du tampon dans lequel le prochain élément sera prélevé. De la même façon, `hi` représente l'index de l'emplacement du tampon où devra être placé le prochain élément. Il est possible que `lo = hi`, ce qui signifie que `0` ou `N-1` éléments se trouvent dans le tampon. C'est la variable `count` qui le détermine.

Figure 2.28 • Une solution Java au problème du producteur-consommateur.

```
public class ProducerConsumer {
    static final int N = 100; // constante donnant la taille du tampon
    static producer p = new producer(); // instancie un nouveau thread
    static consumer c = new consumer(); // instancie un nouveau thread
    static our_monitor mon = new our_monitor(); // instancie un nouveau moniteur
    public static void main(String args[]) {
        p.start(); // démarre le thread du producteur
        c.start(); // démarre le thread du consommateur
    }
    static class producer extends Thread {
        public void run() { // exécute la méthode contenant le code
            int item;
            while (true) { // boucle du producteur
                item = produce_item();
                mon.insert(item);
            }
        }
        private int produce_item() { ... } // véritable production
    }
    static class consumer extends Thread {
        public void run() { la méthode run contient le code du thread
            int item;
        }
    }
}
```

Les méthodes `produce_item()` et `insert(item)` sont réalisées dans les threads et wakened lorsque le produit est disponible. Elles peuvent être réalisées de manière suffisamment simple pour illustrer le principe.

En aucun cas, ce code ne devrait être utilisé dans un programme réel. Il est destiné à illustrer les mécanismes de synchronisation.

```
while (true) { // boucle du consommateur
    item = mon.remove();
    consume_item(item);
}
```

```
private void consume_item(int item) { ... } // véritable
                                            // consommation
```

```
}
```

```
static class our_monitor { // ceci est un moniteur
    private int buffer[] = new int[N];
    private int count = 0, lo = 0, hi = 0; // compteurs et index
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep(); // si le tampon est plein,
                                        // entre en sommeil
        buffer[hi] = val; // insère un élément dans le tampon
        hi = (hi + 1) % N; // emplacement où placer le prochain
        count = count + 1; // voici un élément de plus dans le tampon
        if (count == 1) notify(); // si le consommateur était en
                                // sommeil, le réveille
    }
}

public synchronized int remove() {
    int val;
    if (count == 0) go_to_sleep(); // si le tampon est vide,
                                    // entre en sommeil
    val = buffer[lo]; // prélève un élément dans le tampon
    lo = (lo + 1) % N; // emplacement où récupérer le prochain
                        // élément
    count = count - 1; // un des éléments du tampon
    if (count == N - 1) notify(); // si le producteur était en
                                // sommeil, le réveille
    return val;
}

private void go_to_sleep() { try{wait();}
                            // catch(InterruptedException exc) {}}
}
```

Les méthodes Java synchronisées sont différentes des moniteurs classiques pour une raison essentielle : Java ne dispose pas de variables conditionnelles. Le langage les remplace par deux procédures, `wait` et `notify`, qui sont les équivalents de `sleep` et `wakeup`. Cependant, quand on les utilise au sein de méthodes synchronisées, elles ne sont pas sujettes aux conditions de concurrence. En théorie, la méthode Java peut être interrompue, ce qui explique la présence du code qui l'entoure. Java a besoin d'un code de gestion des exceptions qui soit explicite. Pour notre objectif, il suffit d'imaginer que `go_to_sleep` est la méthode permettant de placer un processus en sommeil.

En automatisant l'exclusion mutuelle des sections critiques, les moniteurs rendent la programmation parallèle nettement moins sujette aux erreurs qu'avec les sémaphores. Mais ils présentent tout de même quelques inconvénients. Ce n'est pas pour rien que nous avons développé nos deux exemples de moniteurs en Sabir Pascal,

au lieu du C qui est le langage choisi pour les autres exemples de ce livre. Nous l'avons déjà dit, les moniteurs sont un concept de langage. Le compilateur doit les reconnaître et prendre en charge d'une manière ou d'une autre l'exclusion mutuelle. Le C, le Pascal et la plupart des autres langages n'ont pas de moniteurs. Il serait donc déraisonnable d'attendre de leurs compilateurs qu'ils mettent en œuvre des règles quelconques d'exclusion mutuelle. En fait, comment le compilateur peut-il savoir si une procédure se trouve ou non dans un moniteur ?

Ces mêmes langages ne disposent pas non plus de sémaphores, mais il est facile d'ajouter deux petites routines de code assembleur à la bibliothèque pour émettre les appels système up et down. Les compilateurs n'ont même pas besoin de savoir qu'elles existent. Bien entendu, les systèmes d'exploitation doivent être conscients de la présence des sémaphores. Mais, au moins, si vous travaillez sur un système d'exploitation reconnaissant les sémaphores, vous pouvez toujours développer les programmes utilisateur pour cela en C ou en C++. Avec les moniteurs, il faut un langage qui les prenne en charge.

Les moniteurs posent un autre problème, également valable en ce qui concerne les sémaphores. Ces deux notions ont été développées pour résoudre le problème de l'exclusion mutuelle sur un ou plusieurs processeurs ayant tous accès à une mémoire commune. En plaçant les sémaphores dans la mémoire partagée et en les protégeant au moyen d'instructions TS, il est possible d'éviter les concurrences. Lorsque l'on passe sur un système distribué composé de plusieurs processeurs, aucun disposant de sa propre mémoire et connecté en réseau local, ces primitives deviennent inapplicables. On peut en conclure que les sémaphores sont des outils de trop bas niveau et que les moniteurs ne sont pas praticables, excepté dans quelques langages. En outre, aucune des primitives ne permet de faire de l'échange d'informations entre ordinateurs. Il nous faut autre chose.

2.3.8 L'échange de messages

La solution est l'échange de messages (*message passing*). Cette méthode de communication interprocessus emploie deux primitives, send et receive. À l'instar des sémaphores, mais à la différence des moniteurs, ces primitives sont des appels système plutôt que des constructions de langage. En tant que tels, on les place facilement dans des procédures de bibliothèque, comme :

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/message.h>

void send(destination, &message);
void receive(source, &message);
```

et :

Le premier appel envoie un message vers une destination donnée, et le second reçoit un message d'une source donnée (ou de *n'importe quelle* destination ou source, si le récepteur n'y voit aucun inconvénient). En l'absence de message disponible, le récepteur peut se bloquer jusqu'à ce que celui-ci arrive. Il peut également retourner immédiatement un code d'erreur.

Concept

Les systèmes d'exploitation des différents fabricants utilisent des méthodes différentes pour établir un dialogue avec le réseau. Pour ce faire,

peuvent être utilisées la réception

un certain nombre de fois

rectement ou indirectement

transmettre à l'autre fois

Il est essentiel de faire

sage et la réception

en plaçant un

message portant

doublon et qui

manque de

réalisées sur

également gérées

sus spécifiée

cation est

peut-il savoir

non avec un

problème

qui situent sur

Le fait de co

d'effectuer

nombreux infor

exemple, on

pas la capaci

via les regist

comme

le système de

comparabil

teur commu

fois que le

message vi

sages reste

de mémoire

consomma

Conception des systèmes d'échange de messages

Les systèmes d'échange de messages posent de nombreux problèmes et représentent des défis de conception que l'on ne rencontre pas avec les sémaphores et les moniteurs, notamment si les processus de communication sont situés sur des ordinateurs différents reliés en réseau. Par exemple, les messages peuvent être perdus par le réseau. Pour se prémunir contre les pertes de messages, l'émetteur et le récepteur peuvent décider que, dès qu'il a reçu un message, le récepteur envoie un accusé de réception (*acknowledgement*). Si l'émetteur n'a pas reçu d'accusé de réception passé un certain délai, il retransmet le message. Mais que se passe-t-il si le message est correctement retransmis, mais que l'accusé de réception est perdu ? L'émetteur va transmettre à nouveau le message, ce qui fait que le récepteur va le recevoir deux fois. Il est essentiel que le récepteur puisse faire la distinction entre un nouveau message et la retransmission d'un ancien message. Généralement, on résout ce problème en plaçant un numéro dans chaque message original. Si le récepteur récupère un message portant le même numéro que le message précédent, il sait qu'il s'agit d'un douteur et qu'il peut l'ignorer. La réussite de la communication face au problème du manque de fiabilité de l'échange de messages est une partie importante des études réalisées sur les réseaux informatiques. Les systèmes fondés sur les messages doivent également gérer la manière dont sont nommés les processus, de façon qu'un processus spécifié dans les appels send ou receive ne génère pas d'ambiguïté. L'authentification est également un problème lié à l'échange de messages : comment le client peut-il savoir à coup sûr qu'il communique avec le véritable serveur de fichiers, et non avec un imposteur ? À l'autre extrémité du spectre, on rencontre également des problèmes de conception non négligeables lorsque l'émetteur et le récepteur se situent sur le même ordinateur. L'un de ces problèmes est celui de la performance. Le fait de copier des messages d'un processus vers un autre est toujours plus lent que d'effectuer une opération avec sémaphores ou d'entrer dans un moniteur. De nombreux informaticiens ont tenté d'améliorer l'efficacité de l'échange de messages. Par exemple, on a suggéré de limiter la taille des messages de sorte qu'ils ne dépassent pas la capacité des registres de l'ordinateur, puis de procéder à l'échange de messages via les registres.

Le problème du producteur-consommateur avec l'échange de messages

Voyons maintenant comment résoudre le problème du producteur-consommateur avec l'échange de messages, sans mémoire partagée. La figure 2.29 montre la solution. Nous supposons que les messages sont tous de la même taille et que les messages émis mais non encore reçus sont automatiquement mis en mémoire tampon par le système d'exploitation. Dans cette solution, on a un total de N messages, ce qui est comparable aux N emplacements d'un tampon de mémoire partagé. Le consommateur commence par émettre N messages vides en direction du producteur. Chaque fois que le producteur possède un élément à remettre au consommateur, il prend un message vide et renvoie un message plein. De cette manière, le nombre total de messages reste constant au sein du système, ce qui permet de les stocker dans un volume de mémoire déterminé par avance. Si le producteur travaille plus rapidement que le consommateur, tous les messages finissent par être pleins ; le producteur est bloqué,

en attente d'un message vide en retour. Si le consommateur est le plus rapide, la situation inverse se produit : tous les messages sont vides ; le consommateur est bloqué, dans l'attente d'un message plein.

Figure 2.29 • Le problème du producteur-consommateur avec N messages.

```
#define N 100                                /* nombre d'emplacements dans le
                                              tampon */

void producer(void)
{
    int item;
    message m;                                /* tampon des messages */
    while (TRUE) {
        item = produce_item(); /* génère quelque chose à placer dans
                                  le tampon */
        receive(consumer, &m); /* attend l'arrivée d'un message
                                  vide */
        build_message(&m, item); /* construit un message à envoyer */
        send(consumer, &m); /* envoie l'élément au consommateur */
    }
}

void consumer(void)
{
    int item, i;
    message m;
    for (i = 0; i < N; i++) send(producer, &m); /* envoie N messages
                                                vides */
    while (TRUE) {
        receive(producer, &m); /* récupère un message contenant
                                  un élément */
        item = extract_item(&m); /* extrait l'élément du message */
        send(producer, &m); /* retourne une réponse vide */
        consume_item(item); /* utilise l'élément */
    }
}
```

Il existe de nombreuses variantes d'échange de messages. Pour commencer, voyons comment sont adressés les messages. Une méthode consiste à assigner à chaque processus une adresse unique et à faire en sorte que les messages soient adressés aux processus. Une autre solution consiste à inventer une nouvelle structure de données que l'on appellera *mailbox* (boîte aux lettres). Celle-ci permet de placer dans le tampon un certain nombre de messages, généralement spécifié lors de la création de la *mailbox*. Lorsque l'on utilise les *mailbox*, les paramètres d'adressage des appels *send* et *receive* déclarent des *mailbox*, et non des processus. Quand un processus tente d'émettre en direction d'une *mailbox* pleine, il est suspendu jusqu'à ce qu'un message disparaîsse de la *mailbox*, laissant de la place pour un nouveau message.

Dans le contexte du producteur-consommateur, tant le producteur que le consommateur vont créer des *mailbox* assez conséquentes pour contenir N messages. Le pro-

ducteur va envoyer des messages contenant des données vers la mailbox du consommateur, et ce dernier va envoyer des messages vides en direction de la mailbox du producteur. Avec les mailbox, le mécanisme de mise en mémoire tampon est clair : la mailbox de destination détient les messages qui ont été envoyés au processus récepteur, mais qui n'ont pas encore été acceptés.

À l'autre extrême, on peut éliminer toute opération de mise en mémoire tampon. Avec cette approche, si le send est effectué avant le receive, le processus émetteur est bloqué jusqu'à ce que receive s'exécute, moment où le message pourra être copié directement depuis l'émetteur vers le récepteur, sans tampon intermédiaire. De même, si le receive intervient d'abord, le récepteur est bloqué jusqu'à l'exécution d'un send. C'est ce que l'on appelle souvent la stratégie du rendez-vous. Elle est plus facile à mettre en œuvre qu'un schéma impliquant des tampons de messages, mais offre moins de souplesse dans la mesure où l'émetteur et le récepteur se bloquent mutuellement.

L'échange de messages est souvent exploité dans les systèmes de programmation en parallèle. Le MPI (*Message-Passing Interface*) en est une réalisation bien connue, largement employée en informatique scientifique.

2.3.9 Les barrières

Notre dernier mécanisme de synchronisation est destiné aux groupes de processus plutôt qu'aux situations de type producteur-consommateur impliquant deux processus. Certaines applications sont décomposées en phases. Elles ont pour règle qu'aucun processus ne peut entrer dans la phase suivante tant que tous les processus ne sont pas prêts à y entrer. Ce comportement peut se produire si l'on place une barrière à la fin de chaque phase. Lorsqu'un processus atteint la barrière, il est bloqué jusqu'à ce que tous les processus l'atteignent également. Le fonctionnement des barrières est illustré à la figure 2.30.

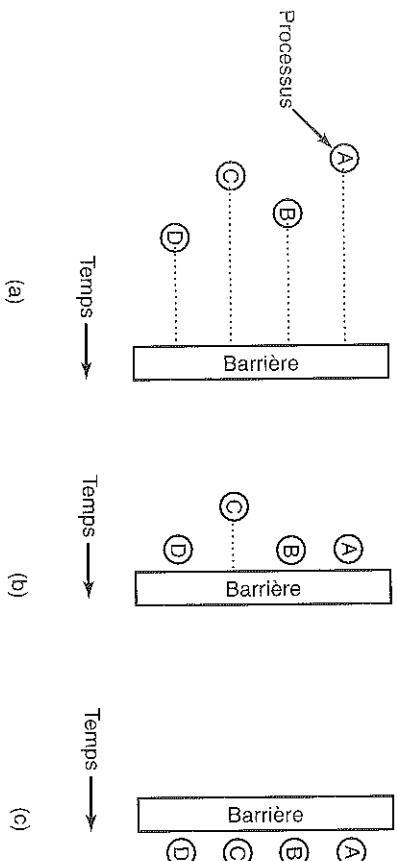


Figure 2.30 • Utilisation d'une barrière. (a) Processus approchant de la barrière. (b) Une fois le dernier processus arrivé, tous les processus peuvent passer.

À la figure 2.30(a), nous voyons quatre processus approcher d'une barrière. Cela signifie qu'ils sont en cours de traitement et qu'ils n'ont pas encore atteint la fin de la phase en cours. Au bout d'un moment, le premier processus termine son traitement pour ce qui est de la première phase. Il est suspendu. Un peu plus tard, un deuxième, puis un troisième processus terminent la première phase et exécutent également leur primitive barrier. La figure 2.30(b) montre cette situation. Enfin, lorsque le dernier processus atteint la barrière, tous les processus sont libérés, comme le montre la figure 2.30(c).

Voici un exemple de situation où les barrières sont intéressantes : dans un cours de physique, on propose aux étudiants de résoudre un problème. On leur donne une matrice contenant un certain nombre de valeurs initiales. Celles-ci représentent les températures en différents points d'une feuille de métal. On leur demande de calculer combien de temps il faut pour que l'effet d'une flamme placée dans un angle se propage à l'ensemble de la feuille. En prenant les valeurs initiales comme point de départ, une transformation est appliquée à la matrice pour récupérer une deuxième version de cette matrice en appliquant les lois de la thermodynamique. Il s'agit de voir quelles sont les températures à l'issue d'un délai T . On répète le processus plusieurs fois, et on note les températures aux points échantillons, chronologiquement, à mesure que la feuille se réchauffe. L'algorithme produit alors une série de matrices étalées dans le temps.

Imaginons maintenant que notre matrice soit très volumineuse (1 million par 1 million), ce qui implique l'intervention de processus parallèles (éventuellement sur un ordinateur multiprocesseur) afin d'accélérer les calculs. Différents processus traînent sur diverses portions de la matrice, calculant les nouveaux éléments de matrice à partir des anciens. Cependant, aucun processus ne peut démarrer sur l'itération $n + 1$ tant que l'itération n n'est pas terminée, autrement dit tant que tous les processus n'ont pas terminé leurs calculs. Pour atteindre un tel objectif, il faut programmer chaque processus de sorte qu'il exécute une opération barrière après avoir fini avec l'itération en cours. Une fois que tous les processus ont terminé, la nouvelle matrice (l'entrée de la prochaine itération) est calculée, et tous les processus sont libérés simultanément pour démarrer l'itération suivante.

2.4 Les problèmes classiques

La documentation sur les systèmes d'exploitation fourmille de problèmes intéressants largement débattus et analysés au moyen d'un éventail étendu de méthodes de synchronisation. Dans les sections suivantes, nous aborderons trois de ces problèmes bien connus.

2.4.1 Le dîner des philosophes

En 1965, Dijkstra a posé et résolu un problème de synchronisation qu'il a appelé le problème du dîner des philosophes. Depuis lors, toutes les personnes ayant mis au

point une nouvelle primitive de synchronisation se sont senties obligées de démontrer son grand intérêt en illustrant à quel point elle résolvait élégamment le problème du dîner des philosophes. Le problème peut être posé très simplement. Cinq philosophes sont assis autour d'une table ronde. Chacun a devant lui une assiette de spaghetti. Mais ceux-ci sont si glissants qu'il faut deux fourchettes pour les manger. Entre deux assiettes se trouve une fourchette. La présentation de la table apparaît à la figure 2.31.

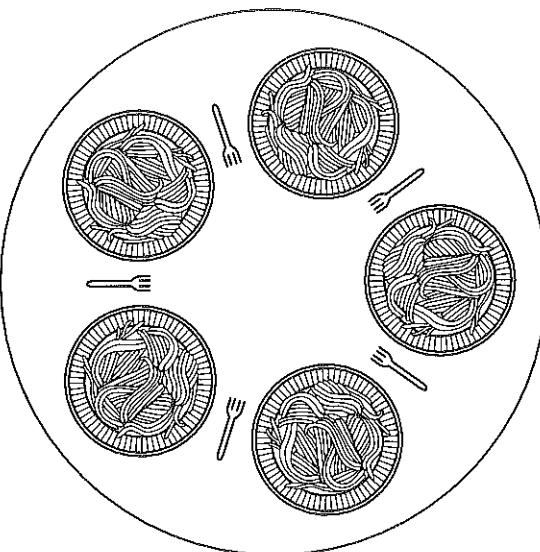


Figure 2.31 • L'heure du repas, au département de philosophie.

Au cours de sa vie, chaque philosophe connaît deux types de périodes : celles où il mange et celles où il pense. C'est assez abstrait, même pour des philosophes, mais les autres activités n'entrent pas en ligne de compte pour la résolution du problème. Quand un philosophe a faim, il tente de s'emparer des fourchettes droite et gauche, sans ordre défini. S'il y parvient, il mange pendant un moment, puis pose les fourchettes et se met à penser. La question est la suivante : pouvez-vous écrire un programme pour que chaque philosophe puisse exercer ces deux activités sans jamais se retrouver bloqué ? On peut souligner que l'exigence des deux fourchettes est un peu artificielle, et que l'on devrait peut-être penser le problème en termes de cuisine chinoise, en remplaçant les fourchettes par des baguettes et les spaghetti par du riz. Quoi qu'il en soit, la figure 2.3.2 montre une solution évidente. La procédure `prendre_fourchette` attend que la fourchette spécifiée soit disponible et s'en saisit. Malheureusement, pour aussi évidente qu'elle soit, cette solution ne fonctionne pas. Supposons que les cinq philosophes prennent leur fourchette gauche en même temps. Aucun ne pourra prendre sa fourchette droite, et il y aura un inter-blocage.

Il est possible de modifier le programme de la façon suivante : après qu'un philosophe a pris sa fourchette gauche, le code détermine si la fourchette droite est disponible. Si ce n'est pas le cas, le philosophe dépose la fourchette gauche, attend pendant un certain temps, puis recommence le processus. Or, cette proposition échoue également, mais pour une raison différente. Avec un peu de malchance, tous les philosophes peuvent reprendre l'algorithme simultanément : chacun prend sa fourchette gauche, puis, constatant que celle de droite n'est pas disponible, la repose et attend, et ainsi de suite. Une telle situation, au cours de laquelle tous les programmes continuent de s'exécuter indéfiniment mais ne progressent jamais, se nomme **privatation de ressources** (ou famine).

Figure 2.32 • Une non-solution au problème du dîner des philosophes.

```
#define N 5
void philosopher(int i)
{
    while (TRUE) {
        penser();
        prendre_fourchette(i);
        prendre_fourchette((i+1) % N); /* prend sa fourchette droite.
                                         ↴ % est le modulo */
        manger();
        poser_fourchette(i);
        poser_fourchette((i+1) % N);
        /* repose la fourchette gauche
         ↴ Sur la table */
        /* repose la fourchette droite
         ↴ sur la table */
    }
}
```

Vous pensez peut-être que si les philosophes se contentaient d'attendre un instant aléatoirement déterminé au lieu d'un même intervalle pour prendre leur fourchette droite, les chances de blocage de la situation seraient très faibles, même sur une période d'une heure. Cette observation est vraie, et dans presque toutes les applications, la technique des tentatives successives fonctionne. Par exemple, dans le très répandu réseau local Ethernet, lorsque deux ordinateurs envoient un paquet en même temps, chacun attend pendant un délai aléatoire et réessaie. Dans la pratique, cette solution fonctionne bel et bien. Cependant, dans certains cas de figures, il est préférable de trouver une solution qui fonctionne systématiquement et qui ne risque pas d'échouer du fait d'une série de nombres aléatoires non maîtrisables. (Pensez au contrôle de la sécurité dans une centrale nucléaire.)

La figure 2.32 illustre une amélioration qui ne produit ni interblocage, ni privation de ressources. Elle consiste à protéger les cinq instructions suivant l'appel à penser par un sémaaphore binaire. Avant de s'emparer d'une fourchette, un philosophe donne émet un down sur mutex. Après avoir remplacé ses fourchettes, il effectue un up sur mutex. Du point de vue théorique, cette solution est appropriée. Mais du point de vue pratique, elle présente un bogue de performance : un seul philosophe peut manger à

la fois. Étant donné que les philosophes doivent pouvoir manger, la solution devient maximal pour l'état, pour être faim et tenté l'état « mange » philosophes i à 2, GAUCHE est

Figure 2.33 • U

```
#define N 5
#define G 1
#define D 0
#define P 2
#define R 3
#define M 4
#define I 5
#define S 6
#define F 7
#define B 8
#define E 9
#define T 10
#define C 11
#define L 12
#define H 13
#define O 14
#define U 15
#define V 16
#define W 17
#define X 18
#define Y 19
#define Z 20
#define A 21
#define B 22
#define C 23
#define D 24
#define E 25
#define F 26
#define G 27
#define H 28
#define I 29
#define J 30
#define K 31
#define L 32
#define M 33
#define N 34
#define O 35
#define P 36
#define Q 37
#define R 38
#define S 39
#define T 40
#define U 41
#define V 42
#define W 43
#define X 44
#define Y 45
#define Z 46
#define A 47
#define B 48
#define C 49
#define D 50
#define E 51
#define F 52
#define G 53
#define H 54
#define I 55
#define J 56
#define K 57
#define L 58
#define M 59
#define N 60
#define O 61
#define P 62
#define Q 63
#define R 64
#define S 65
#define T 66
#define U 67
#define V 68
#define W 69
#define X 70
#define Y 71
#define Z 72
#define A 73
#define B 74
#define C 75
#define D 76
#define E 77
#define F 78
#define G 79
#define H 80
#define I 81
#define J 82
#define K 83
#define L 84
#define M 85
#define N 86
#define O 87
#define P 88
#define Q 89
#define R 90
#define S 91
#define T 92
#define U 93
#define V 94
#define W 95
#define X 96
#define Y 97
#define Z 98
#define A 99
#define B 100
#define C 101
#define D 102
#define E 103
#define F 104
#define G 105
#define H 106
#define I 107
#define J 108
#define K 109
#define L 110
#define M 111
#define N 112
#define O 113
#define P 114
#define Q 115
#define R 116
#define S 117
#define T 118
#define U 119
#define V 120
#define W 121
#define X 122
#define Y 123
#define Z 124
#define A 125
#define B 126
#define C 127
#define D 128
#define E 129
#define F 130
#define G 131
#define H 132
#define I 133
#define J 134
#define K 135
#define L 136
#define M 137
#define N 138
#define O 139
#define P 140
#define Q 141
#define R 142
#define S 143
#define T 144
#define U 145
#define V 146
#define W 147
#define X 148
#define Y 149
#define Z 150
#define A 151
#define B 152
#define C 153
#define D 154
#define E 155
#define F 156
#define G 157
#define H 158
#define I 159
#define J 160
#define K 161
#define L 162
#define M 163
#define N 164
#define O 165
#define P 166
#define Q 167
#define R 168
#define S 169
#define T 170
#define U 171
#define V 172
#define W 173
#define X 174
#define Y 175
#define Z 176
#define A 177
#define B 178
#define C 179
#define D 180
#define E 181
#define F 182
#define G 183
#define H 184
#define I 185
#define J 186
#define K 187
#define L 188
#define M 189
#define N 190
#define O 191
#define P 192
#define Q 193
#define R 194
#define S 195
#define T 196
#define U 197
#define V 198
#define W 199
#define X 200
#define Y 201
#define Z 202
#define A 203
#define B 204
#define C 205
#define D 206
#define E 207
#define F 208
#define G 209
#define H 210
#define I 211
#define J 212
#define K 213
#define L 214
#define M 215
#define N 216
#define O 217
#define P 218
#define Q 219
#define R 220
#define S 221
#define T 222
#define U 223
#define V 224
#define W 225
#define X 226
#define Y 227
#define Z 228
#define A 229
#define B 230
#define C 231
#define D 232
#define E 233
#define F 234
#define G 235
#define H 236
#define I 237
#define J 238
#define K 239
#define L 240
#define M 241
#define N 242
#define O 243
#define P 244
#define Q 245
#define R 246
#define S 247
#define T 248
#define U 249
#define V 250
#define W 251
#define X 252
#define Y 253
#define Z 254
#define A 255
#define B 256
#define C 257
#define D 258
#define E 259
#define F 260
#define G 261
#define H 262
#define I 263
#define J 264
#define K 265
#define L 266
#define M 267
#define N 268
#define O 269
#define P 270
#define Q 271
#define R 272
#define S 273
#define T 274
#define U 275
#define V 276
#define W 277
#define X 278
#define Y 279
#define Z 280
#define A 281
#define B 282
#define C 283
#define D 284
#define E 285
#define F 286
#define G 287
#define H 288
#define I 289
#define J 290
#define K 291
#define L 292
#define M 293
#define N 294
#define O 295
#define P 296
#define Q 297
#define R 298
#define S 299
#define T 300
#define U 301
#define V 302
#define W 303
#define X 304
#define Y 305
#define Z 306
#define A 307
#define B 308
#define C 309
#define D 310
#define E 311
#define F 312
#define G 313
#define H 314
#define I 315
#define J 316
#define K 317
#define L 318
#define M 319
#define N 320
#define O 321
#define P 322
#define Q 323
#define R 324
#define S 325
#define T 326
#define U 327
#define V 328
#define W 329
#define X 330
#define Y 331
#define Z 332
#define A 333
#define B 334
#define C 335
#define D 336
#define E 337
#define F 338
#define G 339
#define H 340
#define I 341
#define J 342
#define K 343
#define L 344
#define M 345
#define N 346
#define O 347
#define P 348
#define Q 349
#define R 350
#define S 351
#define T 352
#define U 353
#define V 354
#define W 355
#define X 356
#define Y 357
#define Z 358
#define A 359
#define B 360
#define C 361
#define D 362
#define E 363
#define F 364
#define G 365
#define H 366
#define I 367
#define J 368
#define K 369
#define L 370
#define M 371
#define N 372
#define O 373
#define P 374
#define Q 375
#define R 376
#define S 377
#define T 378
#define U 379
#define V 380
#define W 381
#define X 382
#define Y 383
#define Z 384
#define A 385
#define B 386
#define C 387
#define D 388
#define E 389
#define F 390
#define G 391
#define H 392
#define I 393
#define J 394
#define K 395
#define L 396
#define M 397
#define N 398
#define O 399
#define P 400
#define Q 401
#define R 402
#define S 403
#define T 404
#define U 405
#define V 406
#define W 407
#define X 408
#define Y 409
#define Z 410
#define A 411
#define B 412
#define C 413
#define D 414
#define E 415
#define F 416
#define G 417
#define H 418
#define I 419
#define J 420
#define K 421
#define L 422
#define M 423
#define N 424
#define O 425
#define P 426
#define Q 427
#define R 428
#define S 429
#define T 430
#define U 431
#define V 432
#define W 433
#define X 434
#define Y 435
#define Z 436
#define A 437
#define B 438
#define C 439
#define D 440
#define E 441
#define F 442
#define G 443
#define H 444
#define I 445
#define J 446
#define K 447
#define L 448
#define M 449
#define N 450
#define O 451
#define P 452
#define Q 453
#define R 454
#define S 455
#define T 456
#define U 457
#define V 458
#define W 459
#define X 460
#define Y 461
#define Z 462
#define A 463
#define B 464
#define C 465
#define D 466
#define E 467
#define F 468
#define G 469
#define H 470
#define I 471
#define J 472
#define K 473
#define L 474
#define M 475
#define N 476
#define O 477
#define P 478
#define Q 479
#define R 480
#define S 481
#define T 482
#define U 483
#define V 484
#define W 485
#define X 486
#define Y 487
#define Z 488
#define A 489
#define B 490
#define C 491
#define D 492
#define E 493
#define F 494
#define G 495
#define H 496
#define I 497
#define J 498
#define K 499
#define L 500
#define M 501
#define N 502
#define O 503
#define P 504
#define Q 505
#define R 506
#define S 507
#define T 508
#define U 509
#define V 510
#define W 511
#define X 512
#define Y 513
#define Z 514
#define A 515
#define B 516
#define C 517
#define D 518
#define E 519
#define F 520
#define G 521
#define H 522
#define I 523
#define J 524
#define K 525
#define L 526
#define M 527
#define N 528
#define O 529
#define P 530
#define Q 531
#define R 532
#define S 533
#define T 534
#define U 535
#define V 536
#define W 537
#define X 538
#define Y 539
#define Z 540
#define A 541
#define B 542
#define C 543
#define D 544
#define E 545
#define F 546
#define G 547
#define H 548
#define I 549
#define J 550
#define K 551
#define L 552
#define M 553
#define N 554
#define O 555
#define P 556
#define Q 557
#define R 558
#define S 559
#define T 560
#define U 561
#define V 562
#define W 563
#define X 564
#define Y 565
#define Z 566
#define A 567
#define B 568
#define C 569
#define D 570
#define E 571
#define F 572
#define G 573
#define H 574
#define I 575
#define J 576
#define K 577
#define L 578
#define M 579
#define N 580
#define O 581
#define P 582
#define Q 583
#define R 584
#define S 585
#define T 586
#define U 587
#define V 588
#define W 589
#define X 590
#define Y 591
#define Z 592
#define A 593
#define B 594
#define C 595
#define D 596
#define E 597
#define F 598
#define G 599
#define H 600
#define I 601
#define J 602
#define K 603
#define L 604
#define M 605
#define N 606
#define O 607
#define P 608
#define Q 609
#define R 610
#define S 611
#define T 612
#define U 613
#define V 614
#define W 615
#define X 616
#define Y 617
#define Z 618
#define A 619
#define B 620
#define C 621
#define D 622
#define E 623
#define F 624
#define G 625
#define H 626
#define I 627
#define J 628
#define K 629
#define L 630
#define M 631
#define N 632
#define O 633
#define P 634
#define Q 635
#define R 636
#define S 637
#define T 638
#define U 639
#define V 640
#define W 641
#define X 642
#define Y 643
#define Z 644
#define A 645
#define B 646
#define C 647
#define D 648
#define E 649
#define F 650
#define G 651
#define H 652
#define I 653
#define J 654
#define K 655
#define L 656
#define M 657
#define N 658
#define O 659
#define P 660
#define Q 661
#define R 662
#define S 663
#define T 664
#define U 665
#define V 666
#define W 667
#define X 668
#define Y 669
#define Z 670
#define A 671
#define B 672
#define C 673
#define D 674
#define E 675
#define F 676
#define G 677
#define H 678
#define I 679
#define J 680
#define K 681
#define L 682
#define M 683
#define N 684
#define O 685
#define P 686
#define Q 687
#define R 688
#define S 689
#define T 690
#define U 691
#define V 692
#define W 693
#define X 694
#define Y 695
#define Z 696
#define A 697
#define B 698
#define C 699
#define D 700
#define E 701
#define F 702
#define G 703
#define H 704
#define I 705
#define J 706
#define K 707
#define L 708
#define M 709
#define N 710
#define O 711
#define P 712
#define Q 713
#define R 714
#define S 715
#define T 716
#define U 717
#define V 718
#define W 719
#define X 720
#define Y 721
#define Z 722
#define A 723
#define B 724
#define C 725
#define D 726
#define E 727
#define F 728
#define G 729
#define H 730
#define I 731
#define J 732
#define K 733
#define L 734
#define M 735
#define N 736
#define O 737
#define P 738
#define Q 739
#define R 740
#define S 741
#define T 742
#define U 743
#define V 744
#define W 745
#define X 746
#define Y 747
#define Z 748
#define A 749
#define B 750
#define C 751
#define D 752
#define E 753
#define F 754
#define G 755
#define H 756
#define I 757
#define J 758
#define K 759
#define L 760
#define M 761
#define N 762
#define O 763
#define P 764
#define Q 765
#define R 766
#define S 767
#define T 768
#define U 769
#define V 770
#define W 771
#define X 772
#define Y 773
#define Z 774
#define A 775
#define B 776
#define C 777
#define D 778
#define E 779
#define F 780
#define G 781
#define H 782
#define I 783
#define J 784
#define K 785
#define L 786
#define M 787
#define N 788
#define O 789
#define P 790
#define Q 791
#define R 792
#define S 793
#define T 794
#define U 795
#define V 796
#define W 797
#define X 798
#define Y 799
#define Z 800
#define A 801
#define B 802
#define C 803
#define D 804
#define E 805
#define F 806
#define G 807
#define H 808
#define I 809
#define J 810
#define K 811
#define L 812
#define M 813
#define N 814
#define O 815
#define P 816
#define Q 817
#define R 818
#define S 819
#define T 820
#define U 821
#define V 822
#define W 823
#define X 824
#define Y 825
#define Z 826
#define A 827
#define B 828
#define C 829
#define D 830
#define E 831
#define F 832
#define G 833
#define H 834
#define I 835
#define J 836
#define K 837
#define L 838
#define M 839
#define N 840
#define O 841
#define P 842
#define Q 843
#define R 844
#define S 845
#define T 846
#define U 847
#define V 848
#define W 849
#define X 850
#define Y 851
#define Z 852
#define A 853
#define B 854
#define C 855
#define D 856
#define E 857
#define F 858
#define G 859
#define H 860
#define I 861
#define J 862
#define K 863
#define L 864
#define M 865
#define N 866
#define O 867
#define P 868
#define Q 869
#define R 870
#define S 871
#define T 872
#define U 873
#define V 874
#define W 875
#define X 876
#define Y 877
#define Z 878
#define A 879
#define B 880
#define C 881
#define D 882
#define E 883
#define F 884
#define G 885
#define H 886
#define I 887
#define J 888
#define K 889
#define L 890
#define M 891
#define N 892
#define O 893
#define P 894
#define Q 895
#define R 896
#define S 897
#define T 898
#define U 899
#define V 900
#define W 901
#define X 902
#define Y 903
#define Z 904
#define A 905
#define B 906
#define C 907
#define D 908
#define E 909
#define F 910
#define G 911
#define H 912
#define I 913
#define J 914
#define K 915
#define L 916
#define M 917
#define N 918
#define O 919
#define P 920
#define Q 921
#define R 922
#define S 923
#define T 924
#define U 925
#define V 926
#define W 927
#define X 928
#define Y 929
#define Z 930
#define A 931
#define B 932
#define C 933
#define D 934
#define E 935
#define F 936
#define G 937
#define H 938
#define I 939
#define J 940
#define K 941
#define L 942
#define M 943
#define N 944
#define O 945
#define P 946
#define Q 947
#define R 948
#define S 949
#define T 950
#define U 951
#define V 952
#define W 953
#define X 954
#define Y 955
#define Z 956
#define A 957
#define B 958
#define C 959
#define D 960
#define E 961
#define F 962
#define G 963
#define H 964
#define I 965
#define J 966
#define K 967
#define L 968
#define M 969
#define N 970
#define O 971
#define P 972
#define Q 973
#define R 974
#define S 975
#define T 976
#define U 977
#define V 978
#define W 979
#define X 980
#define Y 981
#define Z 982
#define A 983
#define B 984
#define C 985
#define D 986
#define E 987
#define F 988
#define G 989
#define H 990
#define I 991
#define J 992
#define K 993
#define L 994
#define M 995
#define N 996
#define O 997
#define P 998
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999
#define J 999
#define K 999
#define L 999
#define M 999
#define N 999
#define O 999
#define P 999
#define Q 999
#define R 999
#define S 999
#define T 999
#define U 999
#define V 999
#define W 999
#define X 999
#define Y 999
#define Z 999
#define A 999
#define B 999
#define C 999
#define D 999
#define E 999
#define F 999
#define G 999
#define H 999
#define I 999

```

la fois. Étant donné que nous avons cinq fourchettes sur la table, deux philosophes doivent pouvoir manger en même temps.

La solution de la figure 2.33 ne produit pas d'interblocage et autorise le parallélisme maximal pour un nombre arbitraire de philosophes. Elle utilise un tableau, appelé state, pour effectuer le suivi de « l'état » du philosophe (il mange, il pense, ou il a faim et tente de s'emparer des fourchettes). Un philosophe donné ne peut passer à l'état « manger » que si aucun de ses voisins ne mange à ce moment-là. Les voisins du philosophe i sont définis par les macros GAUCHE et DROITE. En d'autres termes, si i est à 2, GAUCHE est à 1 et DROITE est à 3.

Figure 2.33 • Une solution au problème du dîner des philosophes.

```
#define N 5
#define GAUCHE ((i+N-1)%N
#define DROITE ((i+1)%N
#define PENSE 0
#define FAIM 1

#define MANGE 2
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];
void philosophie(int i)
{
    while (TRUE) {
        penser();
        prendre_fourchettes(i);
        manger();
        poser_fourchettes(i);
        /* la table */
    }
}

void prendre_fourchettes(int i)/* i: numéro du philosophe, de 0 à N -1*/
{
    down(&mutex);
    state[i] = FAIM;
    test(i);
    up(&mutex);
    down(&s[i]);
    /* entre en section critique */
    /* enregistre le fait que le philosophe
     * a faim */
    /* tente de prendre deux fourchettes */
    /* quitte la section critique */
    /* bloque s'il n'a pas pu prendre
     * les fourchettes */
}

void poser_fourchettes(i)/* i: numéro du philosophe, de 0 à N -1*/
{
    down(&mutex);
    /* entre en section critique */
    /* le philosophe a fini de manger */
    /* voyons si son voisin de gauche
```

```

    test(DROITE);
        /* voyons si son voisin de droite
         * peut manger */
    up(&mutex);
    /* quitte la section critique */

}

void test(i)
{
    if (state[i] == FAIM && state[GAUCHE] != MANGE && state[DROITE]
        != ! = MANGE) {
        state[i] = MANGE;
        up(&s[i]);
    }
}

```

/* i: numéro du philosophe, de 0 à N -1 */

Le programme utilise un tableau de sémaphores, un par individu, de façon que les philosophes soient bloqués si les fourchettes sont prises. Remarquez que chaque processus exécute la procédure `philosophe` en tant que code principal, mais que les autres procédures, à savoir `prendre_fourchettes`, `poser_fourchettes` et `test`, sont des procédures ordinaires et non des processus indépendants.

2.4.2 Le problème des lecteurs et des rédacteurs

Le problème du dîner des philosophes est intéressant pour modéliser des processus qui entrent en concurrence pour un accès exclusif à un nombre limité de ressources, comme c'est le cas des périphériques d'E/S. Un autre problème célèbre est celui des lecteurs et des rédacteurs qui permettent de modéliser les accès aux bases de données. Imaginons, par exemple, un système de réservation de billets d'avion, où de nombreux processus entrent en concurrence pour y effectuer des lectures et des écritures. Il serait acceptable que plusieurs processus puissent lire la base de données en même temps. Mais si un processus est en train d'actualiser la base (écriture), aucun autre processus ne doit pouvoir y accéder, pas même en lecture. Comment programmer ces lecteurs et ces rédacteurs ? Une solution apparaît à la figure 2.34. Le premier lecteur qui accède à la base de données fait un down sur le sémaphore `cb`. Les rédacteurs qui suivent incrémentent simplement un compteur appelé `rc`. À mesure que les rédacteurs quittent la base, ils décrémentent le compteur. Le dernier sorti fait un up sur le sémaphore, autorisant ainsi un rédacteur bloqué, s'il y en a un, à entrer.

La solution ici présentée implique une décision subtile qu'il vaut la peine de commenter. Supposons que pendant qu'un lecteur utilise la base, un autre lecteur se connecte. Étant donné que deux lecteurs peuvent consulter la base en même temps, le deuxième lecteur est admis. Un troisième lecteur, et d'autres le seront également s'ils se présentent.

C'est alors qu'un rédacteur arrive. Il ne peut pas être admis à entrer dans la base, car les rédacteurs doivent disposer d'une exclusivité d'accès. Le rédacteur est donc suspendu. Ensuite, d'autres lecteurs arrivent. Tant qu'un lecteur au moins est actif, les lecteurs suivants le sont aussi. Une telle stratégie fait que tant qu'il y a une succession régulière de lecteurs, ceux-ci entrent dès qu'ils arrivent. Le rédacteur sera suspendu

jusqu'à ce que plus aucun lecteur ne soit présent. Si un nouveau lecteur arrive, disons, toutes les 2 secondes, et que chacun mette 5 secondes à effectuer son travail, le rédacteur ne pourra jamais entrer.

Pour éviter cela, le programme peut être développé de manière légèrement différente : lorsqu'un lecteur se présente alors qu'un rédacteur est en attente, le lecteur est maintenu derrière le rédacteur au lieu d'être admis immédiatement. De cette façon, le rédacteur doit attendre que les lecteurs actifs aient terminé, mais pas ceux qui viennent après lui. L'inconvénient de cette solution est qu'elle réduit les possibilités de concurrence, et par là même les performances. Courtois et al. ont proposé une autre solution qui donne la priorité aux rédacteurs.

Figure 2.34 • Une solution au problème des lecteurs et des rédacteurs.

```

typedef int semaphore;           /* utilisez votre imagination */
semaphore mutex = 1;            /* contrôle l'accès à rc */
semaphore db = 1;               /* contrôle l'accès à la base de données */
/* # de processus lecteurs ou voulant lire */
int rc = 0;

void reader(void)
{
    while (TRUE) {
        down(&mutex);           /* boucle sans fin */
        rc = rc + 1;             /* obtient l'accès exclusif à rc */
        if (rc == 1) down(&db);   /* et un lecteur de plus */
        /* si c'est le premier lecteur... */
        up(&mutex);              /* libère l'accès exclusif à rc */
        read_data_base();         /* accède aux données */
        down(&mutex);             /* récupère l'accès exclusif à rc */
        rc = rc - 1;              /* et un lecteur de moins */
        if (rc == 0) up(&db);     /* si c'est le dernier lecteur... */
        up(&mutex);               /* libère l'accès exclusif à rc */
        use_data_read();          /* section non critique */
    }
}

void writer(void)
{
    while (TRUE) {
        /* boucle sans fin */
        think_up_data();         /* section non critique */
        down(&db);                /* récupère l'accès exclusif */
        write_data_base();         /* actualise les données */
        up(&db);                  /* libère l'accès exclusif */
    }
}

```

2.4.3 Le problème du coiffeur endormi

Un autre problème de communication interprocessus classique a pour cadre un salon de coiffure. On y trouve un coiffeur, un fauteuil, et n chaises pour les clients qui attendent. S'il n'y a pas de client, le coiffeur s'assoit dans le fauteuil et s'endort, comme le montre la figure 2.35. Lorsqu'un client arrive, il doit réveiller le coiffeur. Si d'autres clients arrivent alors que le coiffeur est en train de travailler, ils s'assoient (s'il y a des

chaises vides) ou repartent (si les chaises sont toutes occupées). Le problème consiste à programmer le coiffeur et les clients de façon à ne produire aucune condition de concurrence. Il nous rappelle certaines situations de mise en file d'attente, comme dans un *helpdesk* comptant plusieurs techniciens intervenants, avec un système d'attente d'appel informatisé pour limiter le nombre d'appels entrants.

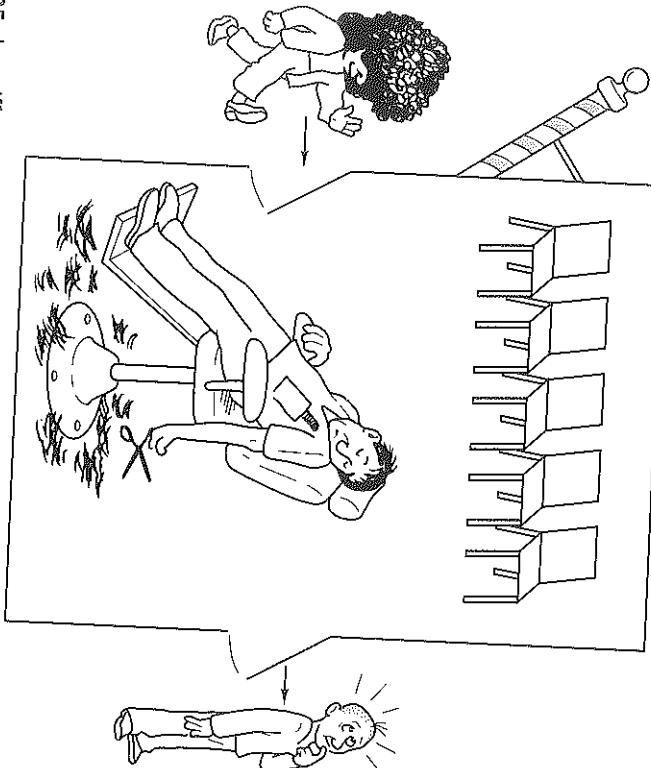


Figure 2.35 • Le coiffeur endormi.

Notre solution utilise trois sémaphores : `customers` (clients), qui compte les clients en attente (excepté celui qui se trouve sur le fauteuil, et qui donc n'attend pas) ; `barbers` (coiffeurs), qui compte le nombre de coiffeurs (0 ou 1) en état inactif, c'est-à-dire attendant un client ; et `mutex`, que l'on utilise pour l'exclusion mutuelle. Nous avons également besoin d'une variable, appelée `waiting`, qui compte les clients en attente. Il s'agit essentiellement d'une copie de `customers`, mais sans elle, il n'y aurait pas moyen de lire la valeur en cours dans un semaphore. Dans cet exemple, le client entrant dans le salon doit compter le nombre de clients en attente. Si celui-ci est inférieur au nombre de chaises, il reste. Sinon, il s'en va. La figure 2.36 illustre le code correspondant. Le fait se bloquer sur le semaphore `customers`, celui-ci étant initialement à 0. Notre coiffeur s'en va donc dormir. Il reste endormi jusqu'à l'arrivée du premier client.

Figure 2.36 • Une solution au problème du coiffeur endormi.

```
#define CHAIRS 5
/* # de chaises en attente de clients */
typedef int semaphore;
/* utilisez votre imagination */
```

```

semaphore customers = 0; /* # de clients attendant une coupe */
semaphore barbers = 0; /* # de coiffeurs attendant un client */
semaphore mutex = 1; /* pour l'exclusion mutuelle */
int waiting = 0;
void barber(void)
{
    while (TRUE) {
        down(&customers); /* s'endort si le # de clients est 0 */
        down(&mutex); /* acquiert l'accès à waiting */
        waiting = waiting - 1; /* décrémente le nombre de clients
                                en attente */
        up(&barbers); /* un coiffeur est prêt à couper des cheveux */
        up(&mutex); /* libère waiting */
        cut_hair();
        /* coupe les cheveux (hors de sa
         * section critique) */
    }
}

void customer(void)
{
    down(&mutex); /* entre en section critique */
    if (waiting < CHAIRS) { /* part si pas de chaise libre */
        waiting = waiting + 1; /* incrémente le nombre de clients
                                en attente */
        up(&customers); /* réveille le coiffeur si nécessaire */
        up(&mutex); /* libère l'accès à waiting */
        down(&barbers); /* s'endort si le # de coiffeurs libres
                            est de 0 */
        get_haircut();
    } else { /* s'assoit pour être servi */
        up(&mutex); /* le salon est plein, n'attend pas */
    }
}

```

Lorsqu'un client arrive, il exécute `customer`, en commençant par acquérir `mutex` pour entrer en section critique. Si un autre client entre juste après, il ne peut rien faire tant que le précédent n'a pas libéré `mutex`. Le client détermine ensuite si le nombre de clients en attente est inférieur au nombre de chaises. S'il ne l'est pas, il libère `mutex` et quitte la boutique sans s'être fait couper les cheveux. S'il y a une chaise disponible, il incrémente la variable entière `waiting`. Ensuite, il effectue un `up` sur le sémaphore `customers`, ce qui réveille le coiffeur. À ce stade, le client et le coiffeur sont tous deux en activité. Lorsque le client libère `mutex`, le coiffeur le récupère, fait son petit ménage et commence la coupe de cheveux.

Lorsque le coiffeur a terminé la coupe, le client quitte la procédure et la boutique. À la différence des exemples précédents, il n'y a pas de boucle pour le client, puisque chaque client ne se fait couper les cheveux qu'une seule fois. En revanche, le coiffeur, lui, dispose d'une boucle qui lui permettra de récupérer le client suivant. S'il y a un client en attente, il effectue une autre coupe. Sinon, il s'endort.

Soulignons que, bien que les problèmes de lecteurs et de rélecteurs, ainsi que ceux qui relèvent de l'exemple du coiffeur endormi n'impliquent pas de transfert de données, on les classe dans les problèmes de communication interprocessus dans la mesure où ils impliquent la synchronisation entre plusieurs processus.