La résolution de tous ces problèmes revient très exactement à réécrire toute la bibliothèque, ce qui n'est pas vraiment simple.

Une autre solution consiste à ajouter du code wrapper à chaque procédure afin de définir un bit pour marquer la bibliothèque comme utilisée. Tant que l'appel antérieur n'a pas été entièrement exécuté, toute tentative émanant d'un autre thread pour utiliser une procédure de bibliothèque est bloquée. Bien qu'il soit possible de rendre 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 originaire 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 gestionnaires 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 développés 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 multithread 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 de réservation aérienne tentent de récupérer pour deux clients différents le dernier siège disponible). Le troisième concerne le séquençage 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 aisé pour les threads dans la mesure où ceux-ci partagent un espace d'adressage commun (les threads qui sont situés dans des espaces d'adressage différents et qui ont besoin de communiquer entrent 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 séquençage 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 2.21 illustre cette situation.

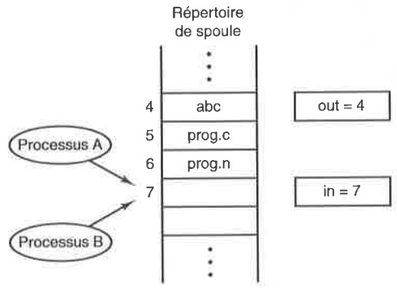

Figure 2.21 • Deux processus veulent accéder à la mémoire partagée en même temps.

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. L'UC, 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 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 partageable –, 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 2.22. 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.

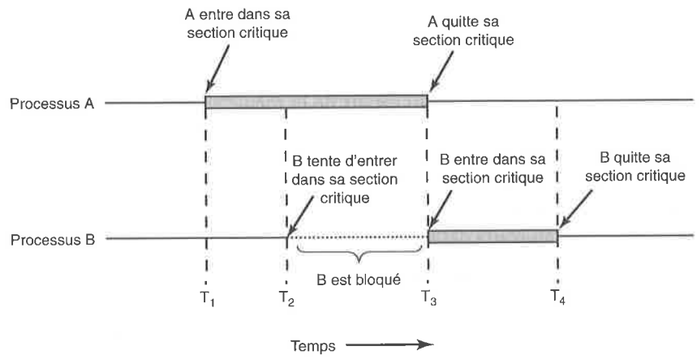

Figure 2.22 • Exclusion mutuelle à l'aide des sections critiques.

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

Sur un système monoprocesseur, 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 utilisateur le pouvoir de désactiver les interruptions. Qu'adviendrait-il s'il 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. 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.

Cette possibilité de réaliser l'exclusion mutuelle en désactivant les interruptions – même dans le noyau – perd beaucoup de son intérêt avec la diffusion des puces multicœurs. Les puces bicœurs sont courantes, les quadricœurs existent sur les machines haut de gamme et les 16 cœurs sont en vue. La désactivation des interruption d'une UC d'une puce multicœur n'empêche pas les autres UC d'interférer avec elle. Il faut mettre en place des méthodes plus élaborées.

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 `0`. Lorsqu'un processus tente d'entrer dans sa section critique, il commence par tester le verrou. Si celui-ci est à `0`, 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 `0`. Ainsi, `0` 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 `0`. 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 2.23 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.