Systèmes d'exploitation

Processus et threads

Pilot Systems - Gaël Le Mignot

INSIA SRT - 2007

Table des matières

Table des figures

Chapitre 1

Processus et threads

1.1 Les processus

1.1.1 Les besoins de la multiprogrammation

Même à l'époque du traitement par lot, le besoin de pouvoir exécuter simultanément plusieurs programmes sur la même machine a commencé à se faire sentir : certains programmes passent une quantité non négligeable de leur temps à lire ou écrire des données sur des périphériques, et il est dommage de faire attendre d'autres programmes, qui eux souhaitent utiliser le CPU, tout en laissant celui-ci inoccupé.

Mais le besoin est devenu fondamental lorsque les ordinateurs sont devenus suffisamment puissant pour permettre à plusieurs utilisateurs de les utiliser simultanément, mais trop onéreux pour permettre d'en avoir un par utilisateur.

De nos jours, même si chaque utilisateur possède en général son propre utilisateur, le besoin d'exécuter plusieurs programmes en parallèle n'a aucunement disparu : qui aimerait devoir quitter son traitement de texte pour consulter son courrier électronique ?

1.1.2 Le concept de processus

Il est donc nécessaire d'avoir, au niveau du système d'exploitation, une entité, nommée "processus" qui permet de représenter un programme en cours d'exécution.

Un processus est en réalité un conteneur de ressources. Il se voit attribuer différentes ressources, comme des zones mémoires (de manière dynamique ou de manière statique suivant les systèmes), des descripteurs de fichier, des accès exclusifs à des périphériques, des droits, ...

Au niveau du système d'exploitation, un processus est en général identifié par un numéro unique. Ce numéro sert d'entrée dans une table, contenant les structures de données nécessaires pour stocker toutes ces informations.

Création et destruction des processus

Une fois le concept de processus acquis, la première question à se poser est comment les processus sont-ils créés et détruits ?

La création de processus se fait avec un appel système. Dans le monde Unix, il s'agit de l'appel `fork`, qui crée une copie exacte du processus parent. Ce processus peut alors se remplacer par un autre programme. Dans le monde Windows, cette création se fait en une seule étape, via un appel système complexe qui crée un nouveau processus. Les deux approches ont leurs avantages et leurs inconvénients, l'approche Unix peut avoir un coût important en performances puisque les informations doivent être dupliquées pour, en général, être détruites peu de temps avant. Mais cette approche apporte plus de flexibilité, par exemple, elle permet de mettre en place un certain nombre de paramètres (variables d'environnement, fichiers ouverts, limites d'utilisation de ressources, ...) sur le processus fils avant de procéder à l'exécution du programme.

La destruction de processus est plus délicate. On peut globalement considérer trois causes de destruction d'un processus :

1. par le processus lui-même, lorsqu'il a fini d'effectuer son traitement ;
2. par le système d'exploitation, en cas d'erreur non réparable ;
3. par un autre processus (par exemple, par une action d'un utilisateur).

La troisième méthode pose elle de très nombreuses questions sur la politique de sécurité, savoir quel processus a le droit, ou non, de détruire un autre processus. Sous Unix, chaque processus appartient à un utilisateur, et ne peut être détruit que par cet utilisateur là, ou par l'administrateur système.

Hiérarchie et groupe de processus

Les processus sont souvent créés pour des raisons spécifiques, afin de coopérer pour réaliser une tâche en commun. Par exemple, un compilateur va exécuter un préprocesseur, un assembleur et un *linker* afin de sous-traiter une partie du travail. Il est important pour lui de savoir si les processus qu'il a créé ont fini leur exécution ou non, et si oui, avec une erreur ou non.

Ce type de situations étant très fréquent, le plupart des systèmes reconnaissent une notion de hiérarchie de processus, chaque processus parent ayant des droits particuliers sur ses processus enfants. Ces droits peuvent inclure le droit de les tuer, ou alors la possibilité de récupérer l'état du processus une fois celui-ci terminé.

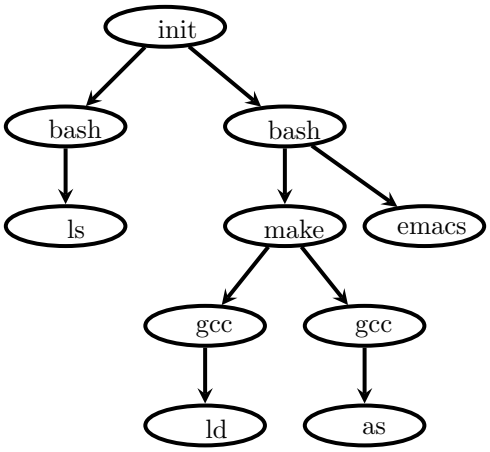

FIG. 1.1 – Arborescence de processus

Cette dernière fonctionnalité est disponible sous Unix, où lorsqu'un processus est tué, celui-ci est en réalité placé en état *zombie*, jusqu'à ce que son parent récupère son état. Si le parent meurt avant son enfant (chose courante dans la vie réelle, mais rare dans le monde injuste de l'informatique), l'orphelin sera alors rattaché à son grand-père. Dans la figure 1.1 par exemple, si le processus `make` venait à mourir d'un accident brutal, ses enfants `gcc` seraient rattaché sur le `bash` servant de parent à `make`.

1.2 Les threads

1.2.1 Le concept de thread

Rappels sur les registres et l'assembleur

Au niveau du microprocesseur, il existe des variables internes, d'accès instantané, nommés des registres. Certains registres sont d'usage généraux (comme `EAX`, `EBX`, `ECX` et `EDX` sur une architecture ia32), d'autres spécifiques (comme `IP` qui contient l'adresse de la prochaine instruction à exécuter).

Au plus bas niveau de programmation, l'assembleur, où chaque instruction correspond exactement à une instruction pour le CPU, la plupart des opérations s'effectuent sur les registres eux-mêmes, avec une instruction particulière pour charger une valeur depuis la mémoire dans un registre, et une autre pour écrire une valeur. Certains processeurs sont capable de manipuler directement des zones mémoires, mais en général ces instructions sont lentes donc peu utilisées.

Les programmes linéaires

Les programmes linéaires, comme nous l'avons vu dans l'introduction, exécutent lors instructions dans l'ordre, avec éventuellement des sauts pour effectuer des boucles ou des conditions. Mais ces programmes là n'ont pas besoin de contexte particulier, la connaissance de la valeur d'IP, c'est à dire, de la prochaine instruction à exécuter suffit entièrement pour reprendre l'exécution d'un programme interrompu.

En réalité, il bien sûr sauvegarder (et restaurer) aussi les registres lorsqu'on l'on veut interrompre et redémarrer un processus.

Addition en C	Addition en assembleur

`a += b + b;`	`mov EAX, a`
	`mov EBX, b`
	`add EAX, EBX`
	`add EAX, EBX`
	`mov a, EAX`

FIG. 1.2 – Exemple de code en assembleur

Les programmes récursifs

La situation se complique quelque peu lorsque les programmes contiennent des fonctions (ou procédures). En effet, savoir à quel endroit de l'exécution d'une fonction se trouve le programme courant ne suffit pas pour continuer son exécution. Il faut aussi savoir à quel endroit retourner, une fois l'appel de fonction courant terminé, et ainsi de suite. Il faut aussi gérer les variables locales aux fonctions et le passage de paramètres.

La solution consiste à utiliser une pile. C'est une zone mémoire spécifique, où les données sont empilées, appel de fonction après appel de fonction, et dépilées lorsque l'on quitte une fonction pour retourner à son appelant.

Les fils d'exécution

Voici donc ce qu'on appelle un *thread* ou en français un fil d'exécution : l'étape des registres du processeur et le contenu de la pile.

En réalité, les systèmes d'exploitation ne sauvegardent et ne restaurent en général que les informations sur la pile (adresse de début et de fin, par exemple), et non son contenu complet, pour des raisons évidentes de performance.

1.2.2 Les problèmes de la programmation séquentielle

Présentation du problème

Dans un programme simple, un seul fil d'exécution suffit amplement. Un shell va lire une ligne de commande, puis l'analyser, puis exécuter le(s) programme(s) puis attendre une nouvelle commande.

Mais dans les programmes complexes, en particuliers interactifs, il y a souvent plusieurs tâches à effectuer au même moment. Par exemple, dans un jeu, il faut traiter en même temps le son, l'affichage, le réseau, les commandes de l'utilisateur et l'intelligence artificielle.

Programmation séquentielle

Le modèle de programmation classique, la programmation séquentielle, permet bien sûr de simuler manuellement de la simultanéité, en faisant des appels périodiques à d'autres fonctions.

Prenons par exemple une application de simulation, qui effectue des calculs lourds mais souhaite maintenir une barre de progression et gérer le bouton “annuler”. On peut dans ce cas, régulièrement, au cours de la boucle principale, appeler le code qui met à jour l'affichage et teste l'appui sur le bouton “annuler”.

Mais ce mode de programmation est assez fastidieux, surtout si le code gérant l'affichage peut lui même prendre du temps pour s'exécuter (par exemple, un affichage via le réseau dans le cadre d'un serveur X distant).

Programmation événementielle

Une autre solution consiste à recevoir, de la part du système d'exploitation (ou du matériel), des évènements. C'est un mode qui est souvent utilisé dans les vieux jeux sous DOS pour gérer le son par exemple, lorsque la carte son a fini de jouer les données qu'elle possédait déjà, elle émet une interruption.

Ce mécanisme, que nous verrons plus en détail lors de l'étude des pilotes de périphériques, consiste à suspendre le fil d'exécution courant du programme, et d'appeler une procédure spéciale (un gestionnaire d'interruption), puis, une fois celui-ci ayant fini son traitement (fournir de nouvelles données à la carte son, par exemple), revenir à l'endroit où on s'était arrêté.

Mais ce mode de fonctionnement est aussi limité, en particulier, il ne peut pas gérer l'exécution de deux traitements consommateurs de temps (comme l'IA et l'affichage graphique).

Les threads

La troisième solution consiste à considérer que dans le même processus, plusieurs fils d'exécution sont en cours de manière simultanée. Une couche logicielle (une bibliothèque en espace utilisateur ou le noyau) doit s'occuper de basculer, régulièrement, d'un thread à l'autre.

Chaque thread, sauf cas particuliers, est implémenté comme s'il était seul.

Comme pour les processus, on possède au minimum deux primitives : une pour créer un nouveau thread (fonction qui prend en général un pointeur de fonction à exécuter), et une pour détruire un thread ou le thread courant.

En général on possède aussi des fonctions capables d'obtenir des informations sur un thread, d'attendre la fin d'exécution d'un thread, ...

Les causes de basculement

On peut citer plusieurs causes possibles pour basculer d'un thread à un autre :
- un thread attend un autre thread ;
- un thread accepte explicitement de donner le CPU à un autre thread ;
- un thread attend un évènement extérieur ;
- un quota de temps CPU est épuisé.

1.3 Les threads : user-space ou kernel-space ?

1.3.1 Les threads en espace utilisateur

Les threads en espace utilisateur simples

Les threads en espace utilisateur sont en général implémentés sous la forme d'une bibliothèque (comme la GNU pth). Dans le modèle le plus simple, cette bibliothèque fournit donc des fonctions de création, destruction et manipulation de threads. Elle stocke en interne la liste de tous les threads créés, avec le contexte nécessaire pour les restaurer. L'opération est relativement simple, il s'agit de sauvegarder tous les registres et les informations sur la pile.

Une telle bibliothèque doit aussi fournir un appel, en général nommé `thread_yield` qui permet au thread courant de libérer le CPU. À l'appel de cette fonction, la bibliothèque va choisir un nouveau thread parmi ceux qui ont été créés et lui donner le CPU.

Ce modèle ressemble à celui de notre application de simulation, qui appelait manuellement le code gérant l'affichage et l'entrée utilisateur de manière régulière, sauf qu'il est beaucoup plus générique : je n'ai pas à savoir si c'est l'affichage, le son, l'entrée utilisateur ou le réseau qu'il faut gérer, un seul appel permet de basculer à une autre partie du programme, choisie selon les besoins.

Le choix lui-même du thread à exécuter peut dépendre de beaucoup de paramètres, comme une priorité, nous verrons ces choix en détail dans la partie du cours sur l'ordonnancement.

La gestion des appels systèmes bloquant

Il reste un gros problème dans le modèle précédant : la gestion des appels systèmes bloquant. Si l'un des threads doit lire des données depuis le réseau, il va effectuer un appel `read` et être bloqué en attendant que les données arrivent, ce qui peut prendre du temps. Or, pendant l'appel `read`, le programme est en mode noyau, et il n'y a aucun moyen d'appeler `thread_yield`.

Une solution consiste à passer les appels en non bloquant, et à exécuter une boucle du type de la figure 1.3. Une boucle de ce type résout le problème, mais au prix d'un coût élevé en CPU : si jamais tous les threads sont bloqués, on va constamment passer d'un thread à l'autre, utilisant tout le CPU disponible.

Les bibliothèques de threads comme la pth implémentent une version plus écologique de `thread_read`, et fournissent même des astuces (via `#define` ou via le *linker*) pour remplacer de manière transparente les appels à `read`.

Le seul problème restant, mais qui n'est pas négligeable, est la nécessité d'appeler `thread_yield` régulièrement. Une partie du programme ne le faisant pas (à cause d'un bug, ou parce que c'est une bibliothèque qui n'avait pas été écrite pour un environnement avec des threads) gardera le CPU pour toujours, et empêchera l'exécution des autres threads.

```
size_t thread_read(int fd, void *data, size_t count)
{
  size_t done;

  while ((done = read(fd, data, count)) < 0)
   {
      if (errno != EAGAIN) /* This was a real error */
        return done;
      thread_yield(); /* Yield the CPU */
   }
  return done; /* We got data */
}
```

FIG. 1.3 – Fonction `read` non bloquante

1.3.2 Les threads en espace noyau

Principes de fonctionnement

Afin de répondre aux deux problèmes précédant (la gestion des appels systèmes bloquant, et la possibilité pour un thread de garder le CPU), il est aussi possible de mettre les threads dans l'espace noyau.

Dans ce cas (comme nous allons le voir plus en détails dans la partie sur l'ordonnancement), le système d'exploitation va interrompre le programme après qu'il ait pu utiliser le CPU pendant un intervalle de temps donné (1ms ou 10ms en général), ou alors s'il fait un appel qui devrait bloquer. Le système choisit alors un autre thread, ou un autre processus, et lui donne le CPU.

Possibilités d'implémentation

Il y a deux solutions pour implémenter les threads en espace utilisateur.

La solution classique La solution classique consiste à ajouter, pour chaque processus, une liste de threads (comme il possède déjà une liste de fichiers ouverts, une liste de zones mémoires où il a accès, ...).

La solution Linux La solution utilisée dans Linux consiste à considérer les threads comme des processus. Sous Linux, l'appel système `clone` est un `fork` ayant des options additionnelles, qui permettent d'indiquer que le nouveau processus va partager telles ou telles ressources (la mémoire utilisée, les fichiers ouverts, ...) avec son fils.

La solution classique est en général plus performante et théoriquement plus pure, mais la solution Linux a l'avantage de permettre de créer des entités hybrides entre les processus et les threads, qui auront par exemple la même mémoire mais pas les mêmes fichiers ouverts.

1.3.3 Avantages et inconvénients des deux modèles

Avantages des threads en espace utilisateur

Le premier argument en faveur des threads utilisateur est qu'ils ne nécessitent aucune modification du système d'exploitation, ils peuvent être utilisés dans des applications portables qui peuvent être amenés à fonctionner sur des systèmes ne gérant pas les threads.

Le deuxième argument en faveur des threads utilisateurs est le coût (CPU) plus réduit de l'implémentation des primitives. En effet, un appel système coûte cher (plusieurs milliers de cycles sur les machines récentes), alors que la plupart des opérations (création, bascule, destruction, ...) ne nécessitent en réalité aucun privilège particulier.

Le troisième argument en faveur des threads en espace utilisateur est qu'ils permettent à l'application de gérer elle-même ses priorités, avec ses propres politiques dépendant du traitement. Par exemple dans un jeu, il est indispensable que le thread gérant le son puisse s'exécuter quand il le souhaite, un délai très faible peut se traduire dans un craquement désagréable à l'oreille, tandis que l'intelligence artificielle peut elle se permettre un délai d'un dixième de seconde supplémentaire. De même dans une base de données, un thread procédant à une ré-indexation en tâche de fond est moins important qu'un thread répondant à une demande précise.

Avantages des threads en espace noyau

Les threads en espace noyau ont surtout l'avantage qu'ils sont préemptifs et éliminent donc le risque de monopolisation du CPU par un seul thread. Ils sont aussi plus simples à utiliser, puisqu'il n'y a pas besoin de

`thread_yield`.

Enfin, ils gèrent les appels systèmes bloquant sans nécessiter aucune bidouille, et sans aucun coût supplémentaire.

En pratique, les threads en espace noyau sont de plus en plus répandus, pour la facilité qu'ils offrent. Le léger coût supplémentaire qu'ils créent est considéré comme négligeable dans un monde de processeurs allant à plusieurs gigahertz.

1.3.4 Les modèles hybrides

Entre les deux modèles vus précédemment, il existe quelques modèles hybrides :

Le multiplexage de threads Ce modèle consiste à utiliser quelques threads noyau, en petit nombre, chacun possédant un ou plusieurs (en général beaucoup) de threads utilisateurs.

Le mécanisme de scheduler activation Ce modèle consiste à implémenter les threads en espace utilisateur, mais à permettre au noyau d'appeler lui-même une version modifiée de `thread_yield` lorsqu'un appel système devrait bloquer, ou lorsque le quota de temps CPU est écoulé. Ce modèle est assez délicat à implémenter, mais possède la plupart des avantages et peu d'inconvénients.

1.4 Problématiques posées par les threads

1.4.1 Les problèmes de synchronisations

Considérons un serveur web multi-threadé, donc chaque thread sert des fichiers. Ce serveur web, pour des raisons de statistiques, souhaite maintenir un compteur global du nombre d'octets transmis sur le réseau.

Le code est en théorie simple : une variable globale est incrémentée après chaque transfert réseau, mais comme on peut le voir sur la figure 1.4, le code assembleur correspondant prend lui 4 instructions. Or si on regarde le scénario de la figure 1.5 on voit bien qu'il y a un problème, ce que l'on nomme une *race condition*.

Code en C	**Code en assembleur**

`total += count;`	`mov EAX, total`
	`mov EBX, count`
	`add EAX, EBX`
	`mov total, EAX`

FIG. 1.4 – Un compteur global

1. Le `total` est initialement à 1000 ;
2. Un thread (1) vient d'écrire 100 octets sur le réseau ;
3. Il charge donc 1000 dans EAX et 100 dans EBX ;
4. Le système interrompt alors le thread et donne le CPU au thread (2) ;
5. Le thread (2) vient lui d'écrire 50 octets sur le réseau ;
6. Il charge donc 1000 dans EAX et 50 dans EBX ;
7. Il réalise l'addition, et met donc 1050 dans EAX ;
8. Il stocke le résultat dans `total` qui vaut maintenant 1050 ;
9. Le système interrompt ce thread (2) et redonne le CPU dans au thread (1) ;
10. Les valeurs d'EAX et EBX sont restaurées à 1000 et 100 ;
11. Le thread (1) effectue l'addition, et met donc 1100 dans EAX ;
12. Il stocke le résultat dans `total` qui vaut maintenant 1100.
13. Nous avons donc $1000 + 100 + 50 = 1100$.

FIG. 1.5 – Scénario du compteur global

On voit donc bien qu'il faut un mécanisme de synchronisation pour éviter ce genre de scénarios, qui sont d'autant plus difficiles à corriger que le comportement est "aléatoire", le programme fonctionnera en général, mais sur une coup de malchance (la bascule se faisant au mauvais moment) il donnera des résultats erronés.

Nous verrons en détail les solutions possibles dans la partie sur la communication entre les processus (IPC).

1.4.2 Les problèmes de transition d'un modèle à l'autre

Il existe aussi des problèmes spécifiques à la migration de programmes et de concepts créés dans un environnement monothreadé vers un environnement multithreadé. Voyons quelques exemples.

Les threads et errno

Le premier problème concerne les variables globales, et en particulier la variable `errno` qui contient le code d'erreur du dernier appel système effectué.

Si on ne prend pas de précaution, le même scénario que celui décrit plus haut est possible : un thread va effectuer une opération de lecture, mais avant d'avoir lu `errno`, un autre thread va obtenir le CPU et effectuer une écriture. Le premier thread va alors lire la valeur de l'erreur de l'opération d'écriture.

La solution consiste à remplacer ces variables globales par des variables locales au thread, avec un mécanisme (au niveau de la libc pour `errno`) qui permet de donner l'impression qu'il s'agit d'une variable globale, alors qu'en réalité il s'agit d'un appel à une fonction.

Les threads et fork

Le deuxième problème concerne l'appel `fork` : quand un processus `fork`, faut-il dupliquer uniquement le thread courant, le thread principal, ou bien tous les threads ?

La réponse dépend du système d'exploitation, sous GNU/Linux, seul le thread courant est recopié.

Les threads et les signaux

Le troisième exemple concerne la gestion des signaux. Les signaux sont un mécanisme sous Unix qui permet d'effectuer une interruption dans un programme. Un exemple de signal est `SIGINT`, envoyé quand on presse `Ctrl-C`.

Mais dans quel thread le signal doit-il être géré ? Il y a plusieurs réponses possibles :

1. Le thread principal ;
2. Le thread courant ;
3. Le thread qui a demandé à recevoir le signal ;
4. Un thread créé spécialement pour le traitement du signal et détruit à la fin du traitement.

Le thread effectuant le traitement est particulièrement important dans le cas d'un langage de haut niveau qui peut traduire ces signaux en exceptions.

La convention est de donner les signaux au thread principal sous Unix, mais il n'y a aucune garantie à ce niveau là.

Chapitre 2

Scheduling

2.1 Principes du scheduling

2.1.1 Définitions

Le *scheduling* ou *ordonnancement* en français consiste à décider dans l'ordre dans lequel les différents processus auront accès au CPU.

Dans le cadre des systèmes d'exploitation, le terme est en général employé uniquement pour le CPU, mais si dans certains cas il peut être utilisé pour d'autres ressources.

Dans l'informatique en général, le terme et les principes du scheduling s'appliquent très souvent, on peut avoir un scheduler de requêtes dans un serveur Web, un scheduler d'accès aux tables dans une base de données, ...

Le scheduling est d'ailleurs important aussi dans des problèmes de la vie courante, comme la gestion des files d'attente, l'attribution des salles d'opération dans un hôpital, ...

2.1.2 Scheduling et threads

Sauf lorsque la mention est utile, nous considérerons dans la suite de cette partie qu'il s'agit d'un système ne gérant pas de threads en mode noyau. Considérer systématiquement les threads ne ferait qu'alourdir et complexifier sans apporter grand chose pour la compréhension des mécanismes.

Un paragraphe sur la gestion des threads sera parfois ajouté afin d'évoquer les aspects particulier de cette gestion.

2.1.3 Les types de scheduling

Comme pour les systèmes d'exploitation, on peut classer les systèmes de scheduling selon deux critères.

Le premier critère consiste à regarder si le système est préemptif ou non-préemptif. Un système préemptif va de lui-même reprendre le contrôle du CPU (en général en utilisant une horloge programmée à une fréquence fixée, comme 100Hz ou 1000Hz) pour le donner à un autre processus. Un système non-préemptif va attendre que le processus donne volontairement le CPU, ou fasse un appel système bloquant.

Le deuxième critère est le type de machines concerné, on considère trois classes de scheduler : les scheduler pour systèmes de traitement par lot, les scheduler pour systèmes interactifs et les scheduler pour systèmes temps-réels. Les systèmes temps réels seront étudiés plus tard dans le cours, et donc seront ignorés pour l'instant.

2.1.4 Les principes du scheduling

Équité Le premier principe est le principe d'équité, deux processus comparables doivent autant que possible avoir accès au CPU avec la même fréquence.

Efficacité Le deuxième principe est l'efficacité. Un bon algorithme de scheduling fera en sorte d'utiliser simultanément les ressources disponibles. Ce point est le plus complexe, et nous le verrons en détail par la suite.

Respect de la politique Le troisième principe est de pouvoir définir, et faire respecter, des politiques. Le type de politique dépend du domaine, mais un exemple de politique peut être : « l'application en premier plan est prioritaire sur les applications en tâche de fond » ou alors dans une institution militaire « les programmes lancés par une personne de grade plus élevé sont prioritaires ».

Faible surcharge Le quatrième principe est que l'algorithme et les mécanismes de scheduling ne doivent eux-mêmes coûter trop cher en CPU. En particulier, pour un algorithme préemptif, il faut compter le temps passé à régulièrement interrompre un processus et basculer sur un autre, une opération qui est souvent lourde.

2.1.5 Processus R et S

Dans tous les systèmes de scheduling, à un instant donné, on peut diviser les processus en deux classes : les processus qui sont bloqués en attente d'I/O (S pour *sleeping*) et ceux qui sont désirent utiliser le CPU (R pour *running*).

Au niveau de l'implémentation, il y a en général deux structures de données (des listes par exemple) qui contiennent ces deux classes de processus. Un algorithme de scheduling ne considère en général que les processus R (en attente du CPU).

2.2 Comportement des processus

2.2.1 Comportement général

Le comportement général d'un processus est d'attendre de recevoir des données (depuis l'utilisateur, le disque, le réseau, ...), puis de traiter ces données, et ensuite d'attendre de recevoir de nouvelles données. Un processus peut aussi être en attente d'écriture de données.

On considère donc que la vie d'un processus alterne entre des périodes d'utilisation du CPU et des périodes d'attente d'un périphérique.

Dans le cadre de processus communiquant entre eux (qui seront étudiés en détails par la suite), un processus en attente d'un autre processus est considéré comme en attente d'I/O (pour le processus lui-même, un autre processus n'est pas fondamentalement différent d'un périphérique).

2.2.2 Processus "I/O bound" et "CPU bound"

Si tout processus se compose de phases d'utilisation du CPU et de phases d'attente, ces phases ne sont pas dans les mêmes proportions ni dans les mêmes durées suivant les types de processus. Un shell par exemple aura des phases d'attente très longues (le temps que l'utilisateur saisisse la ligne de commande) mais une utilisation du CPU très faible (sauf peut-être si vous lancer "echo */*/*"). Un lecteur de films aura lui des temps d'attente beaucoup plus faibles, et des durées d'utilisation du CPU beaucoup plus élevés.

On considère en général qu'il y a deux classes de processus : des processus « I/O bound » et des processus « CPU bound ». Les processus « I/O bound » passent la plupart de leur temps à attendre, les processus « CPU bound » à utiliser le CPU.

Cette classification, comme toute classification en deux catégories, est bien sûr très imprécise. En particulier, un processus peut changer de comportement suivant les étapes de sa vie. Un jeu d'échecs aura des phases où il sera « I/O bound », quand il s'agit du tour de l'utilisateur, et que le programme lui ne fait que gérer l'interface utilisateur (décompte du temps restant, afficher les coups possibles quand on clique sur une pièce, quelques effets graphiques). Par contre, lorsque c'est le tour de l'intelligence artificielle, le jeu sera « CPU bound » et tentera de trouver le meilleur coup possible le plus vite possible.

À noter que pour les algorithmes de scheduling, le plus important n'est pas tant la durée des phases d'I/O que leur fréquence, c'est à dire, la durée des phases de CPU.

2.3 Scheduling des systèmes de traitement par lot

2.3.1 Intérêt de la multiprogrammation

Un système de traitement par lot n'a aucune contrainte d'interactivité, et l'intérêt de la multiprogrammation peut sembler faible dans ce genre d'environnement. Pourquoi complexifier le système (et donc augmenter les risques d'erreur) alors qu'exécuter les processus un à un conviendrait parfaitement ?

La raison est que, justement, les processus alternent entre des phases d'I/O et des phases d'utilisation du CPU. Dans ce cadre, n'avoir qu'un seul processus à la fois signifie que le CPU sera inutilisé pendant une partie du temps. Si nous avons deux processus ayant chacun une seconde d'utilisation du CPU, puis une seconde d'I/O, et ainsi de suite, pendant 20 secondes, et que nous les exécutons à la suite, nous aurons besoin de 40 secondes pour les exécuter tous les deux. Si on utilise de la multiprogrammation, et que chaque processus utilise le CPU pendant que l'autre attend les disques, les deux processus pourront être exécutés en un peu plus de 20 secondes.

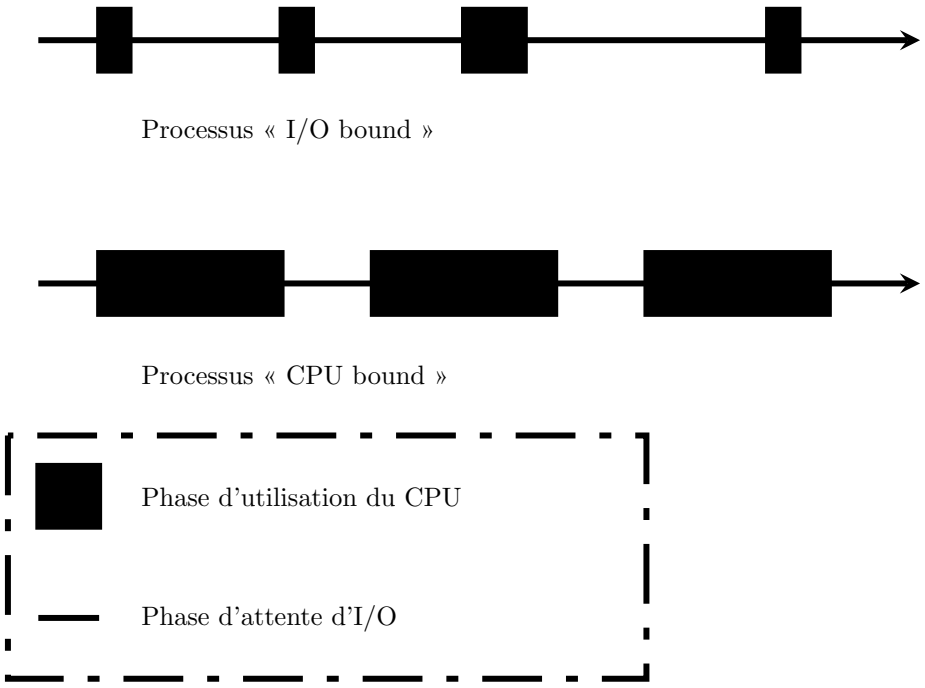

FIG. 2.1 – Comportement des processus

Bien sûr, le cas est rarement aussi idyllique, mais l'idée est bien là : en utilisant de la multiprogrammation on peut accroître les performances globales du système.

Par contre, dans un système de traitement par lot, l'intérêt d'un système préemptif est en général très faible. Nous considérerons par la suite que le système n'est pas préemptif : une fois qu'on a donné le CPU à un processus, il le garde jusqu'à ce qu'il fasse un appel bloquant (ou qu'il ait terminé son exécution).

2.3.2 Objectifs d'un bon algorithme

Comment mesurer l'efficacité d'un système de traitement par lots ? Il existe plusieurs moyens d'essayer de la quantifier.

Minimisation du délai d'attente Le premier consiste à vouloir minimiser le délai d'attente moyen entre l'arrivée d'un programme dans le système et l'obtention des résultats.

Maximisation du débit Le deuxième consiste à vouloir maximiser le nombre de processus ayant pu s'exécuter dans une période de temps donnée (une heure, par exemple).

Utilisation maximale du CPU Le troisième critère consiste à vouloir utiliser au maximum le CPU, et considérer que si le CPU est utilisé à 100% alors nous n'aurions pas pu faire mieux, de manière globale.

2.3.3 Exemples d'algorithmes

FIFO

Le premier algorithme est le plus simple : on prend le premier processus arrivé et on l'exécute, puis le suivant.

Avec de la multiprogrammation, l'algorithme n'a pas besoin d'être beaucoup modifié : on exécuté à chaque moment le processus le plus ancien (celui qui est arrivé le premier) et qui n'est pas en attente.

Cet algorithme est équitable, mais il ne répond pas vraiment aux contraintes de minimisation du temps d'attente et de maximisation du débit. Un processus « CPU bound » nécessitant deux heures de calcul va bloquer le système, et empêcher les petits processus (quelques minutes, voir moins) qui arrivent par la suite d'être traités.

Shortest first

Afin d'éviter ce problème, il existe un algorithme nommé *shortest first*. Il nécessite une estimation préalable du temps que prendra l'exécution de chaque processus (ce qui n'est pas irréaliste dans le cadre d'un système de

traitement par lot).

Il consiste, pour la version simple, à prendre le processus ayant le temps total d'exécution le plus et de lui donner le CPU. Avec la multiprogrammation, il s'agit à un instant donné de choisir le processus dont le temps d'exécution restant (estimé) est le plus faible (parmi les processus R, bien sûr).

Considérons, sans multiprogrammation, 5 processus, d'un temps d'exécution respectif de 5 minutes, 20 minutes, 2 minutes, 10 minutes et 3 minutes. Si on les exécute dans l'ordre, le premier processus aura fini son traitement après 5 minutes, le deuxième après 25 minutes, le troisième après 27 minutes, le troisième après 37 minutes et le dernier après 40 minutes. Le temps de traitement moyen sera donc de $\frac{5+25+27+37+40}{5} = 26$ minutes.

Maintenant, si on les exécute avec l'algorithme *shortest first*, l'ordre sera 3, 5, 1, 4, 2 et ils seront finis après 2, 5, 10, 20 et 40 minutes. Soit un temps de traitement moyen de $\frac{2+5+10+20+40}{5} = 15$ minutes. On constate bien une amélioration importante du temps de traitement moyen.

Le problème est qu'on a maintenant abandonné l'équité, et qu'un processus pourra attendre très longtemps pour même avoir une chance de s'exécuter si on continue à alimenter le système en processus plus courts.

Une solution consiste, à chaque fois qu'un processus plus ancien est ignoré au profit d'un processus plus récent mais plus court, à diminuer (pour l'algorithme de scheduling uniquement) son temps estimé. Par exemple de le multiplier par 0.75 à chaque fois, ainsi, on est sûr qu'après quelques processus courts, le processus long aura sa chance.

Scheduling à trois niveaux

Considérons un système de traitement par lot relativement simple. Il peut contenir jusqu'à 4 processus en mémoire. Mais il est possible de *swapper* un processus entier vers une zone de stockage, et on peut donc avoir jusqu'à 12 processus en même temps dans le système.

Nous avons donc trois niveaux de scheduling :

1. Tout d'abord, un scheduler d'admission, qui va choisir quels processus introduire dans le système pour maintenir les 12 processus actifs. Ce scheduler peut utiliser un *shortest first* par exemple.
2. Ensuite, un scheduler mémoire va choisir quels processus parmi les 12 seront chargés en mémoire. Cette décision peut être reconsidérée régulièrement, mais avec un intervalle plutôt élevé, car l'opération est coûteuse.
3. Enfin, un scheduler CPU, qui ne s'occupe que des 4 processus chargés en mémoire, et qui peut lui prendre des décisions fréquentes.

2.4 Scheduling des systèmes interactifs

2.4.1 Fonctionnement général des schedulers préemptifs

Rappels du principe

Comme nous l'avons rapidement vu en introduction, le scheduler est dans ce cas appelé à intervalles réguliers (en général toutes les 10ms, voir toutes les 1ms pour les systèmes plus récents), et remplace le processus courant par un autre si nécessaire.

Il est bien sûr aussi appelé à chaque I/O bloquante.

Considérations sur les systèmes interactifs

Dans les systèmes interactifs, le plus important n'est pas le débit, mais la réactivité perçue des programmes. Un utilisateur est près à attendre 5 minutes pour une compilation, mais pas à attendre 2 secondes pour afficher une boite de dialogue.

De plus, pour donner l'illusion de simultanéité, il est nécessaire que chaque processus puisse avoir accès au CPU plusieurs fois par seconde, d'où la nécessité d'un mécanisme préemptif de durée assez courte (10ms ou 1ms en général).

2.4.2 Quelques algorithmes de scheduling

Round-robin

Le système le plus simple, appelé round-robin, consiste à mettre tous les processus R dans une liste circulaire, et d'exécuter à chaque fois le processus suivant dans la liste. Ce système est juste, dans le sens ou chaque processus est traité équitablement.

Lorsqu'un nouveau processus arrive (ou lorsqu'un processus a fini d'attendre des données), il est en général ajouté à la fin de la liste, et aura donc le CPU une fois que tous les autres ont eu au moins un quantum.

Priorité

La prise en compte de priorités fixées par l'administrateur système peut se faire en divisant la liste circulaire R en autant de listes que de niveaux de priorité. Chaque processus est affecté à un niveau de priorité particulier (suivant l'utilisateur qui l'a lancé ou suivant que le processus soit en tâche de fond ou en premier plan, par exemple).

L'algorithme regarde d'abord la liste des processus de plus haute priorité, et s'il en existe au moins un, effectue un round-robin parmi ceux de cette liste. Sinon, il regarde le niveau de priorité suivant, et ainsi de suite.

Ce modèle est un modèle à priorité forte : le temps qu'un processus de haute priorité est présent, aucun processus de priorité plus basse ne sera exécuté. En général, on souhaite plutôt des priorités faibles, les processus de priorité plus élevés s'exécutent plus souvent, ceux de priorité plus basse plus rarement. Par exemple, on peut compter le nombre de processus par file de priorité, multiplier par la priorité, et utiliser ce résultat pour choisir une file. Ainsi, avec 2 processus dans une file de priorité 4 (valeur totale 8) et 4 processus dans une file de priorité 1 (valeur totale 4), la file de priorité de valeur 4 sera choisie deux fois plus souvent que la file de valeur 1.

Quantum flexibles

Certains processus ont besoin de calculer longuement pour donner des résultats ; d'autres ont au contraire besoin d'avoir le CPU souvent, mais pour de courtes durées.

Basculer d'un processus à l'autre prend du temps, et il est inefficace de basculer 50 fois d'un processus à l'autre si celui-ci demande 50ms de traitement, alors que le quantum est à 1ms.

L'idée de cet algorithme est d'avoir des classes de priorité, par exemple de 1 à 5. Les processus de la classe 1 auront le CPU souvent, mais pour un seul quantum (1ms). Les processus de la classe auront le CPU deux fois moins souvent mais pour deux quanta (2ms). La classe suivante aura la CPU quatre fois moins souvent, mais pour 4 quanta (4ms). Et ainsi de suite, jusqu'à la classe 5 qui aura le CPU 16 fois moins souvent, mais pour 16 ms.

Les processus sont au départ dans la classe 1. Un processus qui utilise la totalité de son quantum sans faire d'appel bloquant sera déplacé au niveau suivant (donc du niveau 1 au niveau 2). Au contraire, un processus qui a fait un appel bloquant au cours de son quantum sera lui déplacé au niveau précédant.

Ainsi le processus de 50 ms aura d'abord un quantum (1ms), puis se verra déplacé dans la classe 2, où il recevra (certes moins souvent) 2ms d'un coup, puis dans la classe 3 où il aura 4ms d'un coup, et ainsi de suite. Au total il s'exécutera pour 1, 2, 4, 8, 16, 16 et 3 ms, soit seulement en 7 fois, comparé aux 50 fois d'un algorithme classique.

Shortest next

Comme pour le traitement par lot, il peut être intéressant d'exécuter à chaque fois le processus ayant le temps de traitement le plus court avant d'effectuer une I/O, et ceci afin d'augmenter l'utilisation globale des ressources et diminuer le temps d'attente moyen.

Mais dans le cadre des systèmes interactifs, il est impossible de connaître le futur, puisqu'il dépend énormément des entrées de l'utilisateur, ou de paramètres extérieurs (comme le réseau). L'idée est donc d'utiliser le passé pour prédire l'avenir.

Ce qui nous intéresse dans ce cas là est le temps entre deux appels systèmes bloquant. Si entre les deux derniers appels bloquant, le processus a utilisé un quantum de 10ms, puis 4ms du quantum suivant, on peut considérer que son besoin de CPU est de 14ms.

Mais bien sûr considérer uniquement l'itération précédente est souvent un mauvais choix, on utilise donc en général une moyenne pondérée des différentes exécutions précédentes. Le principe s'appelle *aging* ou *scoring* : chaque processus a un score (initialisé au départ à une valeur fixe, par exemple 10ms), et à chaque fois qu'il fait un appel système bloquant, son score (S) est réajusté suivant le temps qu'il a utilisé ce coup-ci (T) à $\alpha S + (1 - \alpha)T$. Plus α est élevé, et plus le passé ancien sera important, plus α est faible, et plus le passé récent le sera.

Loterie

Un autre algorithme de scheduling consiste à donner des tickets de loterie à chacun des processus. Un tirage au sort est effectué, et le possesseur du ticket gagnant aura le CPU. Un processus ayant deux fois plus de tickets aura deux fois plus de chances d'obtenir le CPU.

Ce mécanisme permet une flexibilité assez grande dans la distribution des tickets pour implémenter différentes politiques, mais pose aussi un grand nombre de questions. Par exemple, les tickets sont-ils conservés entre deux itérations ? Les processus peuvent-ils se donner mutuellement des tickets ?

Modèle économique

Dans la même idée que le système de loterie, il est aussi possible d'utiliser un modèle économique pour l'allocation des ressources (CPU, mémoire, bande passante réseau, ...). Chaque processus reçoit des crédits dépendant de la politique choisie par l'administrateur et/ou l'utilisateur, et peut acheter des ressources avec ses crédits.

Le prix des ressources est lui fixé par la loi de l'offre et de la demande, si beaucoup de processus souhaitent utiliser le CPU mais peu la mémoire, alors le CPU sera cher et la mémoire non. Ce système a l'avantage de permettre aux processus de s'adapter par eux-mêmes, choisissant un algorithme ou un autre suivant le prix des différentes ressources, et de fournir une grande flexibilité.

Mais il est très complexe à implémenter et à paramétrer, car beaucoup de questions se posent : peut-on économiser de l'argent ? Peut-on en donner ou en prêter à une autre processus ? Comment lutter contre la spéculation ?

Ce modèle étant très complexe, il n'est pas utilisé dans la pratique, mais de nombreux projets de recherche l'utilisent.

Processus, groupes, utilisateurs et threads

Jusqu'à présent, nous avons traité les processus indépendamment les uns et des autres, et nous n'avons pas considéré les threads.

Ce choix n'est pas forcément le plus approprié. Si un utilisateur A possède une dizaine de processus actifs mais un utilisateur B un seul, et qu'on considère les processus de manière neutre, l'utilisateur A aura beaucoup plus souvent le CPU que l'utilisateur B, ce qui peut être considéré comme injuste. Choisir d'abord l'utilisateur a qui donner le CPU, puis ensuite pour cet utilisateur un processus peut être plus judicieux.

La même chose est imaginable avec des groupes de processus. Un traitement de texte va être un processus unique, tandis qu'une compilation va être un ensemble de processus (`make`, `cpp`, `gcc`, `ld`, `as`), mais chaque groupe réalise en réalité une seule tâche. Il peut donc être judicieux de définir des groupes de processus, et de choisir d'abord un groupe, puis à l'intérieur de ce groupe un processus.

Pour les threads, le problème est similaire. Une première approche est de considérer les threads comme autant de processus. Mais en plus des notions d'équité, on a aussi des considération de performance : basculer d'un thread à un autre dans le même processus est moins coûteux que de basculer d'un processus à un autre. Il peut donc être intéressant de donner le CPU à un autre thread du même processus, en particulier quand le thread n'a pas épuisé son quantum de temps, mais a fait un appel bloquant.

2.5 Politique versus mécanisme

Comme toujours, il est important de se souvenir que si le mécanisme de scheduling doit être fait en espace noyau pour des raisons de sécurité, il est envisageable de coupler ce mécanisme et la politique de scheduling, qui elle peut être mis en espace utilisateur pour plus de flexibilité.

Par exemple, il est possible d'avoir un scheduler système qui répartit le CPU entre chaque utilisateur, mais ensuite chaque utilisateur peut mettre son propre scheduler pour gérer ses processus à lui, selon ses besoins.

Chapitre 3

Communication et deadlocks

3.1 Principes de l'inter-process communication (IPC)

3.1.1 Rôles et utilité de l'IPC

Il est souvent nécessaire à des processus de communiquer entre eux. Par exemple, lorsqu'un compilateur exécute un assembleur, il a besoin de lui communiquer le code assemblé du programme. Un jeu d'échec peut être divisé en deux programmes indépendant, une intelligence artificielle et une interface graphique, afin de dissocier les deux et de permettre de changer un composant tout en gardant l'autre.

La communication entre les programmes graphiques et le serveur X sous Unix est un autre exemple de communication, qui dans ce cas-ci possède des contraintes fortes de performances.

La communication entre les processus porte souvent le nom d'IPC, pour l'anglais *inter-process communication*.

3.1.2 Les mécanismes standards d'IPC

Les signaux

Les signaux consituent la forme la plus primitive d'IPC. Les signaux sont identifiés par un numéro, de 0 à 15 ou de 0 à 63 par exemple, et à chaque signal une application peut y associer une fonction (via un appel système), qui sera appelée lorsque le signal sera reçu.

Les signaux sont envoyés par un autre appel système, qui prend en paramètres un identifiant du processus cible et le numéro du signal. Une vérification de droits simple est effectuée (par exemple un processus ne peut envoyer un signal qu'à un processus appertenant au même utilisateur, sauf exceptions).

Certains signaux peuvent avoir une signification particulière, qui ne peut pas être modifiée par le processus lui-même, comme `SIGKILL` sous Unix qui détruit le processus, ou `SIGSTOP` qui suspend temporairement son exécution.

Les signaux sont en général utilisés pour donner des ordres simples à des processus non interactifs (par exemple pour demander à un serveur de relire son fichier de configuration), ou pour signaler des erreurs (le fameux `SIGSEGV` sous Unix, par exemple).

Il faut noter que les signaux, en général, ne s'empilent pas : si deux signaux est envoyés au même processus avant que celui-ci n'ait eu accès au CPU, le comportement n'est pas défini. Cette limitation empêche toute utilisation fréquente des signaux.

Communication via les fichiers

Le mode de communication le plus ancien entre deux processus, et qui peut même fonctionner si les processus sont exécutés consécutivement et non simultanément, consiste à écrire les données dans un fichier, puis à lire ce fichier. Seul le nom du fichier a besoin d'être connu, et il peut être soit fixé par convention au préalable (par exemple dans un fichier de configuration), soit communiqué dans les arguments du programme (lorsqu'un processus en exécute un autre, il peut en général lui communiquer des arguments).

Cette communication est simple et souvent utilisée, mais possède plusieurs inconvénients :

- l'écriture et la lecture de fichiers sur le disque est une opération coûteuse ;
- les données peuvent être interceptées ou modifiées par d'autres processus ;
- le nom du fichier doit être soit fixé, soit communiqué par un autre moyen.

Sous Unix, il existe une astuce permettant d'éviter en partie les deux derniers problèmes : il est possible de créer un fichier, l'ouvrir en lecture et en écriture, puis supprimer le fichier. Le fichier ne sera plus accessible pour les autres processus, mais restera présent sur le disque le temps que le descripteur de fichier ne sera pas fermé.

Il est alors possible d'utiliser l'appel `fork` afin de créer un autre processus, qui lui aussi aura un descripteur de fichier vers notre fichier temporaire.

Les fichiers virtuels

Une autre solution, plus propre, consiste à modifier le système d'exploitation pour lui permettre de gérer des fichiers virtuels, ayant la même sémantique (lecture et écriture) que des fichiers normaux, mais servant uniquement à communiquer entre deux processus.

Les pipes Le moyen le plus simple est le *pipe* Unix, qui est une sorte de tuyau entre deux processus. Un pipe se compose de deux descripteurs de fichiers connectés entre eux : tout ce qui est écrit sur le premier sera sur le deuxième.

Si on regarde la commande classique `ls | more`, qui envoie la sortie du programme `ls` vers le programme `more`, que se passe-t-il ?

1. Le shell crée un pipe, entre un descripteur de fichier que nous nommerons `pin` et un autre que nous nommerons `pout`.
2. Le shell fork une première fois.
3. Dans le processus fils, il ferme `pin` et remplace la sortie standard par `pout`.
4. Toujours dans ce processus fils, le shell exécute ensuite `ls`.
5. Le shell fork une seconde fois,.
6. Dans ce nouveau processus fils, il ferme `pout` et remplace l'entrée standard par `pin`.
7. Toujours dans ce second fils, il exécute `more`.
8. Le shell peut désormais fermer ses copies de `pin` et `pout`.

FIG. 3.1 – Création d'un pipe

Les pipes sont un moyen de communication à sens unique (même s'il est possible de créer deux pipes pour simuler une communication bidirectionnelle), et ne peuvent pas être créés entre deux processus déjà existant.

Au niveau de l'implémentation, le système possède un tampon par pipe (d'une taille qui peut être fixe ou configurable suivant le système). Tout ce qui est écrit est mis dans ce tampon jusqu'à ce que le tampon soit plein. Lorsque le tampon est plein, une demande d'écriture peut soit bloquer en attendant que des données soient lues pour faire de la place, soit renvoyer une erreur (EAGAIN en général, pour "try again later").

Les tubes nommés Les tubes nommés ont comme but de supprimer cette limite, et de permettre à des processus qui n'ont aucun lien de parenté de communiquer entre eux. Ils sont présents dans le système de fichiers, sous l'aspect d'un fichier. Lorsque deux processus ont ouverts le fichier, un en lecture et l'autre en écriture, tout ce que le deuxième écrit est lu par le premier.

Comme toujours, de nombreuses questions se posent : que se passe-t-il si deux processus essaient de lire ? Si un processus essaye d'écrire sans que personne ne lise ?

Une utilisation typique de ces tubes nommés est celle faite par le logiciel `xmms`, un lecteur multimédia pour Unix. Ce programme créé un tube nommé dans le répertoire `/tmp`, sur lequel il attend des commandes simples du type "mettre en pause", "passer au morceau suivant", et ainsi de suite. Ainsi, l'utilisateur peut contrôler sa liste de lecture avec un programme à lui, ou avec des raccourcis claviers, ou encore via le réseau.

Les sockets Unix Les sockets Unix sont une autre forme de communication, similaires aux tubes nommés, mais tout de même sensiblement différents. Les principales différence est que les sockets Unix sont bidirectionnelles (les processus peuvent communiquer dans les deux sens, alors qu'avec les tubes Unix, un processus spécifique ne fait que lire pendant que l'autre ne fait qu'écrire), et que les sockets Unix sont prévues dans une optique clients-serveur, avec un serveur unique qui contrôle la socket, et de nombreux clients qui peuvent communiquer avec lui.

Les sockets Unix sont typiquement utilisés par un serveur de base de données, ou par un serveur X, pour répondre à des requêtes en local de multiples clients.

La mémoire partagée

Le mode de communication le plus rapide entre deux processus est d'avoir une zone de mémoire partagée entre les deux processus. Ainsi, un processus peut lire directement ce qu'un autre processus a écrit, sans aucune

copie inutile des données, et sans devoir effectuer systématiquement un appel système afin de lire ou d'écrire des données.

La manière d'implémenter la mémoire partagée dépend fortement du fonctionnement de la mémoire virtuelle, qui sera étudiée en détails par la suite.

La mémoire partagée est typiquement utilisée lorsque la rapidité de communication est fondamentale, par exemple entre le serveur X et les programmes graphiques.

Il est à noter qu'il existe des différences fondamentales entre deux processus ayant de la mémoire partagée et deux threads du même processus : tout d'abord, seule partie de la mémoire est partagée, et ensuite, les deux processus ne partagent rien d'autres (ils n'ont pas les mêmes descripteurs de fichiers, l'un peut être tué sans que l'autre ne soit affecté, ...).

Les messages

Le dernier mode de communication est un mode de communication structuré, à base de messages. Ce mode de communication est en général utilisé dans les systèmes à base de micro-noyau où la communication entre les différents serveurs est une composante fondamentale.

Dans les systèmes usuels, la notion de message structurée est en général implémentée dans une bibliothèque en espace utilisateur, au dessus d'un moyen d'IPC plus simple fourni lui par le système d'exploitation.

Plus loin nous verrons rapidement la structure des messages sur un micro-noyau comme Mach.

3.1.3 Les appels de procédures distantes (RPC)

Les RPCs (*Remote Procedure Call*) sont un mode de communication plus haut niveau entre deux processus. Ils consistent à envoyer non pas des données brutes, ni même des messages structurés, mais à effectuer des appels de procédures (ou fonctions) depuis un processus vers un autre.

Un RPC correspond en général à deux IPCs : un message est envoyé par le "client" vers le "serveur" en contenant l'identifiant de la procédure à appeler et les paramètres (encodés selon un protocole précis), puis un message est envoyé de la part du serveur vers le client avec la valeur de retour de la fonction, ou des informations en cas d'éventuelle erreur.

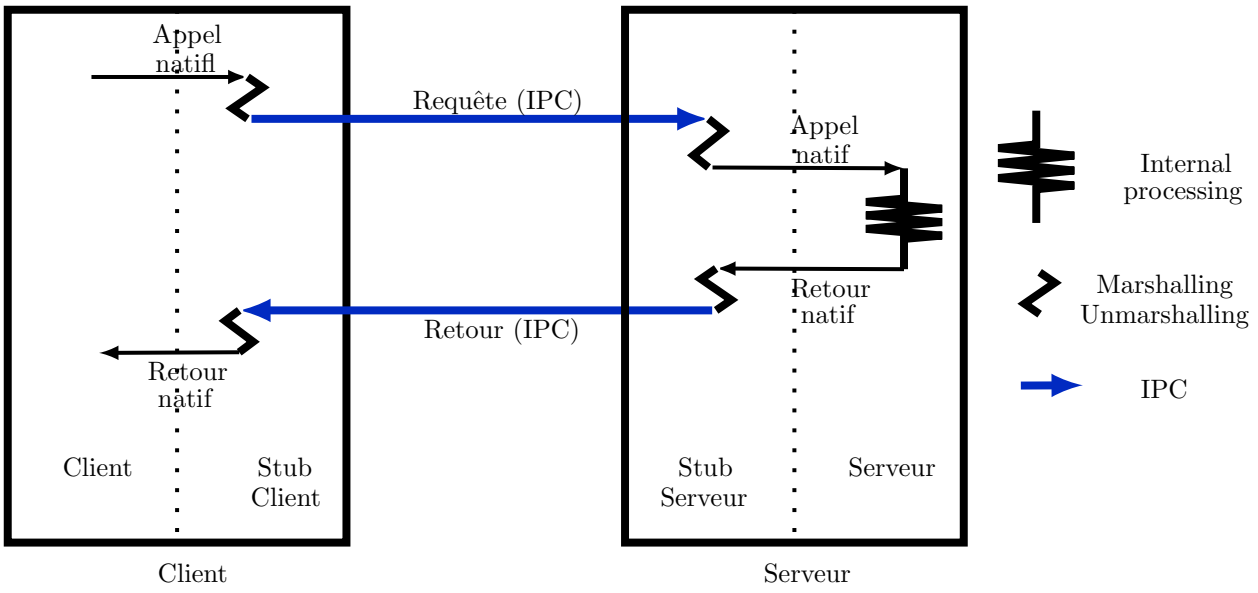

FIG. 3.2 – Schéma d'un RPC

Bien évidemment les paramètres et les noms de fonctions doivent être encodés d'une manière bien particulière afin d'être compris par les deux processus. Il est donc nécessaire de convertir les paramètres au format natif vers un format standardisé dans un sens, et du format standardisé vers le format natif dans l'autre sens. Le format standardisé peut être très différent du format natif, par exemple dans le cadre de l'XML-RPC, il s'agit d'un document XML, avec les données binaires encodées en base 64.

Afin d'éviter aux programmeurs la tâche fastidieuse d'encoder et décoder les paramètres systématiquement, une couche intermédiaire nommée *stub* est en général créée automatiquement (via un générateur de code ou

de manière complètement dynamique). Le client effectue lui un appel natif vers le stub correspondant. Le stub encode les paramètres et se préoccupe de la communication elle-même, vers le stub du serveur. Le stub du serveur décode les paramètres et appelle la fonction de manière native. Le programmeur lui a donc l'impression d'utiliser des appels de fonction locaux.

Hormis les systèmes à base de micro-noyau, les RPCs sont souvent utilisés pour faire communiquer des applications à un haut niveau, comme l'API COM de Windows, KParts de KDE ou Bonobo de Gnome.

Des exemples de RPCs réseau sont l'XML-RPC (et sa sur-couche SOAP) pour les services Web, CORBA pour des programmes distribués, ou Sun RPC pour des services comme NFS (système de fichiers réseau).

3.2 Synchronisation et mutex

3.2.1 Nécessité de la synchronisation

Nous avons déjà vu le problème dans le cadre des threads, avec le compteur d'octets émis sur le réseau. Un autre exemple peut être donné avec un spooler d'impression qui attend qu'on dépose les fichiers dans un répertoire donné et les imprime dés qu'il le peut. Le risque est alors présent que le spooler commence à imprimer alors que le fichier n'est pas encore complètement écrit, ce qui peut se traduire par un saut de page en plein milieu du texte.

Ce genre de situations se présente souvent dés lors que plusieurs processus interagissent, et qu'un ordonnanceur préemptif peut interrompre l'un pour basculer à l'autre à n'importe quel moment, il est donc nécessaire de prévoir des mécanismes de synchronisation.

Ce problème et ses solutions sont présents dans la communication entre des processus différents comme entre des threads du même processus.

3.2.2 Implémentation de la synchronisation

La notion de section critique

La quasi totalité des problèmes de synchronisation se résument à la notion de section critique : une zone précise du code d'un programme (par exemple les 4 instructions qui servent à augmenter le compteur, ou la boucle qui écrit le contenu du fichier) ne doit pas être interrompue sous peine de créer des risques de problème.

La solution la plus simple serait d'interdire tout changement de processus pendant cette section critique, mais cette solution comporte de très nombreux désavantages : tout d'abord, en cas de bug, un processus pourrait rester éternellement sur le CPU. Mais il est aussi sub-optimal de laisser le CPU pendant inutilisé pendant que le programme écrit son fichier pour le spooler d'imprimante alors qu'un autre processus, qui n'a rien à voir avec l'imprimante, cherche à obtenir le CPU.

En réalité, la condition formulée plus haut est un peu trop forte : cette section critique peut être interrompue, et même pour donner le CPU à un autre processus impliqué (que le processus A soit entrain d'écrire un fichier pour le spooler n'empêche pas le spooler de continuer à imprimer le fichier donné par un autre processus). En réalité, ce qui ne doit jamais arrivé est d'avoir deux processus (ou deux threads) différents dans des sections critiques concernant la même ressource au même moment.

Le principe des mutex

La solution générale aux problèmes de synchronisation est une primitive nommée *mutex* pour *mutual exclusion*. Un mutex est un objet (dont nous verrons la nature par la suite) partagé entre deux processus (ou plus). Ce mutex peut avoir deux états : l'état libre et verrouillé.

Lorsqu'un mutex est à l'état libre, n'importe quel processus peut le faire passer à l'état verrouillé (on dit alors qu'il verrouille ou qu'il acquière le mutex). Lorsqu'il est à l'état verrouillé, il peut être déverrouillé par le processus qui l'avait verrouillé (on dit alors qu'il libère ou qu'il déverrouille le mutex).

Par contre, lorsqu'un mutex est déjà verrouillé par un processus (ou thread), aucun autre processus (ou thread) ne peut le verrouiller. Suivant l'implémentation (ou le choix du programmeur), l'opération renvoie une erreur, ou alors le processus est bloqué jusqu'à ce que le mutex soit de nouveau libre.

En entourant les sections critiques par un verrouillage du mutex avant d'y pénétrer, et un déverrouillage du mutex en la quittant, on peut implémenter la notion de section critique, et s'assurer qu'un seul processus soit dans sa section critique au même moment.

Les mutex avec des fichiers

Comme pour la communication, la solution la plus simple pour implémenter un mutex (mais pas la plus efficace) est d'utiliser un fichier.

Un processus verrouiller un mutex tente de créer un fichier. Si l'opération réussit, le mutex est pour lui. Il lui suffit de supprimer le fichier pour libérer le mutex. Par contre, si le fichier existait déjà (c'est à dire si un autre processus avait déjà le mutex), le système d'exploitation va répondre avec une erreur.

Bien sûr, l'opération de création ou de destruction de fichiers n'est pas atomique, et pourrait donc éventuellement elle-même contenir des *race conditions*. Mais ces opérations étant réalisées (en général) par le noyau lui-même, il peut facilement garantir qu'aucun appel à `create` ne soit traité au milieu d'un autre, ou prendre d'autres précautions (comme interdire de basculer d'un processus à l'autre pendant cette opération, qui est souvent presque instantanée puisque l'écriture sur le disque est réalisée de manière asynchrone).

Différentes implémentations des mutex

Un mutex peut être implémenté avec un simple entier : un 0 indique que le mutex est libre, un 1 qu'il est verrouillé. Mais le soucis initialement présent demeure : pour acquérir un mutex, il faut lire la valeur, puis si elle était bien à 0 écrire 1. Cependant, un changement de contexte peut avoir lieu entre ces deux instructions.

La première implémentation possible, comme pour les fichiers, consiste à utiliser un appel système, et à faire confiance au noyau. Celui-ci peut désactiver les interruptions pendant les quelques instructions nécessaires à gérer le mutex, ou utiliser une autre astuce. Ce mode de fonctionnement est le seul possible quand il s'agit de communiquer entre processus, mais dans le cas de threads il peut être sous-efficace puisqu'un appel système coûte cher.

Le deuxième solution utilise une instruction matérielle spéciale, en générale nommée TSL (pour *test and set lock*). Elle prend en paramètres un registre et une adresse mémoire. À la fin de son exéuction, l'ancien contenu de la zone mémoire est mis dans le registre, et la mémoire est mise soit à 1, soit à l'ancienne valeur du registre. Dans les deux cas, l'atomicité est garantie, ce qui permet d'éviter tout risque de conflit.

Il existe une troisième solution, inventée par un dénommé Peterson qui permet d'implémenter un mutex sans aucune assistance de la part du système d'exploitation ni du matériel. Cependant, le matériel fournit en général l'instruction TSL sous une forme ou sous une autre, et cette solution n'est donc pas nécessaire.

Mutex bloquant et non bloquant

Les solutions présentées permettent d'être sûr qu'un processus ne peut pas acquérir un mutex pendant qu'un autre le possède déjà. Mais que faire si le mutex est déjà verrouillé ?

La solution naïve consiste à boucler indéfiniment, jusqu'à ce que le mutex soit libéré. Cette solution, nommée un *spinlock* est très peu optimale en terme d'utilisation du CPU. Une optimisation consiste à faire un `yield` (aurpès du système ou de la bibliothèque de threads) si l'opération échoue après chaque échec. Mais même ainsi, on peut essayer d'acquérir le mutex un grand nombre de fois avant d'y arriver.

L'implémentation la plus optimale consiste à maintenir pour chaque mutex une liste de processus en attente. Lorsqu'un processus essaie de verrouiller un mutex déjà occupé, le processus est ajouté à celle file d'attente de ce mutex. Il est aussi retiré de la liste des processus en attente de CPU et mis dans la liste des processus endormis. Lorsque le mutex sera libéré, l'un des processus en attente sur le mutex sera réveillé (mis de nouveau dans la liste des processus R, ceux que le scheduler utilise).

Les mutex récursifs

La plupart des mutex sont dits "simples" : si le même processus (ou le même thread) tente de le verrouiller deux fois de suite, il se bloque lui-même. Cette situation peut être résolue par le programmeur, en s'assurant qu'il ne verrouille jamais deux fois le même mutex. Mais considérons par exemple un cache ayant une taille fixée. Lorsqu'on insère une nouvelle entrée dans le cache, on va peut-être devoir en enlever une autre pour faire de la place. Or, dans une implémentation simple, la fonction qui ajoute une entrée tout comme celle qui en enlève une vont verrouiller le même mutex : celui qui protège les données interne du cache.

La solution la plus élégante est de fournir des mutex récursifs, qui sont capables de réagir intelligemment à ce genre de situations. Ces mutex possèdent un compteur interne, et si une tentative de les verrouiller est effectué par le processus (ou le thread) qui a déjà ce mutex, il incrémente uniquement le compteur. Lorsque le processus libère le mutex, le compteur est décrémenté, et le mutex n'est réellement libéré que s'il atteint 0.

Les sémaphores

Les mutex permettent d'obtenir une synchronisation binaire : soit il est libre, soit il est verrouillé. Dans certains cas, par exemple si on souhaite synchroniser l'accès à une ressource présente en plusieurs quantités (comme les imprimantes sur un serveur d'impression en possédant plusieurs), il peut être utile d'autoriser de 1 à n processus dans une section critique au même moment.

Ce mécanisme s'appelle les sémaphores. Une sémaphore est un compteur qui possède deux opérations : up et down. L'opération up augmente la valeur du compteur, jusqu'à une valeur maximale prédéfinie. L'opération down diminue la valeur du compteur, ou bloque si le compteur était déjà à 0.

Une sémaphore est implémentable directement en modifiant légèrement la plupart des techniques d'implémentation d'un mutex, ou alors en utilisant un simple entier partagé entre tous les processus et lui-même protégé par un mutex.

3.3 Modèles d'IPC évolués

3.3.1 L'API System V

L'API System V d'IPC est disponible sur la plupart des systèmes Unix. Elle fournit trois services : des sémaphores, des files de messages, et de la mémoire partagée.

Les ressources sont identifiées par une clé (un entier sur 32-bits). Chaque ressource possède un propriétaire, et des droits (comme pour les fichiers). Un processus ne peut utiliser une ressource que s'il connaît sa clé et possède les droits suffisants dessus.

Les ressources de l'IPC System V sont globales au système et peuvent être manipulées par les commandes `ipcs` et `ipcrm`. Une commande `ipcs` depuis une session X donnera en général une liste de quelques zones de mémoire partagée, utilisées entre le serveur X et les clients X.

3.3.2 Les messages de Mach

Le micro-noyau Mach, comme la plupart des micro-noyaux, possède une API riche de gestion des messages. Loin de rentrer dans les détails, le paragraphe suivant expose rapidement les concepts de l'IPC sous Mach.

Port Le concept de base de l'IPC sur Mach est un port. Un port est un canal de communication unidirectionnel. Les ports ne sont jamais manipulés directement par les applications, mais uniquement les droits sur les ports.

Port right Un "port right" (droit sur un port) est ce qui permet de manipuler un port. Au niveau du noyau, un "port right" est un entier local au processus (le droit numéro 1 pour un processus n'a pas forcément quelque chose à voir avec le droit numéro 0 pour un autre).

Read right Un droit de lecture permet de lire les messages envoyés vers ce port. Un seul processus peut posséder un droit de lecture sur un port.

Write right Un droit en écriture permet d'envoyer des messages sur le port. Le processus qui possède le droit de lecture peut créer des droits en écriture sur ce port, et le donner à d'autres processus comme il le souhaite.

Message Les ports permettent d'envoyer des messages, qui sont structurés. Ils contiennent des types de données précis, comme des entiers, des chaînes de caractères, ou des droits sur des ports.

3.4 Les deadlocks

3.4.1 Présentation du problème

Un exemple de deadlock

Il est parfois nécessaire, à l'intérieur d'une même section critique, d'acquérir plusieurs mutex. Considérons par exemple une base de données, où l'on peut verrouiller un mutex par table pour éviter d'avoir des données désynchronisées. Cette base de données possède (entre autres) deux tables : une avec la liste des articles disponibles dans le magasin, et l'autre avec les commandes.

À la création d'une commande, on vérifie d'abord la présence des articles, puis on créé la commande et enfin on préserve les articles en utilisant l'identifiant de la commande précédemment créée. Un exemple de code simplifié est disponible sur la figure 3.3. Il est nécessaire de garder le mutex sur "articles" jusqu'à la fin pour éviter qu'une autre réservation effectuée entre temps ne consomme la dernière unité présente.

À la validation d'une commande, on récupère les articles concernés depuis la table commandes (qui contient, pour la postérité, la liste de tous les articles commandés). Ensuite, on confirme la réservation des articles et finalement on marque la commande comme validée. Un exemple est disponible sur la figure 3.4.

On constate que les mutex sont acquis dans un ordre contraire : dans un cas d'abord `articles` puis `commandes`, et dans l'autre cas d'abord `commandes` puis `articles`. Que se passe-t-il si jamais le code de `nouvelle_commande` est interrompu entre les deux acquisitions de mutex pour traiter la validation de la commande d'un autre client ? `valider_commande` va acquérir le mutex sur `commandes`, puis attendre que le mutex sur

```
nouvelle_commande(utilisateur, articles)
{
  verrouiller_table("articles");
  verifier_presence(articles);
  verrouiller_table("commandes");
  commande = creer_commande(utilisateur, articles);
  reserver_articles(commande);
  deverrouiller_table("commandes");
  deverrouiller_table("articles");
}
```

FIG. 3.3 – Création d'une nouvelle commande

```
valider_commande(commande)
{
  verrouiller_table("commandes");
  articles = recuperer_articles(commande);
  utilisateur = recuperer_utilisateur(commande);
  verrouiller_table("articles");
  confirmer_reservation(articles);
  confirmer_commande(commande);
  deverrouiller_table("articles");
  deverrouiller_table("commandes");
}
```

FIG. 3.4 – Validation d'une commande

`articles` soit libéré pour continuer. Tôt ou tard, `nouvelle_commande` va pouvoir continuer son exécution, jusqu'au moment où elle va tenter d'acquérir le mutex sur `commandes` qui est déjà possédé par `valider_commande`.

Chaque fonction va attendre que l'autre ait pu terminer son exécution pour continuer la sienne. Aucune ne pourra donc jamais avancer, le système est désormais bloquer. Nous avons un *deadlock*.

Les solutions pour cet exemple

Dans ce cas précis, il y a plusieurs solutions. La première consiste à utiliser un mutex unique, et non un par table. L'inconvénient est qu'une fonction qui affiche la liste des articles sera bloquée par une fonction qui génère une facture, alors que les deux auraient pu fonctionner avec chacune sa table.

Une autre solution est de considérer que les tables doivent toujours être verrouillées et déverrouillées dans un ordre précis. Par exemple, si on a besoin de `commandes` et de `articles`, on doit toujours verrouiller `commandes` d'abord.

3.4.2 Les conditions nécessaires à leur apparition

Pour qu'un deadlock puisse se produire, il a été démontré qu'il faut que les trois conditions suivantes soient remplies :

1. Exclusion mutuelle : une ressource (par exemple un mutex) ne peut être possédé que par un seul processus à la fois.
2. Requêtes successives : un processus qui possède déjà des ressources verrouillées pour lui peut en demander de nouvelles.
3. Absence de préemption sur les ressources : le système ne peut pas reprendre de force une ressource qui a été précédemment attribuée.

3.4.3 Les techniques de contournement

Détection des deadlocks

En utilisant des graphes de ressources et de processus, il est possible de détecter un deadlocks une fois que celui-ci à eu lieu (sous la forme d'un cycle dans le graphe). Le système peut alors, par exemple, détruire un processus impliqué, ou simplement en informer l'administrateur.

Il existe aussi des techniques (à base de matrices et de vecteurs de ressources) qui permettent de détecter à l'avance les risques de deadlocks, mais ces algorithmes nécessitent tous de connaître des informations assez précises sur ce que les processus vont faire dans le futur ; une information qui hélas n'est rarement disponible.

À cause des contraintes trop fortes qu'ils demandent, ces algorithmes ne sont que très peu utilisés en pratique, et ne seront donc pas détaillés ici. Ce qu'il faut savoir est qu'ils existent, lorsqu'il est nécessaire d'éviter tout risque, même au prix d'une plus grande complexité pour les programmeurs (par exemple, dans une centrale nucléaire ou dans un centre de contrôle d'une agence spatiale).

Prévention des deadlocks

Si détecter les deadlocks est difficile, peut-on les rendre impossible en brisant l'une des trois conditions ?

L'exclusion mutuelle est difficile à prévenir. On peut réduire son ampleur en essayant de diminuer la taille des sections critiques, en utilisant des techniques comme des files d'attente, ou des spoolers.

Il est possible d'interdire les requêtes successives en demandant à un processus (ou à un thread) de verrouiller tous les mutex dont il peut avoir besoin d'un coup, et de ne pas l'autoriser à en acquérir de nouveaux le temps qu'il ne les a pas libéré. Cette solution n'est pas optimale (si on a besoin d'un mutex pendant 100ms et d'un autre seulement pendant les 10ms finales, on va les accaparer tous les deux sur 100ms tout de même) et peu être assez complexe à programmer lorsqu'on utilise des composants ou des librairies provenant de plusieurs auteurs.

La non préemption peut parfois être résolue, par exemple si le programme supporte les transactions. Dans ce cas, il est possible de reprendre de force les ressources déjà accordées, tout en annulant la transaction en cours et en demandant au programme de la rejouer. Mais dans le cas général, cette solution n'est pas possible non plus.

3.5 Quelques problèmes classiques

3.5.1 Le dîner des philosophes

Présentation du problème

Cinq philosophes se retrouvent pour dîner. Au menu, il y a des spaghettis. Chaque philosophe alterne entre une phase où il pense, et une phase où il mange. La durée de ces phases est variable.

Mais il y a un problème dans ce tableau idyllique : les spaghettis sont tellement glissant que chaque philosophe nécessite deux fourchettes pour les manger. Or, il n'y a que 5 fourchettes, positionnées entre les assiettes des philosophes. Un philosophe ne peut manger que s'il acquiert la fourchette de gauche et celle de droite.

Une solution possible

La solution naïve consiste à considérer que chaque fourchette est un mutex, et un philosophe acquiert les deux mutex pour pouvoir manger. Mais cette solution est propice aux deadlocks : si chaque philosophe acquiert sa fourchette de gauche, et tente ensuite d'avoir celle de droite, personne ne pourra plus jamais manger.

Une solution consiste à ajouter un mutex global. Un philosophe souhaitant manger acquiert d'abord le mutex global, puis les deux fourchettes, et enfin libère le mutex global. Pour les fourchettes, un mutex non bloquant est utilisé, si une fourchette, n'est pas disponible, on réessaye plus tard (en libérant l'éventuelle fourchette déjà acquise). Cette solution est inefficace car utilisant une boucle active, mais fonctionne.

Il existe des solutions capable d'éviter ce désagrément en utilisant un mutex supplémentaire par philosophe. La littérature est copieuse à ce sujet, donc cette solution ne sera pas détaillée ici.

3.5.2 Le modèle producteur — consommateur

Présentation du problème

Dans ce modèle, un producteur produit des données (par exemple des fichiers à imprimer) et un consommateur utilise ces données (par exemple un spooler d'impression). Il y a une file d'attente d'une taille maximale fixée dans laquelle le producteur peut écrire ses données. Si le consommateur n'a pas de données à traiter, il s'endort. Si le producteur n'a plus de place disponible dans la file, il s'endort aussi.

Il existe des modèles un peu plus complexes avec plusieurs consommateurs et plusieurs producteurs.

Une solution possible

La solution classique à ce problème consiste à utiliser un mutex global, ainsi que deux sémaphores : un qui compte les slots libres dans la queue, et un qui compte les slots utilisés. Le mutex sert à protéger l'insertion ou la suppression de la file d'attente. Les deux sémaphores permettent de bloquer le producteur ou le consommateur.

```
while (true)
  {
     decrease(used);
     acquire(mutex);
     item = pop_item();
     release(mutex);
     increase(empty);
     process_item(item);
  }
```

FIG. 3.5 – Exemple de consommateur

```
while (true)
  {
     item = produce_item();
     decrease(empty);
     acquire(mutex);
     push_item(item);
     release(mutex);
     increase(used);
  }
```

FIG. 3.6 – Exemple de producteur

Un exemple d'implémentation est donné dans la figure 3.5 et la figure 3.6.

Au départ, le sémaphore qui compte les slots libres est au maximum, celui qui compte les slots utilisés au minimum. Le consommateur va donc se bloquer sur sa tentative de décrémenter. Le producteur va lui produire un élément, puis décrémenter le nombre de slots libres, afin de réserver un slot. Il ajoute ensuite l'objet dans la file d'attente, à l'intérieur de sa section critique, puis incrémente le sémaphore qui contient le nombre de slots utilisés pour débloquer le consommateur.

Si la file est pleine, c'est le producteur qui va être bloqué juste après avoir produit un élément, et qui sera débloqué une fois que le consommateur en aura consommé un.

Informations légales

La version PDF et le code source LaTeX sont disponibles sur `http://insia.pilotsystems.net`.