

# Processus : ordonnancement et synchronisation

par Michel RIVEILL - Professeur à l'INP Grenoble

(texte initial de Sacha Krakowiak, professeur à l'Université Joseph Fourier)

<b>1. Introduction</b> .....	<b>1</b>
<b>2. Processus et synchronisation</b> .....	<b>4</b>
2.1. Notion de processus .....	4
2.2. Exclusion mutuelle.....	5
2.3. Synchronisation des processus.....	6
2.4. Mise en œuvre des processus .....	10
<b>3. Allocation des ressources</b> .....	<b>12</b>
3.1. Objectifs d'une politique d'allocation.....	12
3.2. Allocation de processeur.....	13

## 1. Introduction

Le **système d'exploitation** d'un ordinateur, ou d'un ensemble d'ordinateurs connectés en réseau, est un ensemble de programmes qui remplissent deux grandes fonctions.

- Allouer les ressources matérielles et logicielles pour satisfaire les besoins des programmes d'application.
- Présenter aux applications une interface mieux adaptée à leurs besoins que celle directement fournie par le matériel ; on peut dire en ce sens qu'un système d'exploitation réalise une machine virtuelle.

Il est plus exact de parler d'un ensemble d'interfaces. En effet, les utilisateurs et les applications peuvent généralement interagir avec différentes couches du système d'exploitation. Au niveau le plus haut, l'interface se présente comme un langage de commande symbolique ou graphique ; en fait, l'interprète de ce langage est en général en dehors du système d'exploitation proprement dit et peut donc être remplacé ou modifié. A un niveau plus bas se situe l'interface principale, constituée d'un ensemble d'«appels systèmes», qui définissent la machine virtuelle et permettent d'utiliser ses différentes entités : processus, fichiers, messages, etc. Cette interface peut être utilisée par les applications, soit directement, soit le plus souvent au travers de bibliothèques spécialisées associées à un langage de programmation. Enfin, certains systèmes d'exploitation (systèmes à micro-noyau, systèmes à objets) ont une structure modulaire qui permet

d'accéder à plusieurs niveaux différents de fonctions plus ou moins proches de la machine physique.

L'évolution des systèmes d'exploitation est gouvernée à la fois par l'évolution des architectures matérielles et par l'évolution des besoins des applications. Les faits marquants des années 90 peuvent être résumés comme suit.

- Transition progressive des systèmes dits "propriétaires" (associés à une architecture particulière de machines, fournissant une interface spécifique) vers des systèmes "ouverts" (portables sur une gamme étendue de machines et présentant des interfaces standard, voire normalisées, en vue de faciliter le transport et l'interconnexion des applications).

- Passage d'une informatique centralisée à une informatique distribuée, dans laquelle le système d'exploitation doit prendre en compte la répartition des ressources matérielles, des données et des applications, tout en dissimulant cette répartition aux utilisateurs.

- Globalisation des échanges, par l'interconnexion généralisée des réseaux et l'informatique "nomade" (connexion d'ordinateurs mobiles) ; forte interaction de l'informatique et des télécommunications ; développement des applications coopératives et des communications multimédia.

Bien qu'il n'existe pas de structure standard s'appliquant à tout système d'exploitation, on peut en général identifier les principaux composants de base suivants.

- Le **noyau** a pour fonction l'allocation des processeurs, le traitement des interruptions et la gestion des horloges, la gestion des processus, et la supervision des entrées-sorties (dont l'exécution est confiée à des composants appelés **pilotes** (*drivers*), spécifiques de chaque classe d'organes d'entrée-sortie ou de communication).

- Le **gérant de mémoire** a en charge la gestion de la mémoire virtuelle et l'allocation de la mémoire physique.

- Le **système de gestion de fichiers** gère la désignation et la conservation de l'information dans les fichiers ; il fait appel au gérant de mémoire et aux pilotes des unités de mémoire secondaire.

Ces composants de base sont utilisés par divers sous-systèmes : interprètes de langage de commande, gérants de fenêtres, gérants de réseau, bibliothèques d'exécution pour des langages de programmation. Ces sous-systèmes ne font pas partie du système d'exploitation proprement dit, mais sont indispensables à son utilisation effective.

Les systèmes d'exploitation traditionnels sont monolithiques : les composants de base ci-dessus ne sont pas des modules identifiés et séparables, mais sont étroitement intégrés (Fig. 1a) dans une structure unique (le "noyau"). Cette organisation permet d'atteindre des hautes performances en accélérant les communications internes, mais elle rend difficiles l'évolution et l'adaptation des systèmes. D'autre part, des classes spécialisées d'applications n'ont pas nécessairement besoin de toutes les fonctions d'un système : pour commander des procédés en temps réel, il faut essentiellement un gérant de processus et des pilotes de périphériques et de réseau ; les besoins en gestion de fichiers ou en mémoire virtuelle sont inexistantes ou rudimentaires.

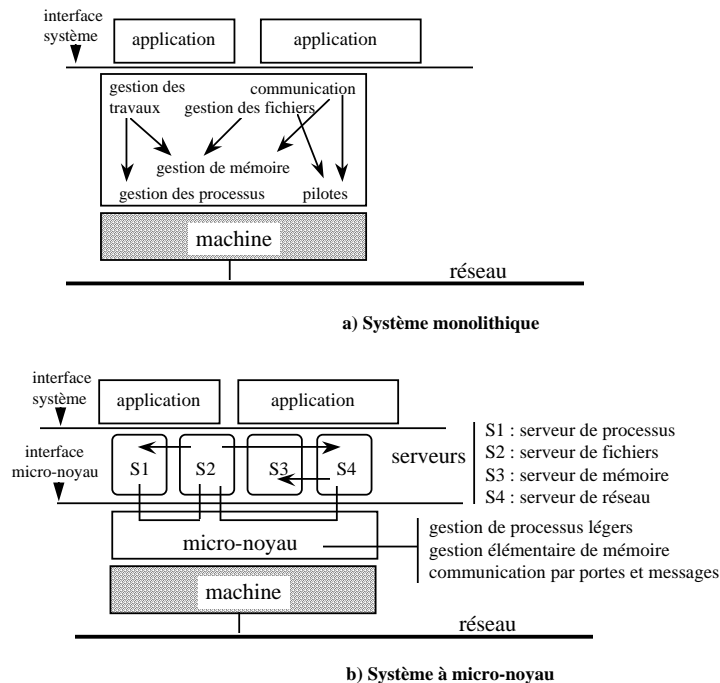


Figure 1 - Structures schématiques de systèmes d'exploitation

Ces considérations ont conduit à définir une nouvelle architecture de systèmes, dans laquelle un "micro-noyau" concentre les fonctions élémentaires de gestion du processeur, de la mémoire et des communications, les autres composants (gérant de processus, de mémoire virtuelle, de fichiers, de réseaux, etc) étant réalisés par des serveurs inter-communicants utilisant les fonctions du micro-noyau (Fig. 1b). Un système traditionnel tel qu'Unix peut alors être réalisé par un ensemble de serveurs. Un système spécialisé (gérant de communication, système serveur d'un terminal X) ne comporte que les serveurs strictement nécessaires à ses fonctions. La modularité des systèmes à micro-noyau les rend particulièrement intéressants pour la construction des systèmes répartis.

Les principaux concepts des systèmes d'exploitation ont été dégagés dans les années 1965-75. De cette époque datent les principales notions relatives aux processus et à la synchronisation, aux fichiers, à la désignation, à la liaison et à la protection des informations, à l'allocation des ressources, ainsi que le début de l'étude quantitative des systèmes par la modélisation mathématique et la simulation. Ces concepts sont toujours largement valides. Les principales nouveautés viennent du développement des systèmes répartis, ainsi que des progrès dans la structuration du logiciel, qui visent à augmenter sa modularité et sa réutilisabilité, notamment par l'usage de techniques à base d'objets. La répartition a des influences multiples : incertitude sur l'état global due à l'absence de

mémoire commune et d'horloge globale ; nécessité de prendre en compte des probabilités élevées de défaillance d'une partie du système ; exigences accrues pour la sécurité des informations ; adaptation des algorithmes à la croissance du système. L'évolution du matériel a également modifié l'importance relative de certains problèmes. C'est ainsi que l'augmentation considérable des tailles de mémoire centrale (d'un facteur de 100 en 15 ans) a réduit l'importance des algorithmes de pagination, mais a mis l'accent sur les problèmes de cohérence d'informations en copies multiples en raison de l'emploi généralisé de caches logiciels.

Selon le point de vue auquel on se place, un système d'exploitation peut être considéré comme un gérant d'activités, un gérant d'information, ou un gérant de ressources. Cet aspect fonctionnel guide le plan de la présentation.

La gestion d'activités parallèles et coopérantes, étroitement liée à l'allocation des processeurs, est la fonction centrale du noyau d'un système. Les principes en sont présentés au chapitre 2.

La gestion des informations (fichiers des utilisateurs ou informations internes au système) nécessite de définir les modalités de leur désignation et de leur conservation ; les concepts correspondants font l'objet du chapitre 3.

Un système d'exploitation peut enfin être considéré comme le gérant des ressources d'une machine. Les principes de l'allocation de ressources, et leur application à la gestion des processeurs et de la mémoire, sont développés au chapitre 4.

L'accès d'une communauté d'utilisateurs à des informations et à des ressources partagées a pour corollaire la nécessité d'assurer la sécurité des informations et le bon usage des ressources, face aux risques d'erreur ou de malveillance. C'est le rôle des mécanismes de protection et d'authentification, présentés au chapitre 5.

Enfin, les ordinateurs sont de plus en plus fréquemment utilisés en réseau, pour exécuter des applications réparties. Les fonctions liées à la communication et à la répartition font l'objet du chapitre 6. L'état actuel des systèmes d'exploitation disponibles et les tendances de l'évolution sont résumés en conclusion (chapitre 7).

## 2. Processus et synchronisation

### 2.1. Notion de processus

Un **processus** est l'entité dynamique qui représente l'exécution d'un programme sur un processeur. Le déroulement d'un processus est une suite d'actions élémentaires (l'exécution des instructions du processeur). Le contexte d'exécution (ou simplement le contexte) d'un processus est l'ensemble des ressources (éléments matériels et information) accessibles à ce processus pendant son exécution. On distingue le contexte privé (propre au processus) et le contexte global (partagé avec d'autres processus). Deux processus ne peuvent interagir que si leur contexte a une partie commune.

Chaque action d'un processus peut modifier l'état de son contexte. Cet état n'est défini qu'en une suite discrète de points observables ; la suite des états en ces points constitue l'histoire (ou encore la trace temporelle) du processus.

Considérons maintenant deux processus (soit  $p$  et  $q$ ). Notons  $début(p)$  et  $fin(p)$  les instants de début et de fin d'exécution de  $p$ , et de même pour  $q$ . Si on a la relation :

$$\text{fin}(p) > \text{début}(q) \text{ et } \text{fin}(q) > \text{début}(p),$$

alors on dit que  $p$  et  $q$  s'exécutent en parallèle. Si, à un instant donné, on peut observer l'exécution simultanée d'une action de  $p$  et d'une action de  $q$ , il y a parallélisme réel. Si on ne peut observer qu'une action à la fois, il y a pseudo-parallélisme. Le parallélisme réel nécessite autant de processeurs physiques que de processus parallèles. Le pseudo-parallélisme est un moyen de simuler le parallélisme réel sur un processeur unique, en entrelaçant les traces temporelles des différents processus. L'exploitation des ordinateurs en temps partagé (*time-sharing*) s'appuie sur ce principe.

Les relations entre processus peuvent prendre deux formes :

- **Compétition** pour l'usage d'une ressource partagée.
- **Coopération** pour l'exécution coordonnée d'une tâche commune.

L'exemple le plus courant de compétition est celui de l'allocation de processeur. Une politique d'allocation (voir 4.2) attribue le processeur, par tranches de temps successives, aux processus qui en ont besoin. Un exemple simple de coopération est celui de la transmission de messages entre deux processus à travers une zone de mémoire commune (la boîte aux lettres) : si la boîte aux lettres est vide, le processus destinataire doit attendre que le processus émetteur y dépose un message ; si la boîte aux lettres est pleine (c'est-à-dire si on ne peut plus y déposer de message sans détruire un de ceux qui s'y trouvent), le processus émetteur doit attendre que le récepteur retire un message. Dans une situation intermédiaire où la boîte aux lettres n'est ni vide ni entièrement remplie, les deux processus peuvent fonctionner en parallèle à condition de ne pas accéder simultanément au même message. Ce problème est examiné en détail en 2.3.

On appelle **synchronisation** la mise en œuvre des relations entre processus en vue de respecter les contraintes de coopération. L'exemple ci-dessus montre que la synchronisation nécessite de "faire attendre" un processus jusqu'à ce qu'une condition de passage soit réalisée. Cela peut être obtenu de deux manières :

- par **attente active** : on fait boucler le processus en attendant que la condition soit réalisée,
- par **blocage** : on fait passer le processus dans un état dit bloqué, dans lequel il cesse de progresser ; lorsque la condition de passage est réalisée, le processus est explicitement débloqué (ou réveillé), et reprend son exécution.

Pour l'étude de la synchronisation, nous considérons l'exécution des processus d'un point de vue purement logique, sans tenir compte des vitesses réelles d'exécution. De ce point de vue, il importe peu que l'exécution se fasse en parallélisme réel ou en pseudo-parallélisme. Nous séparons ainsi les relations de synchronisation de l'allocation des processeurs, sur laquelle nous revenons en 2.4 et 4.2.

## 2.2. Exclusion mutuelle

L'**exclusion mutuelle** est la forme la plus simple de synchronisation entre processus. Deux processus sont en exclusion mutuelle, pour l'exécution d'une séquence appelée **section critique** dans le programme de chacun d'eux, si à tout instant un seul des processus au plus peut se trouver dans sa section critique. Par exemple, la modification concurrente d'informations partagées ne peut se faire qu'en exclusion mutuelle, sous peine d'aboutir à un état incohérent.

Un **sémaphore** est un mécanisme de synchronisation qui fonctionne comme une barrière, bloquante dans certaines conditions. Un sémaphore  $s$  comporte un compteur  $n(s)$  et une file d'attente de processus  $f(s)$ . Son fonctionnement est le suivant :

Initialement (à la création de  $s$ ), le compteur  $n(s)$  reçoit une valeur  $n0(s)$ ,  $n0(s) \geq 0$ , et la file  $f(s)$  est vide.

Les processus qui partagent  $s$  peuvent effectuer deux opérations notées  $P(s)$  et  $V(s)$  (d'autres notations sont rencontrées, telles que *up* ou *wait* pour  $P$ , *down* ou *signal* pour  $V$ ). L'effet de ces opérations est le suivant :

$P(s) : \quad n(s) := n(s) - 1$ $\text{if } n(s) < 0 \text{ then}$ $\quad \text{état}(p) := \text{bloqué}$ $\quad \text{entrer}(p, f(s))$ $\text{endif}$	$V(s) : \quad n(s) := n(s) + 1$ $\text{if } n(s) \leq 0 \text{ then}$ $\quad \text{sortir}(q, f(s))$ $\quad \text{état}(q) := \text{actif}$ $\text{endif}$
--	--

Ces opérations sont réalisées de manière atomique. Leur mise en œuvre nécessite donc une technique d'exclusion mutuelle élémentaire (voir 2.4).

Le fonctionnement d'un sémaphore est analogue à celui d'un portillon automatique actionné par des jetons. Tant qu'il y a des jetons d'avance dans l'appareil, le portillon laisse passer les personnes qui se présentent. Quand il n'y a pas de jetons, le portillon se bloque, et une file d'attente se forme à l'entrée. L'opération  $P$  est une tentative de franchissement, l'opération  $V$  le dépôt d'un jeton. Le compteur  $n(s)$  compte le nombre de jetons présents, et  $n0(s)$  est la provision initiale ; si  $n(s) < 0$ , alors  $|n(s)|$  est la taille de la file  $f(s)$ .

L'exclusion mutuelle se programme alors simplement comme suit, en utilisant un sémaphore (soit *mutex*), avec  $n0(\text{mutex}) = 1$ .

```

...
P(mutex)
{section critique}
V(mutex)
...

```

Il est possible de vérifier que ce mécanisme garantit l'absence d'interblocage. Si la file d'attente  $f(s)$  est gérée en "premier entré, premier servi", la privation est également évitée. On doit supposer que tout processus sort de section critique au bout d'un temps fini, ce qui peut être garanti, même en cas de panne, au moyen d'une horloge de garde.

## 2.3. Synchronisation des processus

### Synchronisation par sémaphores

Nous illustrons la synchronisation par sémaphores au moyen d'un exemple de synchronisation représentatif des problèmes de communication. Un processus (le producteur) transmet une séquence de messages à un autre processus (le consommateur), à travers un tampon (ou boîte aux lettres) pouvant contenir un nombre fixe  $N$  de messages. Les conditions de synchronisation sont les suivantes :

- Initialement, le tampon est vide (toutes ses cases sont libres).
- Un message ne peut être déposé que dans une case libre.
- Un message déposé est lu une fois et une seule ; la place qu'il occupe n'est libérée qu'après qu'il a été lu.

Un moyen d'assurer la synchronisation consiste à bloquer un processus qui tente d'exécuter une opération "impossible" (retirer un message depuis un tampon vide, ou déposer un message alors qu'il n'y a pas de case libre). Le programme suivant assure cette synchronisation.

```
semaphore nplein init 0, nvide init N;
tête:=0 ; queue:=0;
```

<pre>Producteur : produire (m)               P(nvide)               T[tête]:=m               tête:=tête+1 mod N               V(nplein)</pre>	<pre>Consommateur : P(nplein)                 m:=T[queue]                 queue:=queue+1 mod N                 V(nvide)                 consommer(m)</pre>
---	--

Les sémaphores *nplein* et *nvide* fonctionnent ici comme des compteurs, qui enregistrent respectivement le nombre de cases pleines et vides. Le tampon est représenté par un tableau de *N* cases, numérotées de 0 à *N*-1 ; le pointeur *tête* désigne le prochain message à retirer, le pointeur *queue* la prochaine case vide à remplir. On peut montrer que le schéma ci-dessus garantit l'absence de blocage et l'exclusion mutuelle pour l'accès aux messages. Il s'étend facilement à un nombre quelconque de producteurs et de consommateurs : il suffit de garantir l'exclusion mutuelle des producteurs entre eux et des consommateurs entre eux. Il est également facile de l'étendre au cas où les cases des tampons de communication sont allouées dynamiquement. Ce schéma est à la base des techniques usuelles de communication entre processus par boîtes aux lettres, portes ou files de messages.

#### Autres mécanismes de synchronisation

Les sémaphores sont un outil simple et efficace pour assurer la synchronisation. Ils présentent néanmoins des limitations qui expliquent que d'autres mécanismes aient été élaborés.

- Les sémaphores sont un outil de bas niveau sémantique. Les constructions qui les utilisent sont difficiles à comprendre et à modifier dès que le problème est un peu complexe.
- Les sémaphores donnent lieu à des constructions non structurées ; il n'y a pas de règles de portée ou de parenthésage qui pourraient aider à la construction de schémas standard. Il est difficile de prouver la validité d'un schéma de synchronisation utilisant des sémaphores.

Ces limitations sont analogues à celles de la construction *go to* en programmation séquentielle.

#### Moniteurs

Le mécanisme des moniteurs vise à fournir un mode d'expression de la synchronisation plus structuré que les sémaphores, en vue de faciliter la compréhension et l'écriture des schémas de synchronisation, de donner à ces schémas une forme synthétique, et de permettre la construction de preuves.

Un **moniteur** est un module de programme, comportant des variables d'état internes et des procédures (ou points d'entrée) accessibles depuis l'extérieur. Un moniteur est une structure partagée entre plusieurs processus, qui se synchronisent en appelant ses points d'entrée ; les variables d'état ne sont pas directement accessibles. Les procédures

du moniteur contiennent des opérations qui permettent de bloquer ou de réveiller les processus qui utilisent le moniteur. Une procédure d'initialisation est exécutée une seule fois lors de la création du moniteur.

Les constructions de synchronisation s'expriment à l'aide de conditions. Une condition *c* est déclarée comme une variable, mais ne peut être manipulée qu'au moyen de trois primitives (*p* désigne le processus appelant) :

```
c.wait      : bloque le processus p, et le place dans la situation "en attente de c".
c.empty     : délivre une valeur booléenne (vrai si aucun processus n'est en attente
              de c, faux sinon)
c.signal    : if ¬c.empty then {réveiller un des processus en attente de c} endif.
```

Les variables d'état du moniteur doivent être manipulées en exclusion mutuelle pour garantir leur cohérence. Lorsqu'un processus *p* réveille un processus *q* par l'exécution de *signal*, le processus *p* doit donc se bloquer pour assurer cette exclusion mutuelle, jusqu'au blocage ou à la sortie de *q*. Un processus qui est ainsi temporairement bloqué après avoir exécuté *signal* bénéficie d'une priorité pour la reprise de son exécution, en vue d'éviter un blocage indéfini.

Nous illustrons l'utilisation des moniteurs par le schéma du producteur et du consommateur. Ces deux processus partagent un moniteur *tampon*, qui représente la ressource commune constituée par le tampon de communication et exporte deux procédures *déposer* et *retirer*. Le programme des processus est donné ci-après.

<pre>processus producteur ... produire(message_émis) tampon.déposer(message_émis) ...</pre>	<pre>processus consommateur ... consommer(message_reçu) tampon.retirer(message_reçu) ...</pre>
---	--

Le programme du moniteur *tampon* est donné ci-après.

```
tampon : monitor
var n : 0..N ; non_plein, non_vide : condition ;
{déclaration des procédures entrer et sortir}
```

<pre>procédure déposer(m:message)   if n=N then     non_plein.wait   endif ;   n:=n+1 ;   entrer(m) ;   non_vide.signal end déposer</pre>	<pre>procédure retirer(m:message)   if n=0 then     non_vide.wait   endif ;   sortir(m) ;   n:=n-1 ;   non_plein.signal end retirer</pre>
---	---

```
initialisation : n:=0
end tampon ;
```

On notera, sur l'exemple ci-dessus, le caractère synthétique de l'expression de la synchronisation, ainsi que la modularité, qui permet de réutiliser de manière générique le schéma de synchronisation défini par le moniteur.

Les moniteurs ont une puissance d'expression équivalente à celle des sémaphores : chacun de ces mécanismes peut être programmé en utilisant l'autre.

### Autres mécanismes de synchronisation : messages, verrous, événements

Les opérations de communication par boîte aux lettres décrites plus haut peuvent elles-mêmes être utilisées comme mécanismes élémentaires de synchronisation. On définit pour cela des objets de communication primitifs appelés **portes** qui jouent le rôle des boîtes aux lettres. Une porte fonctionne donc comme une file de messages. On peut associer à une porte un processus récepteur, qui peut retirer les messages envoyés à la porte. On peut ainsi programmer un service par un processus serveur cyclique associé à une porte. Les demandeurs du service envoient leurs requêtes à la porte ; le serveur retire les messages et traite successivement les requêtes. Une requête peut contenir l'identité d'une porte sur laquelle le processus demandeur attend le résultat de la requête. La puissance d'expression de ce mécanisme équivaut à celle des deux précédents.

L'accès de plusieurs processus à des données partagées pose des problèmes de synchronisation que l'on peut résoudre par l'un des mécanismes précédents. Il est également possible d'utiliser un mécanisme spécifique, les **verrous**. Soit une information  $d$  (variable, zone de mémoire, fichier, etc) partagée par plusieurs processus. On définit les primitives suivantes :

verrouiller-exclusif( $d$ )  
verrouiller-partagé( $d$ )  
déverrouiller( $d$ )

Un processus qui doit modifier une donnée  $d$  doit encadrer la séquence de modification par *verrouiller-exclusif( $d$ )* et *déverrouiller( $d$ )*. Un processus qui doit consulter une donnée  $d$  doit encadrer la séquence de consultation par *verrouiller-partagé( $d$ )* et *déverrouiller( $d$ )*. Le verrouillage exclusif d'une donnée par un processus interdit tout autre verrouillage, exclusif ou partagé. Le verrouillage partagé d'une donnée par un processus interdit tout verrouillage exclusif de cette donnée, mais autorise son verrouillage partagé par d'autres processus. L'utilisation des verrous introduit le risque d'interblocage (cf. 4.1).

Les **événements** sont un mécanisme de synchronisation analogue aux interruptions. Un événement est un signal asynchrone, généralement non mémorisé, envoyé par un processus à un ou plusieurs autres, et qui provoque une réaction immédiate (exécution d'une séquence prédéfinie) chez le ou les destinataires. Un événement peut éventuellement être masqué et ne provoque alors aucune réaction. Les signaux du système Unix sont un exemple d'événements ; ils servent surtout à traiter des conditions exceptionnelles (forcer un processus à réagir à une erreur) ou à détruire le processus destinataire.

### Conclusion sur les mécanismes de synchronisation

Il existe une grande variété de mécanismes de synchronisation, dont la puissance d'expression est équivalente, et qui diffèrent par leurs conditions d'utilisation.

Les sémaphores sont des mécanismes de bas niveau, généralement utilisés à l'intérieur des noyaux de systèmes. Pour des séquences brèves d'exclusion mutuelle, les multiprocesseurs peuvent en outre utiliser l'attente active.

Les messages et les portes sont également utilisés à l'intérieur des systèmes. Leur intérêt est d'être aisément transposables à un système réparti, puisque les messages sont l'outil primitif de communication entre sites. Les mécanismes de communication asyn-

chrone (signaux, événements) permettent des interactions en temps réel, avec ou sans mémorisation ; ils sont également transposables aux systèmes répartis.

Les verrous permettent de mettre en œuvre dans les applications la synchronisation nécessaire au partage d'information (exclusif ou non) entre plusieurs processus.

Enfin, certains langages de programmation comportent des constructions de synchronisation de haut niveau, mises en œuvre par les systèmes sous-jacents au moyen des mécanismes élémentaires. C'est le cas des moniteurs dans Modula-2 ou des primitives de rendez-vous dans Ada.

## 2.4. Mise en œuvre des processus

### Gestion dynamique et représentation interne des processus

Dans un système d'exploitation, les processus sont en général créés et détruits dynamiquement. A partir de quelques processus de base créés à l'initialisation du système, un "arbre généalogique" de processus est ainsi progressivement constitué.

L'opération de création d'un processus doit spécifier son contexte initial. En général, ce contexte (contenu de la mémoire virtuelle, descripteurs de fichiers, etc) est hérité du contexte du processus créateur (le père). Par exemple, dans le système Unix, le contexte initial du processus fils est une copie conforme du contexte de son père ; cette copie peut ensuite être modifiée par chargement de programmes spécifiques. La destruction d'un processus libère toutes les ressources qui lui étaient allouées. Pour faciliter la gestion des ressources, un processus ne peut être détruit que par son père, et la destruction d'un processus entraîne celle de sa descendance.

La représentation interne d'un processus est un descripteur qui contient les éléments suivants : vecteur d'état du processus (contenu du mot d'état et des registres du processeur) ; caractéristiques propres (priorité, protection, pointeur vers une représentation du contexte) ; liens divers (liens de filiation, chaînage dans une file d'attente). Cette représentation interne est manipulée par les programmes du système chargés de la gestion des processus. L'ensemble de ces programmes constitue le noyau de synchronisation.

### Noyau de synchronisation

La fonction du noyau est d'allouer les processeurs aux processus et de réaliser les primitives de synchronisation. Ces deux fonctions sont étroitement liées : en effet, un processus qui se bloque cesse d'avoir besoin d'un processeur et peut donc laisser la place à un processus prêt à s'exécuter ; inversement, lorsqu'un processus bloqué est réveillé, il ne reprend pas nécessairement son exécution, mais devient simplement candidat à l'allocation d'un processeur. La figure 2 représente les transitions d'un processus entre ses états, dans le cas le plus simple où la seule ressource allouée est le processeur. L'allocation de mémoire (voir 4.3) peut introduire des états supplémentaires.

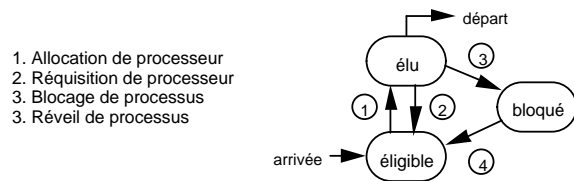


Figure 2 - États d'un processus

L'état actif est subdivisé en deux sous-états : élu (s'exécutant sur un processeur), et éligible (candidat à l'allocation de processeur). La structure principale permettant l'allocation de processeur est la file d'attente des processus éligibles. L'entrée dans cette file (après réveil), ainsi que la politique de réquisition (ou libération forcée) des processeurs est réglée par la politique d'allocation de processeur (voir 4.2), mise en œuvre par un programme **ordonnanceur** (*scheduler*). L'allocation effective de processeur (notamment le choix du processeur lorsqu'il y en a plusieurs) est réalisée par un programme **distributeur** (*dispatcher*). L'allocation de processeur et la synchronisation des processus se traduisent par des mouvements entre différentes files d'attente (fig. 3).

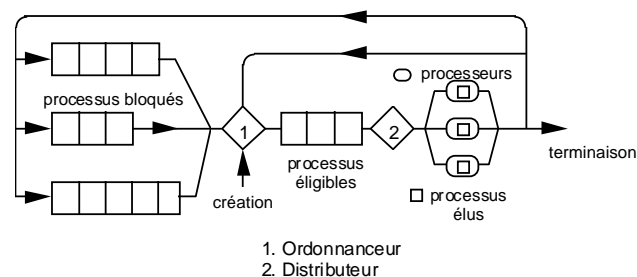


Figure 3 - Allocation de processeur aux processus

Concrètement, ces mouvements sont réalisés par des changements de pointeurs, lors du passage d'une file à une autre, et par des chargements et sauvegardes du contexte des processus, lors de l'allocation et la libération effectives de processeur.

Cette commutation du contexte des processus constitue la composante principale du coût de l'allocation de processeur. Selon que ce contexte comporte uniquement le vecteur d'état du processeur (mot d'état et registres), ou contient en plus la description de la mémoire virtuelle du processus, on parle de processus "légers" ou "lourds". Le système Unix, par exemple, gère des processus "lourds", dont chacun est associé à un espace distinct de mémoire virtuelle. Les micro-noyaux séparent la gestion des espaces de mémoire de celle des processus : les processus légers (*threads*) s'exécutent à l'intérieur d'un espace d'adressage préexistant, et permettent la gestion économique d'activités multiples à l'intérieur d'un tel espace. Ces processus communiquent directement par leur mémoire commune. En contrepartie, ils ne bénéficient plus de la protection mutuelle qu'apporte la séparation des espaces d'adressage.

Une étape supplémentaire dans la recherche d'une commutation rapide de contexte conduit à gérer des processus légers hors du noyau, à l'intérieur même d'une application. Lors des commutations de contexte, on gagne alors le temps de l'appel et du retour d'un programme du noyau et des vérifications associées. Ces "pseudo-processus" (*user-level threads*) sont ordonnancés par un programme de bibliothèque, à l'intérieur d'un processus "lourd" ordinaire. Ils restent donc globalement soumis à l'ordonnancement de ce processus par le système d'exploitation, ce qui limite la généralité de leur utilisation.

Les travaux récents visent à combiner les avantages de la gestion des processus légers dans le noyau et ceux de la gestion de pseudo-processus dans l'espace des utilisateurs, sur des multiprocesseurs à mémoire commune. Le principe consiste à utiliser un mécanisme à deux niveaux pour l'allocation de processeur. Un allocateur "haut" ordonnance les pseudo-processus, dans chaque espace d'adressage, en leur allouant les processeurs couramment affectés à cet espace. Un allocateur "bas" répartit les processeurs entre les espaces d'adressage. Lorsqu'un pseudo-processus exécute une opération bloquante dans le noyau, le processeur qu'il utilisait est remis à la disposition de l'allocateur "haut" dans son espace d'adressage. Lorsqu'une interruption arrive, elle doit être traitée rapidement ; un processeur est alors "emprunté" à l'un des espaces d'adressage (ce qui entraîne une réallocation des pseudo-processus dans cet espace), puis est restitué au retour de l'interruption.

### 3. Allocation des ressources

#### 3.1. Objectifs d'une politique d'allocation

On appelle **ressource** tout objet nécessaire à l'exécution d'un processus. Les ressources gérées par le système d'exploitation sont les processeurs, la mémoire principale et secondaire, les organes d'entrée-sortie et les voies de communication.

Une politique d'allocation de ressources doit satisfaire de manière équitable les demandes des processus, en évitant les phénomènes indésirables que sont l'interblocage, la privation et l'écroulement. L'allocation de ressources est évaluée par des facteurs de qualité de service dont la définition dépend de la classe de ressources considérée : temps d'attente (moyenne et variance), temps total de traitement, débit des travaux, taux de couverture de la demande, coût supplémentaire de gestion, etc. Une allocation est équitable lorsque deux processus de même priorité reçoivent en moyenne un traitement équivalent, et lorsque la qualité de service obtenue par un processus est une fonction croissante de sa priorité.

L'**interblocage** est une situation dans laquelle deux ou plusieurs processus sont bloqués indéfiniment, chacun attendant la libération de ressources allouées à un autre processus du groupe. Par exemple, la séquence ci-après peut conduire à l'interblocage (les ressources ici allouées sont les verrous qui contrôlent l'accès aux fichiers).

<b>processus p1</b>	<b>processus p2</b>
verrouiller-exclusif(f1)	verrouiller-exclusif(f2)
...	...
verrouiller-exclusif(f2)	verrouiller-exclusif(f1)
...	...
déverrouiller(f2)	déverrouiller(f1)
déverrouiller(f1)	déverrouiller(f2)

L'interblocage aurait été évité si les processus avaient verrouillé les fichiers dans le même ordre. Plus généralement, une méthode de prévention de l'interblocage consiste à répartir les ressources en classes ordonnées, et à allouer les ressources à chaque processus dans l'ordre des classes.

Lorsque l'interblocage est un événement rare, une autre méthode consiste à le traiter quand il se produit, par libération forcée des ressources ; on accepte alors le coût d'un retour en arrière et d'une reprise des processus victimes de la réquisition.

La **privation** (ou famine) est l'attente indéfinie que peut subir un processus lorsque les ressources sont systématiquement allouées à des processus plus prioritaires que lui. On peut l'éviter en traitant les demandes dans l'ordre des arrivées, ou en faisant croître la priorité d'un processus avec son temps d'attente. L'allocateur peut également utiliser la **réquisition** après un temps d'utilisation fixé. La réquisition (*preemption*) consiste à retirer une ressource à un processus qui en a encore besoin.

L'**écroulement** est un phénomène de congestion qui résulte d'une demande de ressources supérieure à un seuil de saturation. Les coûts d'allocation deviennent dominants, et le taux d'utilisation des ressources (donc la vitesse de progression des processus) se dégrade. L'écroulement est évité par une régulation dynamique de charge (cf. 4.3) qui vise à limiter la demande instantanée en restreignant temporairement le nombre de processus demandeurs.

### 3.2. Allocation de processeur

La politique d'allocation du processeur définit l'attribution du processeur aux processus éligibles. On distingue plusieurs classes de politiques, selon la nature des événements qui déclenchent la réallocation d'un processeur. Dans les politiques sans réquisition, un processus qui s'exécute sur un processeur ne le relâche que s'il se termine ou se bloque. Les risques de privation conduisent à introduire des délais de garde : un processeur n'est alloué à un processus que pour un temps limité. Enfin, l'introduction de priorités entre processus, et la prise en compte de contraintes de temps réel, conduisent à définir des politiques dites "préemptives", dans lesquelles l'arrivée ou le réveil d'un processus  $p$  plus prioritaire que celui qui s'exécute sur un processeur  $P$  provoque la réallocation de  $P$  au profit de  $p$ . Le tableau ci-après résume cette classification.

Politiques	Événements déclenchants	Exemples
Sans réquisition	Fin ou blocage de processus	FIFO, SJF
Avec délai de garde	- <i>idem</i> + interruption d'horloge	RR (tourniquet)
Préemptives	- <i>idem</i> + réveil processus prioritaire	RR multi-niveaux, priorité variable, EDF

Quelle que soit la politique adoptée, il existe un degré de charge (débit de travail par unité de temps) qui sature le système. Au voisinage de ce seuil, le temps de réponse moyen tend à augmenter rapidement ; une dispersion importante des temps d'exécution autour de leur valeur moyenne tend également à accroître le temps de réponse moyen, et cela d'autant plus que le système est chargé.

On constate que l'attente est d'autant moins bien supportée que la durée effective du travail demandé est faible. En l'absence de contraintes de temps réel, le principe directeur des politiques d'allocation de processeur vise donc à *réduire le temps d'attente des travaux courts*. Nous examinons successivement les principales politiques pratiquées.

a) *Politiques sans réquisition*. Ces politiques ont été introduites pour le traitement de travaux par lots. La politique "Premier arrivé, premier servi" (*First In, First Out*, ou *FIFO*), pénalise les travaux courts qui arrivent après des travaux longs. L'ordonnement optimal (qui minimise le temps d'attente moyen) consiste à traiter les demandes dans l'ordre croissant des temps d'exécution (*Shortest Job First*, ou *SJF*). On doit connaître à l'avance les temps d'exécution, ou au moins en avoir une bonne estimation. D'autre part, la politique SJF présente un risque de privation si le rythme d'arrivée des travaux courts est grand. Les politiques suivantes visent à éviter ces inconvénients.

b) *Politiques à délai de garde*. En général, la durée d'exécution est inconnue *a priori*. La politique SJF peut être approchée par celle du tourniquet (*round robin*, ou *RR*), qui consiste alors à allouer successivement le processeur aux processus demandeurs par "quanta" de taille fixe (de l'ordre de 100 millisecondes). Dans un système où la majorité des processus sont interactifs, la plupart des demandes sont satisfaites en un quantum de temps. Cette politique réalise une approximation d'un modèle dit "du processeur partagé", dans lequel chaque travail disposerait sans délai d'un processeur dont la vitesse serait celle du processeur réel divisée par le nombre courant de travaux présents. Dans ce modèle idéal, le temps de séjour d'un travail dans le système est exactement proportionnel à sa durée d'exécution, le facteur de proportionnalité dépendant de la charge du système.

c) *Politiques préemptives*. Lorsque des priorités sont attachées aux processus, elles peuvent être prises en compte de deux manières : par la réquisition d'un processeur au profit d'un processus prioritaire, et par un traitement différencié des processus selon leur priorité, qui amène à gérer des files d'attente multiples. Le tourniquet multi-niveaux combine ces deux procédés. C'est la politique la plus fréquemment utilisée dans les systèmes d'exploitation actuels, avec des variantes diverses.

On définit plusieurs niveaux associés à des valeurs décroissantes de la priorité, avec une file d'attente par niveau. Lorsqu'un processus est créé ou réveillé, il entre dans la file correspondant à sa priorité. Lorsqu'il a épuisé son quantum au niveau  $i$  sans se terminer, il entre dans la file de niveau  $i+1$  (ou reste dans la file  $n$  s'il y était déjà). Des valeurs croissantes du quantum sont associées aux différents niveaux (dans l'exemple de la figure 6, la valeur du quantum est doublée au passage d'un niveau au suivant).

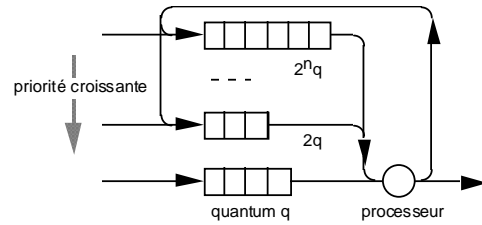


Figure 6 - Tourniquet multi-niveaux

La politique est préemptive en ce sens que l'arrivée ou le réveil d'un processus de priorité  $i$ , alors qu'un processus de priorité inférieure s'exécute, provoque la réallocation du processeur. Un processus de niveau  $i$  ne peut s'exécuter que si les files de tous les niveaux inférieurs sont vides. Le système peut modifier la priorité d'un processus (lors de son réveil) pour tenir compte, par exemple, du temps déjà passé dans le système ou des ressources demandées.

Les contraintes de "temps réel" se traduisent par la nécessité de terminer l'exécution d'un processus avant une échéance fixée. Une politique fréquemment adoptée pour ce type de travaux consiste à leur attribuer une priorité élevée et à les traiter dans l'ordre des échéances (*Earliest Deadline First*, ou *EDF*), et de façon préemptive.

Pour les systèmes multiprocesseurs, il est utile de combiner les politiques générales ci-dessus avec une gestion de processus à deux niveaux, dont le principe a été indiqué en 2.4. Différentes applications peuvent utiliser des politiques différentes d'ordonnement pour les pseudo-processus (processus légers gérés par les utilisateurs). Le choix du processeur à allouer n'est pas neutre : il est en effet préférable, lorsque c'est possible, de réallouer à un processus un processeur qu'il a récemment utilisé : on augmente ainsi la probabilité de retrouver des données utiles dans le cache du processeur, et donc d'économiser le temps de leur chargement.