Gestion des processus

Table des matières

1. Définitions..2
2. États d'un processus..2
3. Ordonnancement..4
  3.1. Contexte d'un processus..4
4. La gestion des processus..8
  4.1. Algorithme du tourniquet (Round Robin)..9
5. Synchronisation de processus..10
  5.1. Section critique..10
  5.2. Outils de synchronisation..11
    5.2.1. TAS..11
    5.2.2. Sémaphores..11
  5.3. Producteur et consommateur..13
  5.4. Lecteur et écrivain..14

Un processus (en anglais, process) est un programme en cours d'exécution par un ordinateur. Un ordinateur équipé d'un système d'exploitation à temps partagé est capable d'exécuter plusieurs processus de façon quasi-simultanée. Par analogie avec les télécommunications, on nomme multiplexage ce procédé. S'il y a plusieurs processeurs, l'exécution des processus est distribuée de façon équitable sur ces processeurs.

1. Définitions

Programme :

Un programme est une suite figée d'instructions, un ensemble statique.

Processus :

Plusieurs définitions existent ; on peut citer :

- Un processus est l'instance d'exécution d'un programme dans un certain contexte pour un ensemble particulier de données.
- On peut aussi caractériser un processus par les tables qui le décrivent dans l'espace mémoire réservé au système d'exploitation (partie résidente du système). Ces tables définissent les contextes matériel (partie dynamique du contexte : contenu des registres...) et logiciel (partie statique du contexte : droits d'accès aux ressources) dans lesquels travaille ce processus.

2. États d'un processus

Dans un système multitâches, plusieurs processus peuvent se trouver simultanément en cours d'exécution : ils se partagent l'accès au processeur.

Un processus peut prendre 3 états

- État actif ou élu (running): le processus utilise le processeur.
- État prêt ou éligible (ready. le processus pourrait utiliser le processeur si il était libre (et si c'était son tour).
- État en attente ou bloqué : le processus attend une ressource (ex : fin d'une entrée-sortie).

Le S.E choisit un processus qui deviendra actif parmi ceux qui sont prêts.

Tout processus qui se bloque en attente d'un événement, passe dans l'état bloqué tant que l'événement attendu n'est pas arrivé. Lors de l'occurrence de cet événement, le processus passe dans l'état prêt. Il sera alors susceptible de se voir attribuer le processeur pour continuer ses activités.

Les processus en attente sont marqués comme bloqués dans la table des processus. Ils se trouvent généralement dans la file d'attente liée à une ressource (imprimante, disque, ...).

Le changement d'état d'un processus peut être provoqué par :

- un autre processus (qui lui a envoyé un signal, par exemple)
- le processus lui-même (appel à une fonction d'entrée-sortie bloquante,
- une interruption (fin de quantum, terminaison d'entrée-sortie, ...)

La sortie de l'état actif pour passer à l'état prêt se produit dans le cas des ordonnancements préemptifs lorsqu'un processus plus prioritaire que le processus actif courant devient prêt.

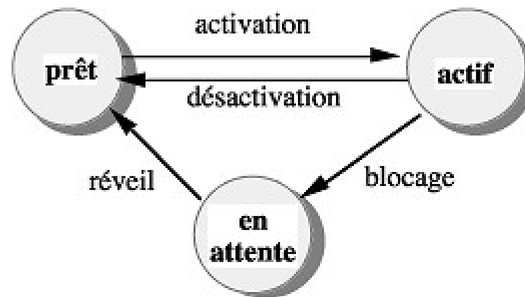
D'une manière générale :

- le passage de l'état actif à l'état prêt est provoqué par le système en fonction de sa politique d'ordonnancement (fin de quantum, préemption du processus actif si un processus plus prioritaire devient prêt dans le cas des politiques préemptives, ...),
- le passage de l'état actif à l'état bloqué est provoqué par le programme exécuté par le

processus.

- Les transitions entre états provoquent le passage d'un état à un autre :

Activation	prêt -> actif.
Désactivation (ou préemption,ou réquisition)	actif -> prêt.
Mise en attente (ou blocage)	actif -> en attente.
Réveil	attente -> prêt.



- Les opérations que l'on peut effectuer sur les processus sont :
  - Création (ex : fork, création d'un processus sous UNIX),
  - Destruction (ex : kilt - 9 , destruction d'un processus sous UNIX),
  - Activation, désactivation, blocage(ex : wait sous UNIX),

Le système d'exploitation gère les transitions entre les états. Pour ce faire il maintient une liste de processus éligibles et une liste de processus en attente. Il doit également avoir une politique d'activation des processus éligibles (ou politique d'allocation du processeur aux processus) : parmi les processus éligibles, faut-il activer celui qui est éligible depuis le plus de temps, celui qui aurait la plus forte priorité ou celui qui demande l'unité centrale pour le temps le plus court (à supposer que cette information soit disponible)? Les processus sont classés dans la file des processus éligibles par

l'ordonnanceur (en anglais scheduler). C'est le distributeur (en anglais dispatcher) qui se chargera de leur activation au moment voulu.

Afin d'éviter qu'un processus accapare l'unité centrale, le système d'exploitation déclenche un temporisateur à chaque fois qu'il alloue l'unité centrale à un processus. Quand ce temporisateur expire, si le processus occupe toujours l'unité centrale (il peut avoir été interrompu par un événement extérieur ou s'être mis en attente), le système d'exploitation va le faire passer dans l'état prêt et activer un autre processus de la liste des processus éligibles. La désactivation d'un processus peut se faire sur expiration du temporisateur ou sur réquisition (en anglais preemption) du processeur central par un autre processus.

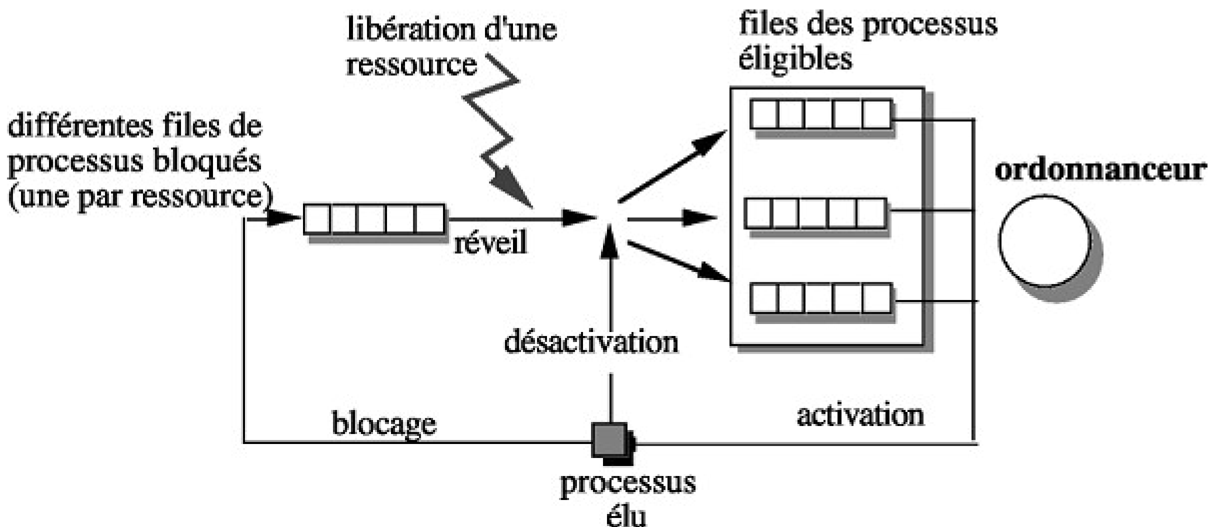
Un temporisateur peut être vu comme un système de compte à rebours ou de sablier. On lui donne une valeur initiale qu'il décrémente au rythme de l'horloge de l'ordinateur jusqu'à atteindre la valeur zéro. Ici, la valeur initiale du temporisateur est le quantum de temps alloué à l'exécution du processus. A l'expiration du temporisateur le système d'exploitation est immédiatement prévenu par une interruption. Nous verrons cette notion plus en détail dans la suite de ce chapitre.

D'autres événements sont signalés au système par des interruptions c'est le cas des fins d'entrées-sorties. Le système est ainsi prévenu que le ou les processus qui étaient en attente sur cette entrée-

sortie peuvent éventuellement changer d'état (être réveillés).

3. Ordonnancement

Dans un système multitâches, le système d'exploitation doit gérer l'allocation du processeur aux processus. On parle d'ordonnancement des processus.



Il existe deux contextes : le contexte matériel et le contexte logiciel.

Le contexte matériel est la photographie de l'état des registres à un instant t : compteur ordinal, pointeur de pile, registre d'état qui indique si le processeur fonctionne en mode utilisateur (user) ou en mode noyau (kernel), etc.

Le contexte logiciel contient des informations sur les conditions et droits d'accès aux ressources de la machine : priorité de base, quotas sur les différentes ressources (nombre de fichiers ouverts, espaces disponibles en mémoires virtuelle et physique...).

Chaque fois que le processeur passe à l'exécution d'un nouveau processus il doit changer les configurations de ses registres. On parle de changement de contexte. Certaines informations (une partie du contexte logiciel) concernant les processus doivent rester en permanence en mémoire dans l'espace système. D'autres peuvent être transférées sur disque avec le processus quand celui ci doit passer de la mémoire au disque (quand il n'y a plus de place en mémoire pour y laisser tous les processus, par exemple). Il s'agit en général de tout le contexte matériel et d'une partie du contexte logiciel.

3.1. Contexte d'un processus

Par contexte, on entend toutes les informations relatives à l'exécution d'un processus, telles que :

- l'état des registres (compteur ordinal, registre d'état, registres généraux),
- les registres décrivant l'espace virtuel du processus et son espace physique d'implantation,
- les pointeurs de piles,
- les ressources accédées (ex : droits d'accès, fichiers ouverts),
- autres informations (ex : valeur d'horloge).

Le contexte doit être sauvegardé quand le processus est désactivé ou bloqué et doit être restauré

lorsqu'il est réactivé.

Les processus sont décrits dans une table des processus, une entrée de cette table donne des informations sur l'état du processus et permet de retrouver les 3 régions : code, data et stack, c'est-à-dire instructions, données et pile. Seules les informations dont le système a besoin en permanence sont dans la table des processus, les autres sont dans la u-structure qui peut être éventuellement passer sur le disque si nécessaire.

Sous Unix, chaque commande lancée par un utilisateur est exécutée dans le cadre d'un processus différent, fils du shell avec lequel travaille cet utilisateur.

Unix alloue 3 régions en mémoire pour chaque processus:

- une région pour le code exécutable (text segment),
- une région pour la pile (stack segment),
- une région pour les données (data segment).

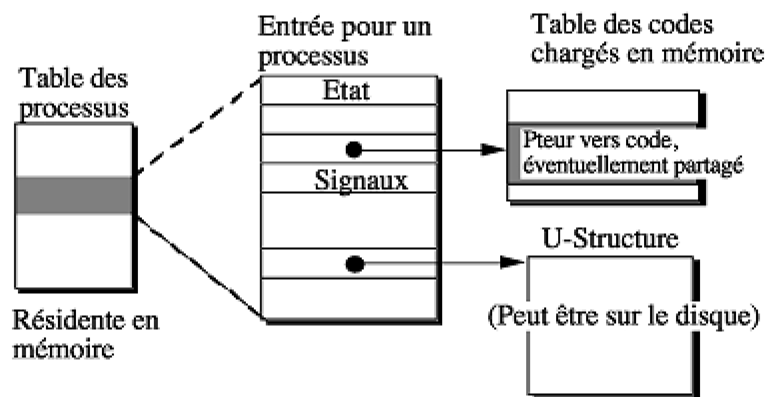
Unix mémorise pour chaque processus des informations de gestion du type :

- identificateur du processus (pid)
- configuration des registres (pile, compteur ordinal),
- répertoire courant, fichiers ouverts,

Remarque :

Le pointeur vers la table des textes (fichiers exécutables !) donne, non pas le début du programme à exécuter mais un pointeur vers ce programme, ce qui permet à la fois le partage de code exécutable par plusieurs processus différents et le déplacement de celui-ci par le système.

A chaque processus correspond une entrée dans la table des processus (process table).



Création d'un processus sous Unix, réalisée par l'appel système fork :

- duplication des zones mémoires attribuées au père, le père et le fils ne partagent pas de mémoire,
- le fils hérite de l'environnement fichiers de son père, les fichiers déjà ouverts par le père sont vus par le fils. Ces fichiers sont partagés.

Juste après l'exécution de fork, la seule information qui diffère dans les espaces mémoire attribués au père et au fils est le contenu de la case contenant la valeur de retour de fork (ici f).

Cette case contient le pid du fils dans l'espace mémoire du père et contient zéro dans l'espace

mémoire du fils.

```
int main ()
{
	int f = fork ();
	if (f == -1){
		printf("Erreur : le processus ne peut etre cree\n")
		exit (1);
		}
	if (f == 0){
		printf("Coucou, ici processus fils\n")
		...;
	}

	if (f != 0){
		printf("Ici processus pere\n")
		...;
	}
}
```

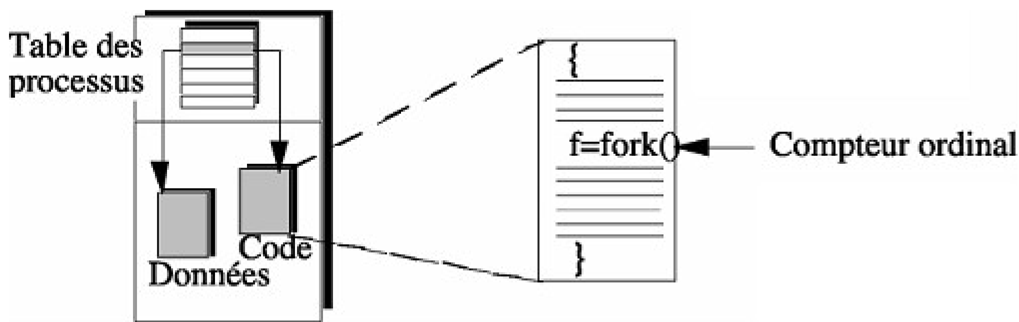
Après l'appel à fork, les espaces d'adressage du père et du fils contiennent chacun une case dont le nom est f. La case f située dans l'espace d'adressage du fils contient zéro, la case f qui se trouve dans l'espace du père contient le numéro de processus attribué au fils (le pid, pour process identifier).

Exemple :

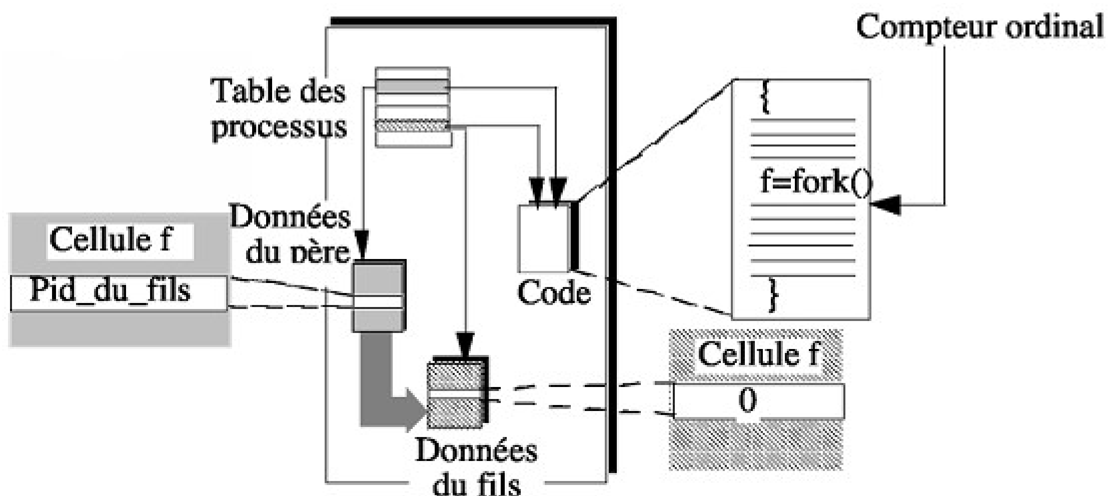
```
// programme myfork.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main ()
{
	int f = fork();

	printf("Valeur retournee par la fonction fork: %d\n", (int)f);
	printf("Je suis le processus numero %d\n", (int)getpid());

	return 0;
}
```

```
$ gcc myfork.c -o  myfork
$  ./myfork
Valeur retournee par la fonction fork: 717
Je suis le processus numero 716
```

avant fork :



après fork :



Exemple : ce programme créé un processus qui va faire appel à exec pour exécuter le fichier dont on lui passe le nom sur la ligne de commande.

```
// programme fexec.c
int main (int argc, char *argv[])
{
	int Pid, Fils, etat;

	if ( argc != 2 ) {
		printf(" Utilisation : %s fic. a executer ! \n", argv[0]);
		exit(1);
		}

	printf("Je suis le pid %d je vais faire fork\n", (int)getpid());
	Pid = fork();

	switch ( Pid ) {
		case 0 :
			printf("Coucou ! je suis le fils %d\n", (int)getpid());
			printf("%d : Code remplace par %s\n", (int)getpid(), argv[1]);
			execl(argv[1], argv[1], (char *)0);
			printf(" %d : Erreur lors du exec \n", (int) getpid());
			exit (2);
		case -1 :
			printf("Le fork n'a pas reussi");
			exit (3) ;
		default :
```

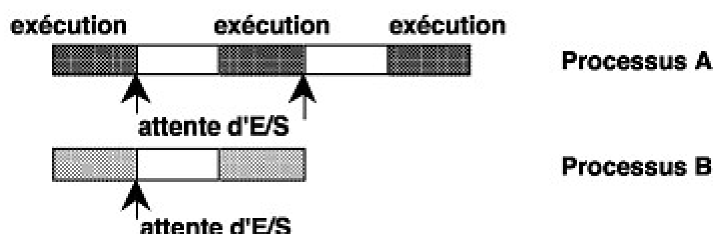
```
                /* le pere attend la fin du fils */
                printf("Pere numero %d attend\n ", (int)getpid());
                Fils = wait (&etat);
                printf("Le fils etait : %d ", Fils);
                printf(".. son etat etait :%0x (hexa) \n", etat);
                exit(0);
        }
}
```

```
$ ./fexec ./myfork
Je suis le pid 5195 je vais faire fork
Pere numero 5195 attend
Coucou ! je suis le fils 5196
5196 : Code remplace par exol
Valeur retournee par la fonction fork: 5197
Je suis le processus numero 5196
Le fils etait : 5196 ... son etat etait :0 (hexa)
$ Valeur retournee par la fonction fork: 0
Je suis le processus numero 5197
```

4. La gestion des processus

Un système propose la multiprogrammation s'il permet de charger simultanément plusieurs processus en mémoire et de leur faire partager le l'unité centrale. Le principe de fonctionnement est de laisser un processus s'exécuter jusqu'à ce qu'il ait besoin de se mettre en attente (par exemple sur une fin d'entrée / sortie). Pendant que ce processus est en attente, un autre processus peut utiliser à son tour le processeur.

Pour illustrer ce fonctionnement, supposons que l'on ait deux processus A et B à exécuter, chacun alternant des périodes d'exécution dans l'unité centrale et des périodes d'attente. L'exécution totale du processus A nécessite 5 unités de temps, dont 2 d'attente, celle du processus B nécessite 3 unités de temps, dont 1 d'attente.



L'exécution de ces deux processus sans multiprogrammation prend 8 unités de temps. Avec multiprogrammation, elle ne prend plus que 5 unités de temps (soit la durée d'exécution du processus A).

sans gestion multitâches

avec gestion multitâches

Dans la suite de ce chapitre, nous allons montrer le fonctionnement des systèmes qui supportent la multiprogrammation. Il s'agira de décider comment l'unité centrale est partagée entre les processus

chargés en mémoire. C'est ce que l'on appelle la politique (ou algorithme) d'allocation du processeur ou encore politique d'ordonnancement. On peut décider que c'est le premier processus qui est prêt à s'exécuter qui s'exécute (politique Premier Arrivé Premier Servi). On peut aussi choisir le processus dont la durée d'exécution est la plus courte pour lui attribuer l'unité centrale d'abord (politique Plus Court d'Abord). Nous examinerons différentes politiques d'allocation dans la suite de ce chapitre.

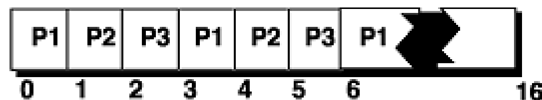
Plusieurs politiques d'allocations étant possibles, la question se pose de savoir comment les évaluer. Il faudra tout d'abord s'assurer que la politique ne crée pas de famine (il y a famine si un processus prêt à s'exécuter n'accède pas à l'unité centrale en un temps fini). On pourra ensuite utiliser différents critères d'évaluation tels que le rendement, le temps de service, le temps d'attente et le temps de réponse, qui sont définis dans le paragraphe suivant.

4.1. Algorithme du tourniquet (Round Robin)

Le temps est découpé en tranches ou quantum de temps. Un quantum de temps est alloué à chaque processus de la file des processus éligibles à tour de rôle.

Conçu pour les systèmes "temps partagé", afin d'obtenir une utilisation équitable du processeur par les processus présents en mémoire.

Exemple : si le quantum est d'une unité de temps



- Temps moyen de service : 9 = (5+6+16)/ 3, si tous sont prêts au temps 0. (Meilleur que celui donné par l'algorithme FCFS pour le même ordre d'arrivé.)
- Performances sensibles à la taille choisie pour le quantum.

Un processus utilisateur démarre avec une priorité de base PBase (PBase = 60). Toutes les secondes le système réévalue cette priorité et la pondère de façon à favoriser les processus interactifs, c'est-à-dire ceux qui font beaucoup d'entrées-sorties. Le mécanisme de pondération est décrit ci-dessous.

À chaque processus est associé un coefficient C dont la valeur est fonction du temps passé dans le processeur la dernière fois qu'il a été actif. C est initialisé ainsi :

C = temps cpu consommé

La valeur C est recalculée toutes les secondes de la façon suivante:

$$C = \frac{1}{2} C$$

et on attribue au processus la priorité P suivante, après avoir recalculé C

$$P = PBase + C$$

Le processus peut modifier sa priorité en ajustant le paramètre nice (valeurs ≥ 20), sa priorité deviendra:

$$P = PBase + C + (nice - 20)$$

De façon générale, quelle que soit la manière de réévaluer le coefficient C :

P = PBase + f(cpu consommé, temps d'attente) + (nice -20)

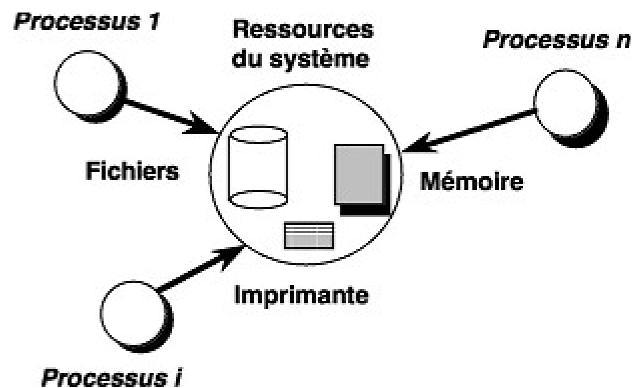
Lors de son réveil après attente sur un événement E (e/s, pause, ...) un processus se voit attribuer

une priorité P = f(E), avec interruption éventuelle du processus actif courant, si ce dernier est moins prioritaire. La priorité d'un processus en attente sur un événement dépend donc uniquement du type de ce dernier, ceci pose un problème pour la gestion des travaux de type batch.

La priorité 25 est la limite qui indique si le processus est interruptible ou non sur réception d'un signal.

5. Synchronisation de processus

Un système d'exploitation dispose de ressources (imprimantes, disques, mémoire, fichiers, base de données, ...), que les processus peuvent vouloir partager.



Ils sont alors en situation de concurrence (race) vis-à-vis des ressources. Il faut synchroniser leurs actions sur les ressources partagées.

Il n'y a problème de synchronisation que si un des processus impliqués dans l'accès en parallèle à la ressource partagée la modifie.

Si les processus en accès simultané sont tous en mode consultation, il n'y a pas de problème de synchronisation puisque la ressource partagée n'est pas modifiée.

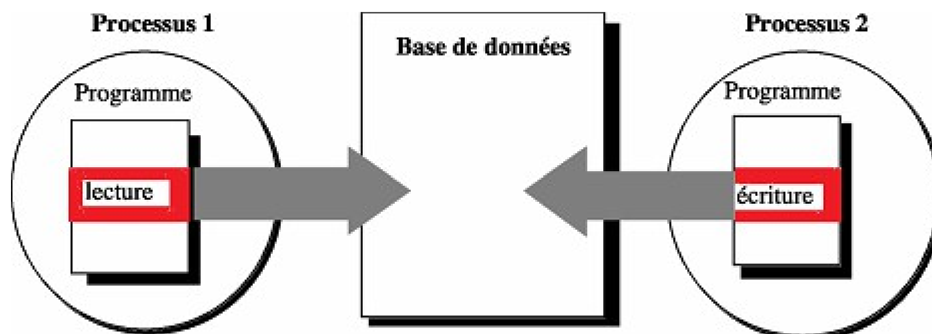
5.1. Section critique

La partie de programme dans laquelle se font des accès à une ressource partagée s'appelle une section critique.

Dans la plupart des cas l'accès à cette section critique devra se faire en exclusion mutuelle, ce qui implique de rendre indivisible ou atomique la séquence d'instructions composant la section critique.

Exemple : accès à une base de données.

> Plusieurs processus peuvent la lire simultanément, mais quand un processus la met à jour, tous les autres accès doivent être interdits.



sections critiques des processus relativement à la base de données

Les conditions d'accès à une section critique sont les suivantes :

- deux processus ne peuvent être ensemble en section critique,
- un processus bloqué en dehors de sa section critique ne doit pas empêcher un autre d'entrer dans la sienne,
- si deux processus attendent l'entrée dans leurs sections critiques respectives, le choix doit se faire en un temps fini,
- tous les processus doivent avoir la même chance d'entrer en section critique,
- pas d'hypothèses sur les vitesses relatives des processus (indépendance vis à vis du matériel).

5.2. Outils de synchronisation

- Matériels
  - emploi d'une instruction de type Test and Set qui teste l'état d'une case mémoire commune aux deux processus et change sa valeur,
  - inhibition des interruptions (possible seulement dans le cas d'un processus se déroulant en mode privilégié).
- Logiciels :
  - sémaphores (outils logiciels mis à la disposition des utilisateurs par le système d'exploitation),
  - algorithmes de synchronisation.

5.2.1. TAS

On utilise une instruction du langage machine parce qu'une instruction de ce niveau est insécable. Lorsque le processeur commence l'exécution d'une instruction du langage machine, il va jusqu'au bout même si une interruption arrive.

Les inconvénients de cette technique du TAS sont :

- l'attente active (busy waiting) : un processus qui trouve la variable de synchronisation à 1 continuera à la tester jusqu'à ce qu'elle passe à 0. Il reste donc actif, alors qu'il pourrait être suspendu et rangé dans une file d'attente.
- le fait qu'on ne sait pas qui va prendre la main. Ce problème est lié au précédent. En effet, puisqu'il n'y a pas de gestion de la file des processus en attente sur le passage de la variable à 0, le premier qui obtient une réponse satisfaisante au test entre dans sa section critique. Ce n'est pas forcément celui qui la scrute depuis le plus longtemps !

Remarque : cette instruction TAS permet de synchroniser des activités en environnement multiprocesseur.

5.2.2. Sémaphores

Les opérations d'initialisation (Init), de demande de ressources (P) et d'allocation de ressources (V) sont données par le SE qui garantit leur atomicité.

Il fait en sorte, que, du point de vue des processus impliqués, ces opérations s'exécutent sans être

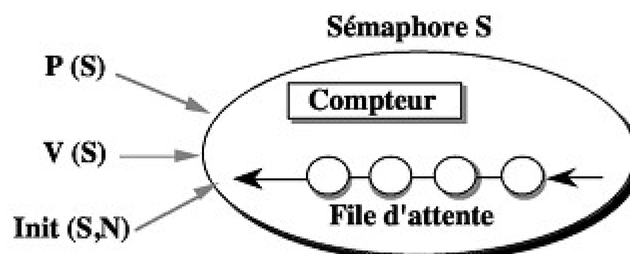
interrompues.

Ceci n'est pas forcément vrai d'un point de vue plus global, en effet le système peut traiter une autre activité, plus importante pour lui, pendant l'exécution de ces fonctions.

Remarques :

1. sur la valeur du compteur associé au sémaphore :
   - si il elle est positive, elle indique le nombre de ressources disponibles ou le nombre de processus pouvant avoir accès à la ressource,
   - si elle est négative, sa valeur absolue donne le nombre de processus bloqués et figurant dans la file d'attente.
2. sur les états des processus :
   - les processus qui font passer le compteur à une valeur négative en appelant la fonction P, sont rangés dans la file d'attente et passent dans l' état : bloqué,
   - une opération V va sans doute débloquer un des processus qui se trouvent dans l'état bloqué. On souligne ici que la transition : bloqué -> prêt dont bénéficie un processus ne peut être faite par un processus lui-même actif.

- Un sémaphore s associé à une ressource R est un objet composé :
  - d'un compteur S. COMPT qui indique le nombre d'éléments disponibles ou le nombre de processus en attente sur R,
  - d'une file d'attente S . FA où sont chaînés les processus en attente sur la ressource R.



- Il est accessible par les opérations atomiques suivantes :

Init (S, val) : S. COMPT = val (nombre d'éléments de ressource) et S. FA = vide.

- P (Passeren) pour demander un élément de la ressource.

P (S) : On décrémente S.COMPT. Si S.COMPT < 0 alors le processus est mis dans S.FA et passe à l'état bloqué, sinon un élément de ressource lui est alloué.

- V (Vrijgeven) libère un élément de la ressource.

V(S) : On incrémente S.COMPT. Si S.COMPT ≤ 0 alors on sort un processus de S.FA, il passe dans la file des prêts et on peut lui allouer un élément de ressource.

```
//Réalisation d'un sémaphore

/***************** Init *******************/
Init(Semaphore *Le_Semaphore, int Valeur)
{
```

```
        (*Le_Semaphore).Compteur   = Valeur ;
        (*Le_Semaphore).FiledAttente initialisée à vide ;
}

/***************** P *******************/
void P(Semaphore *Le_Semaphore)
{
                (*Le_Semaphore).Compteur = (*Le_Semaphore).Compteur - 1;
                if ( (*Le_Semaphore).Compteur < 0 ) {
                        // mettre le processus dans la file d'attente et le faire passer
dans l'état bloqué
                        }
}

/***************** V *******************/
void V(Semaphore *Le_Semaphore) {
        (*Le_Semaphore).Compteur = (*Le_Semaphore).Compteur + 1;
        if ( (*Le_Semaphore).Compteur <= 0 ) {
                // sortir un processus de la file d'attente
                // et le faire passer dans l'état prêt
        }
}
```

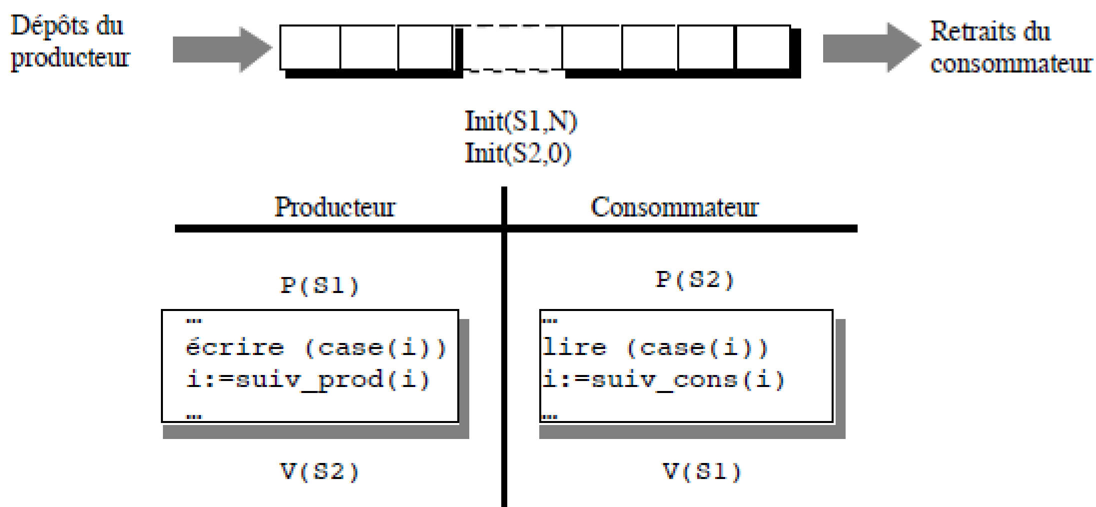
5.3. Producteur et consommateur

Un producteur dépose des messages dans un tampon, ceux-ci sont retirés par un consommateur. Les retraits et dépôts peuvent être simultanés.

Le producteur doit vérifier que le tampon comporte des cases libres avant d'y déposer un message et le consommateur doit s'assurer qu'il y a bien des messages à retirer. Si ces conditions ne sont pas requises, le processus (producteur ou consommateur) doit se mettre en attente.

Deux sémaphores sont utilisés :

- S1 qui indique les ressources disponibles pour le producteur, c'est-à-dire le nombre de cases vides,
- S2 qui indique les ressources disponibles pour le consommateur, c'est-à- dire le nombre de cases pleines,



Ce type de synchronisation est mis en œuvre dans le système Unix sous forme de la primitive pipe. Celle-ci crée un fichier (appelé pipe, ou tube en français) à double accès qui ne peut être lu que si l'on a déjà écrit dedans et dans lequel on ne peut écrire que s'il y a des cases libres.

Le système d'exploitation se charge de la gestion des accès, évitant à l'utilisateur le maniement, délicat, de sémaphores. L'utilisateur voit le tampon commun aux processus comme un fichier normal, il sera ouvert par une commande open et manipulé avec des read et write.

5.4. Lecteur et écrivain

Il s'agit de gérer l'accès à un fichier partagé. Plusieurs consultations (lectures) en parallèle sont possibles, mais à partir du moment où une mise à jour (une écriture) est en cours, tous les autres accès, que ce soit en lecture ou en écriture, doivent être interdits.

Programme pour les lecteurs	Programme pour les écrivains

P(S); /* Si un écrivain arrive, le laisser passer */
V(S);
P(SL);
    Nb_Lect = Nb_Lect + 1;
    si (Nb_Lect == 1) P(X);
V(SL);
Consulter
P(SL);
    Nb_Lect = Nb_Lect - 1;
    si (Nb_Lect == 0) V(X);
V(SL);
	P(S); /* Interdire les accès aux lecteurs et écrivains suivants */
	P(X);
	Modifier
	V(S) /* Permettre l'accès au suivant */
	V(X);

L'accès à la variable partagée par les lecteurs Nb_Lect est contrôlé par le verrou appelé XL.

Les lecteurs sont prioritaires et ne provoque pas la famine pour les écrivains : un écrivain peut interrompre le flot des lecteurs et interdire à ceux qui le suivent de lui prendre son tour.