La Gestion des Processus

Plan

1 **Définitions, exemples**

2. Politiques d'allocation du processeur

3. Synchronisation des processus

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.
- C'est l'état de la machine à un instant t.

Plusieurs processus peuvent exécuter parallèlement le même programme (notion de réentrance). L'exemple le plus classique est celui des éditeurs (edit, vi, emacs, ...). Chacun des utilisateurs simultanés de ces programmes travaille dans son propre contexte (le compteur ordinal de leur contexte mémorise leur position dans l'éditeur) sur ses données propres (fichiers créés et modifiés).

Remarques :

- Les performances d'un même programme seront donc très différentes suivant qu'il se déroule dans le contexte d'un processus plus ou moins privilégié. Et le déroulement du même processus pourra se faire très différemment suivant la charge de la machine (non déterminisme des processus).
- Le remplacement sur les ordinateurs dits personnels (du type IBM-PC) de MSDOS par Windows ou Unix les transforme en machine à processus au même titre que les grands systèmes. La compréhension et la maîtrise du fonctionnement de ces machines demande alors le même niveau de connaissance que celles des *mainframes*.

Programme vs Processus

- Un *programme* est une suite d'instructions (un objet statique).
- Un *processus* est un programme en exécution et son *contexte* (un objet dynamique).
    - Dans un environnement monotâche la notion de processus est réduite à sa plus simple expression.
    - Dans un système multitâches (ex : Linux, UNIX), plusieurs processus s'exécutent "simultanément". Ils doivent se partager l'accès au processeur.
    - Plusieurs processus peuvent exécuter simultanément des copies (ou instances) d'un même programme.
    - Plusieurs processus peuvent exécuter simultanément la même copie d'un même programme : on parle alors de *réentrance*.

Le partage des ressources entre tâches concurrentes se fait sous les contraintes suivantes :

1- les offrir dans un délai raisonnable (éviter la famine),
2- optimiser l'utilisation de ces ressources,
3- exclusion mutuelle sur les ressources non partageables,
4- prévenir les blocages

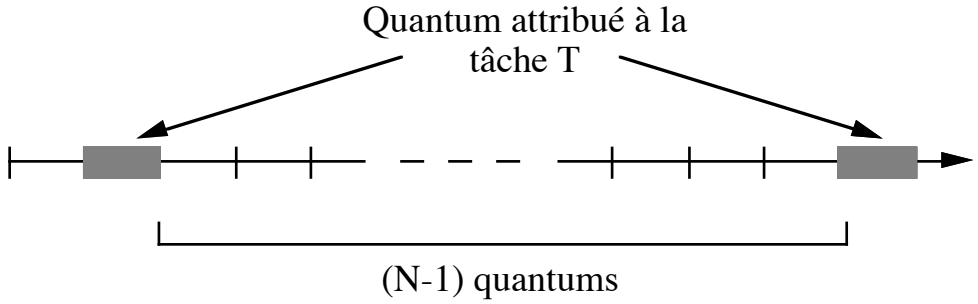
L'exploitation temps partagé privilégie le point 1, ce qui peut être contradictoire avec 2. Le système se dote d'une politique d'allocation des ressources : pour chaque type de ressource un algorithme s'efforce de résoudre les conflits d'accès.

Le processeur est alloué aux travaux par unités de temps appelées quantums. Si il y a N travaux en cours et si un quantum vaut q millisecondes, chaque travail se verra attribuer un quantum tout les (N-1)q millisecondes. Le temps de réponse reste agréable tant que N n'est pas trop grand !

Chaque utilisateur dispose de sa propre machine virtuelle qu'il personnalise en définissant un environnement de travail (fichiers de configuration, *look and feel* des systèmes de fenêtrage).

Modes d'exploitation : le temps partagé

- A chaque reprise de son exécution, une tâche dispose d'un quantum de temps q :



- N utilisateurs --› N machines virtuelles,
- Critère : temps de réponse "court",
- Gestion des ressources sophistiquée : mémoire virtuelle, accès au processeur en fonction de priorités variables, optimisation des accès disques, etc.
- Rôle de plus en plus important des interfaces : multifenêtrage, graphique,...

Plusieurs processus peuvent se trouver simultanément en cours d'exécution (multiprogrammation et temps partagé), si un système informatique ne comporte qu'un seul processeur, alors, à un instant donné, un seul processus aura accès à ce processeur. En conséquence, un programme en exécution peut avoir plusieurs *états*.

Note : la multiprogrammation a été introduite en partie pour éviter qu'un programme en attente de fin d'entrées-sorties ne monopolise le processeur.

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 :
    - Etat *actif* ou *élu* (*running*): le processus utilise le processeur.
    - Etat *prêt* ou *éligible* (*ready*: le processus pourrait utiliser le processeur si il était libre (et si c'était son tour).
    - Etat *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 (par exemple l'attente de la frappe d'un caractère au clavier lors d'un `scanf`) 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, ...)

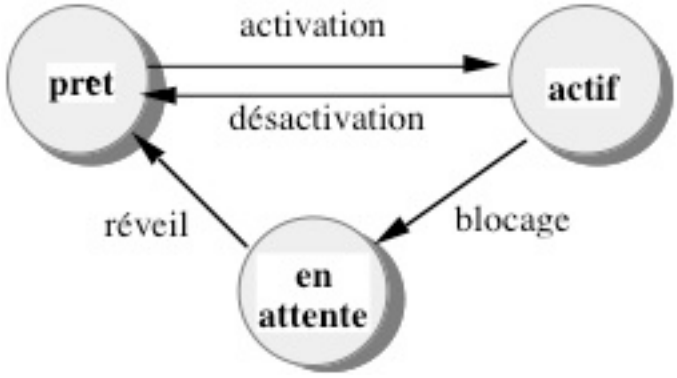
La sortie de l'état actif pour passer à l'état prêt se produit dans le cas des ordonnancements préemptifs lorsqu'un processus plus prioritaire que le processus actif courant devient prêt.

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. On peut citer : opération P sur un sémaphore dont le compteur est négatif, E/S bloquante du type scanf, opérations sleep ou wait sous UNIX).

Transitions entre états

- 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 qu'un utilisateur de S.E. peut effectuer sur les processus sont :
  - Création (ex : `fork`, création d'un processus sous UNIX),
  - Destruction (ex : `kill -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.

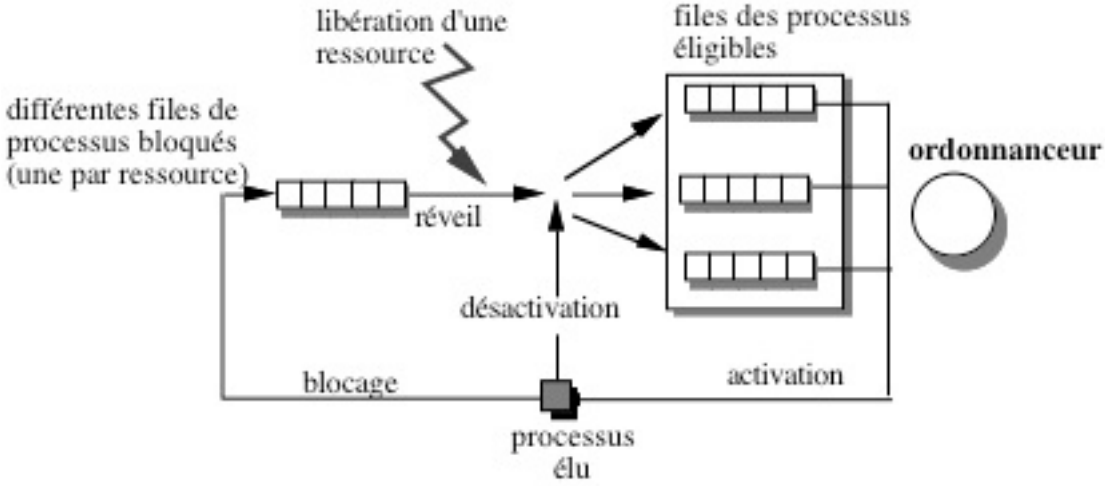
Afin d'éviter qu'un processus accapare l'unité centrale, le système d'exploitation déclenche un temporisateur à chaque fois qu'il alloue l'unité centrale à un processus. Quand ce temporisateur expire, si le processus occupe toujours l'unité centrale (il peut avoir été interrompu par un événement extérieur 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).

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 différentes politiques d'allocation :
  - avec ou sans priorité
    - sans : premier arrivé premier servi (*first come, first served* : *FCFS*)
    - avec :
      - la priorité peut être fixe ou dynamique
      - il peut y avoir préemption, ou non

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.

La plupart des processeurs proposent des instructions spéciales de changement de contexte (cf. machines Motorola, Intel, ...).

Le *bloc de contrôle d'un processus* (en anglais *Process Control Block* ou *PCB*) est une structure de données qui contient toutes les informations relatives au contexte d'un processus. Quand un processus est désactivé ou bloqué, toutes les information relatives à son contexte son sauvegardées dans son bloc de contrôle. Quand un processus est activé, on restaure son contexte à partir de son bloc de contrôle.

Contexte d'un processus

- Par *contexte*, on entend toutes les informations relatives à l'exécution d'un processus, telles que :
  - les registres :
    - le compteur ordinal, le pointeur de pile, le registre de drapeaux,
    - les registres généraux
  - les ressources utilisées et les quotas associés :
    - fichiers ouverts
    - espace mémoire alloué

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 :

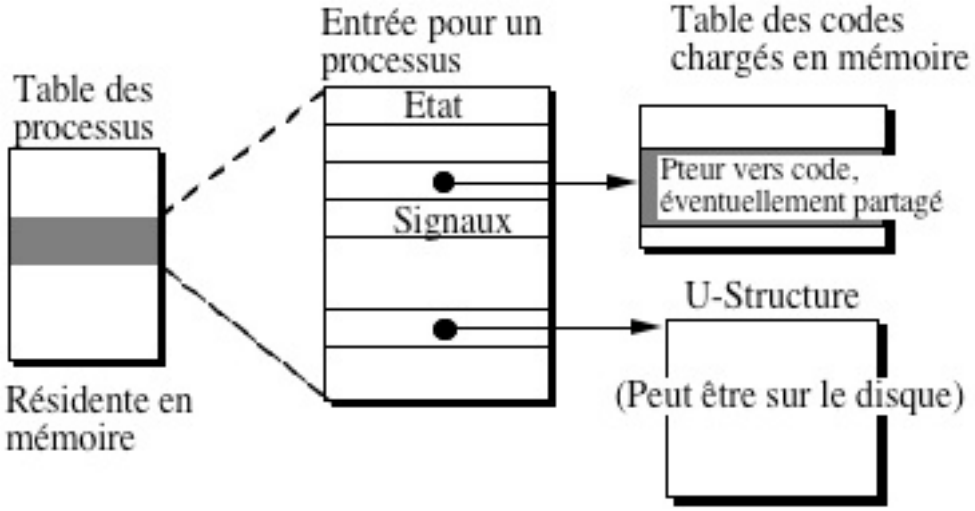
- identificateur du processus (pid)
- configuration des registres (pile, compteur ordinal),
- répertoire courant, fichiers ouverts,

Remarque :

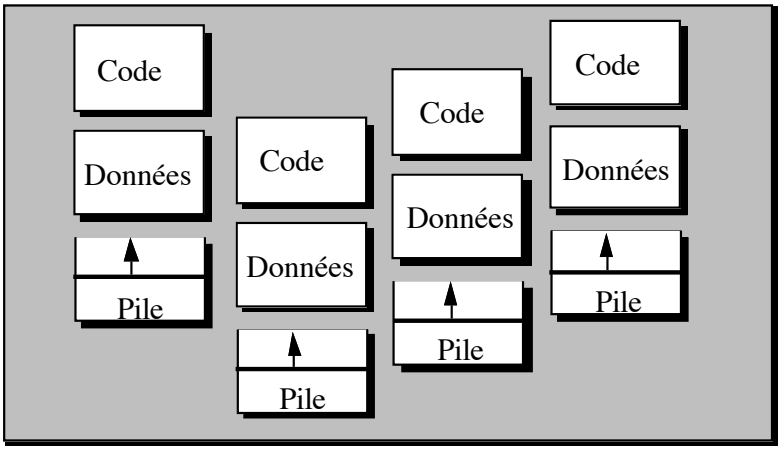
Le pointeur vers la table des *textes* (fichiers exécutables !) donne, non pas le début du programme à exécuter mais un pointeur vers ce programme, ce qui permet à la fois le partage de code exécutable par plusieurs processus différents et le déplacement de celui-ci par le système.

Processus sous UNIX

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



- Unix alloue 3 régions en mémoire pour chaque processus :



- **Remarque** : gestion des processus et gestion mémoire sont étroitement liées.

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.

Création de processus sous Unix

• Exemple de programme utilisant `fork` :

```
int main (){
  int f;
  •••;
  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")
     •••;
   }
}
```

• La primitive **`fork`** permet de créer un processus identique au créateur par copie de ses 3 régions. Le créateur est appelé *père (parent process),* et le créé est appelé *fils (child process)*.

On illustre ci-contre ce qui se passe en mémoire lors de l'appel à la fonction fork. 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).

Voici un exemple d'exécution d'un programme mettant en évidence ce mécanisme :

```
$ cat exo1.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main () {
  int f;
  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 exo1.c -o exo1
```

Voici quelques résultats d'exécution :

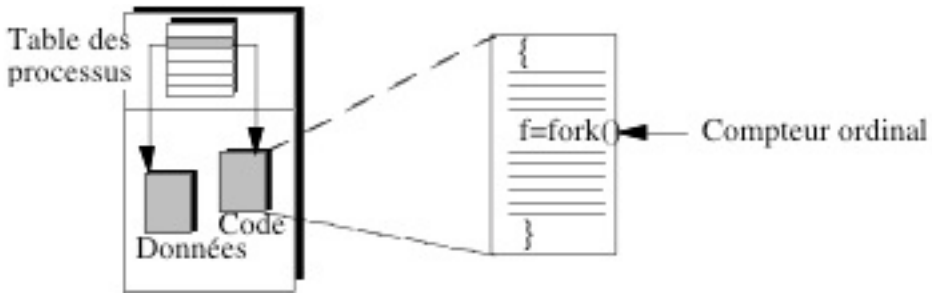
```
$ exo1
 Valeur retournee par la fonction fork: 717
 Je suis le processus numero 716
 Valeur retournee par la fonction fork: 0
 Je suis le processus numero 717

$ exo1
 Valeur retournee par la fonction fork: 899
 Valeur retournee par la fonction fork: 0
 Je suis le processus numero 898
 Je suis le processus numero 899
```

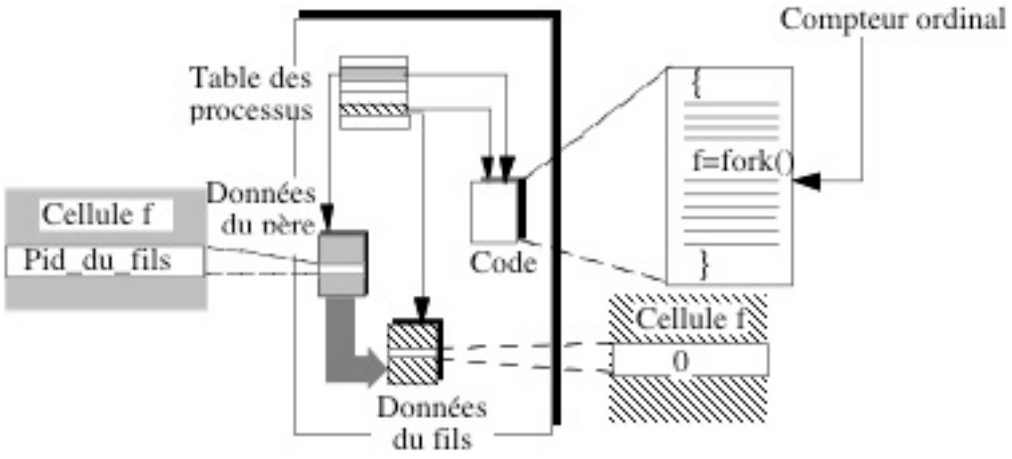
Les "`pid`", la filiation des processus, etc peuvent être observés en utilisant la commande `ps`.

Gestion mémoire lors de `fork` :

• Avant fork :



• Après fork :



On donne ici le schéma fonctionnel décrivant ce qui se passe lors de l'appel aux fonctions fork, exec, wait et exit.

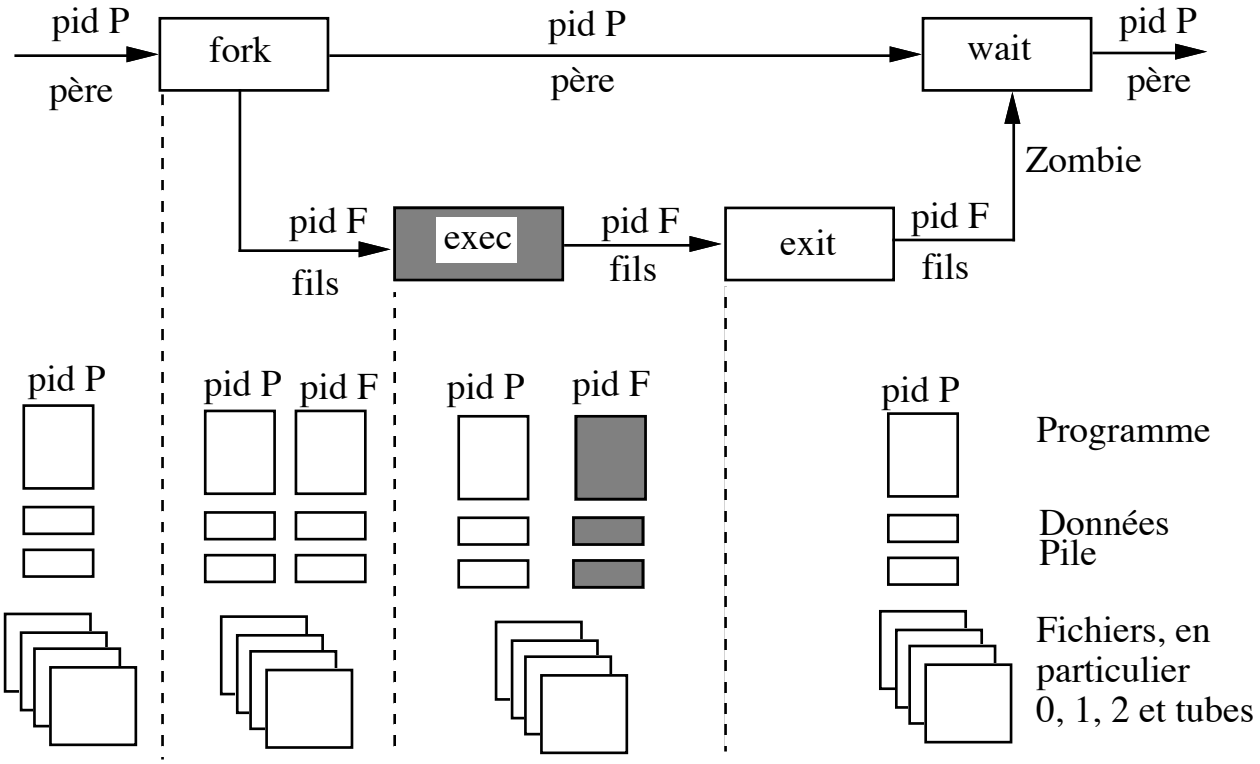
Cet enchaînement se retrouve lors de l'appel à une commande du shell, et presque toutes les applications multitâches sous UNIX utilisent ce schéma.

Ce schéma est purement fonctionnel, dans l'implantation les zones de code ne sont pas dupliquées, mais partagées tant que l'appel à `exec` n'a pas été fait. Les régions data et stack sont allouées dynamiquement, à la demande.

A l'exécution de la fonction `exec` la région de code est remplacée par le fichier fichier et la pile et les données sont réinitialisées

Création de processus sous Unix

• Mécanisme "`fork-exec`" :



Remarque : le segment de code n'est pas dupliqué, mais partagé : deux pointeurs différents désignent le même code exécutable.

Pas de mémoire commune -> les "IPC System V" offrent mémoire partagée et sémaphores sous Unix (pas vus en BCI).

On donne ci-contre un exemple proposé en TP.

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.

L'exécutable s'appelle `fexec`.
On va exécuter la commande `fexec exo1`.

`exo1` est le fichier exécutable correspondant au fichier source (appelé `exo1.c`) vu précédemment :

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

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

Création de processus sous Unix : exemple.

```
int main (int argc, char *argv[]){
int Pid, Fils ;
•••
  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 exo1
 Je suis le pid 5195 je vais faire fork
 Pere numero 5195 attend
  Coucou ! je suis le fils 5196
 5196 : Code  remplace par exo1
 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
```

La Gestion des Processus

1. Définitions, exemples

+ **Politiques d'allocation du processeur**

3. Synchronisation 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 (cf. schéma ci-contre).

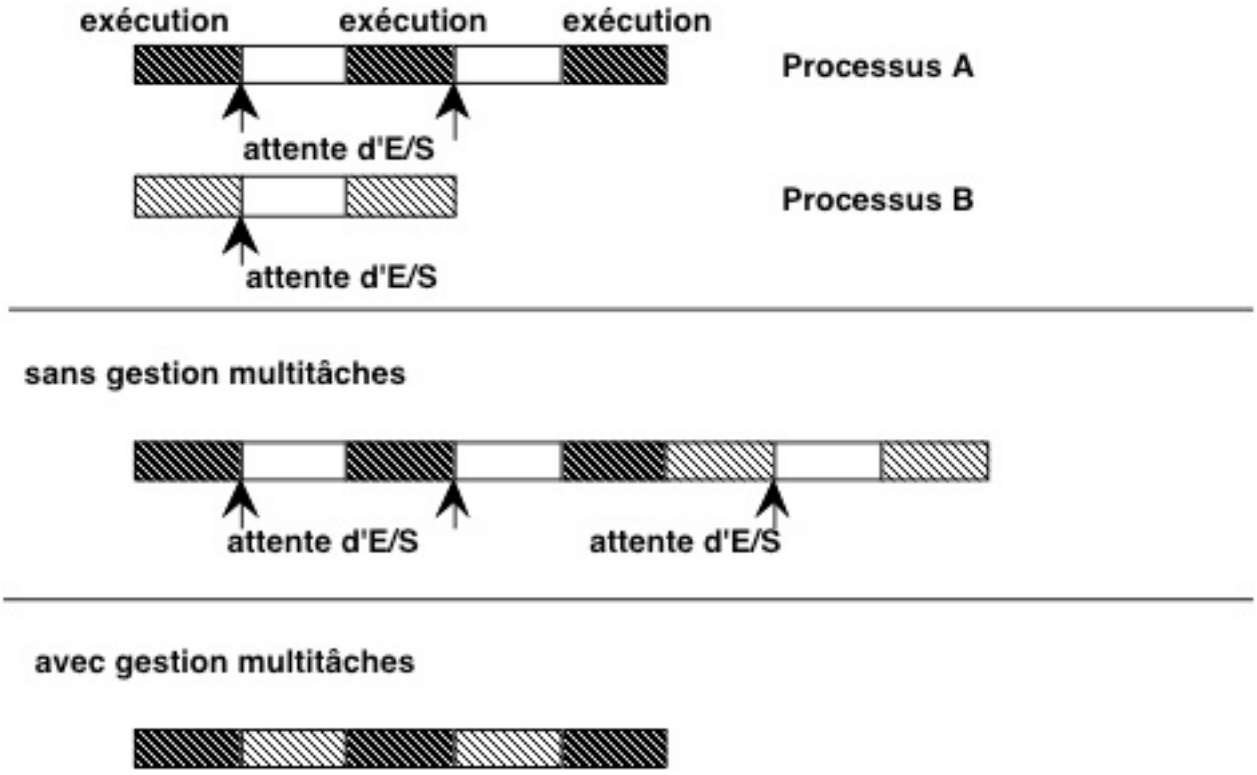
L'exécution de ces deux processus sans multiprogrammation prend 8 unités de temps. Avec multiprogrammation, elle ne prend plus que 5 unités de temps (soit la durée d'exécution du processus A), cf. schéma ci-contre.

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.

Intérêt du multitâches

- Exemple d'utilisation de la ressource processeur par deux processus A et B



- Problème : quelle politique d'allocation du processeur au processus (ou **ordonnacement ou *scheduling***) choisir pour utiliser le processeur "au mieux" ?

Le *temps de réponse* (en anglais *response time*) est le temps qui s'écoule entre la soumission d'une requête et la première réponse obtenue. On utilise en général un *temps moyen de réponse* calculé pour tous les processus mis en jeu pendant une période d'observation donnée. Le temps de réponse est utile dans l'évaluation des systèmes de type transactionnel où il est intéressant de mesurer le temps qui s'écoule entre le moment où l'utilisateur formule une requête, et celui où la réponse lui parvient.

Le *temps de service (turnaround time)* est le temps qui s'écoule entre le moment où un processus devient prêt à s'exécuter et le moment où il finit de s'exécuter (temps d'accès mémoire + temps d'attente dans la file des processus éligibles + temps d'exécution dans l'unité centrale + temps d'attente + temps d'exécution dans les périphériques d'entrée/sortie). On utilise en général un *temps moyen de service* calculé pour tous les processus mis en jeu pendant une période d'observation donnée.

Le *temps d'attente* (en anglais *waiting time*) est le temps passé dans la file des processus éligibles. On utilise en général un *temps moyen d'attente* calculé pour tous les processus mis en jeu pendant une période d'observation donnée.

Le *rendement* d'un système (en anglais *throughput*) est le nombre de processus exécutés par unité de temps.

Critères d'évaluation de performances

- *Temps de réponse (response time):* Temps qui s'écoule entre la soumission d'une requête et la première réponse obtenue.

- *Temps de service (turnaround time):* Temps qui s'écoule entre le moment où un travail est soumis et où il est exécuté (temps d'accès mémoire + temps d'attente en file des éligibles + temps d'exécution dans le processeur et E/S).

- *Temps d'attente (waiting time) :* Temps passé dans la file des processus éligibles.

- *Rendement (throughput) :* Nombre de travaux exécutés par unité de temps.

First Come First Served (FCFS)
Premier Arrivé Premier Servi (PAPS)

- L'ordre d'exécution est identique à l'ordre d'arrivée dans la file des processus éligibles.
- Exemple :
  - On a trois processus 1, 2, 3 à exécuter de durées d'exécution respectives 12, 2 et 2 unités de temps.

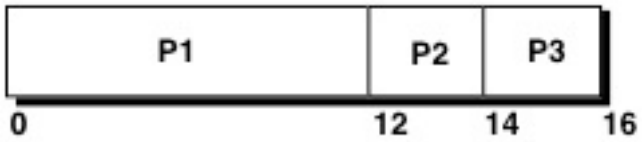


Diagramme de Gant, algorithme PAPS

  - Si l'ordre d'arrivée est 1, 2, 3, les temps de service respectifs sont 12, 14, et 16 unités de temps, et le temps de service moyen est de 14 unités de temps.
  - Mais si l'ordre d'arrivée est 2,3,1, le temps de service moyen est : (2+4+16) / 3 = 7,3

L'exemple traité ci-dessus, suggère qu'il est plus efficace d'exécuter les processus qui ont le temps d'exécution le plus court, d'abord. C'est le principe de l'algorithme PCA.

Cet algorithme donne des performances intéressantes. Le problème est de prédire la durée d'exécution des processus a priori. Pour y remédier, on peut faire de la prédiction sur la durée d'exécution des processus. On peut aussi demander à l'utilisateur du système d'accompagner la demande d'exécution de chaque programme d'une durée maximum d'exécution autorisée (c'est fait dans beaucoup de systèmes), et utiliser cette durée comme durée d'exécution.

L'autre problème est qu'il y a un risque de famine (les processus de longue durée peuvent n'avoir jamais accès à l'unité centrale si des processus de courte durée arrivent en permanence).

Le Plus Court d'Abord (PCA)
Shortest Job First (SJF)

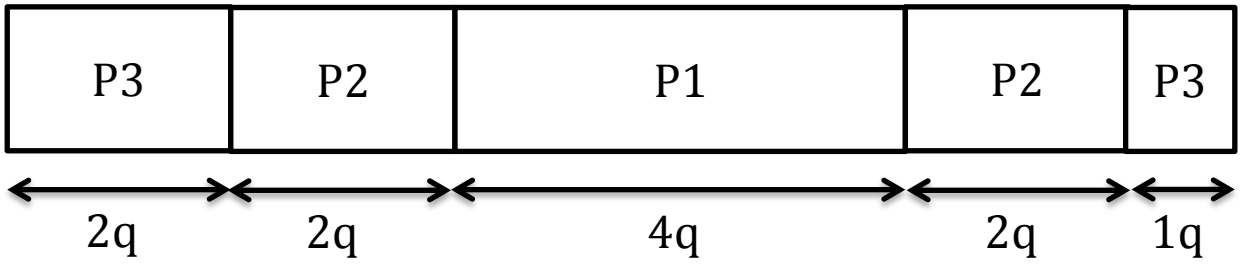
- On exécute d'abord les processus qui ont le temps d'exécution le plus court.
- Problème : comment prédire la durée d'exécution des processus a priori ?
- Solutions :
  - faire de la prédiction sur la durée d'exécution des processus.
  - faire accompagner la demande d'exécution de chaque programme d'une durée maximum d'exécution autorisée qui est utilisée comme durée d'exécution.
- Autre problème : risque de *famine* (les processus de longue durée peuvent n'avoir jamais accès au processeur si des processus de courte durée arrivent en permanence).

Le principe de l'algorithme de priorité est d'associer une priorité à chaque processus. On exécute alors les processus de forte priorité avant ceux de plus basse priorité. On peut remarquer que si l'on alloue à chaque processus une priorité inversement proportionnelle à sa durée, on retrouve l'algorithme PCA.

Le problème de l'algorithme de priorité est qu'il y a risque de famine (les processus de faible priorité pouvant n'avoir jamais accès à l'unité centrale si des processus de forte priorité arrivent en permanence). Pour y remédier, on peut utiliser une technique dite de *vieillissement* (en anglais *aging*). Le principe est d'accroître la priorité des processus à intervalles de temps réguliers, jusqu'à ce qu'ils accèdent à l'unité centrale.

Priorité

- On associe une priorité à chaque processus. Les processus sont exécutés par ordre de priorité.
- Exemple : P1, P2, P3, avec Prio(P1)>Prio(P2)>Prio(P3). Date de début de P1 : 4, pour P2 : 2, pour P3 : 0. Durée de P1 : 4 quantum, P2 : 4 q, P3 : 3q



- Tps de service moyen : (11+4+8)/3=7,67
- Problème : risque de famine.
- Solution : utiliser une technique dite de *vieillissement* (*aging*) qui augmente périodiquement la priorité des processus.
- Intérêt : certains processus système doivent être exécutés en priorité.

Cet algorithme a été conçu pour les systèmes *temps partagé*, afin d'obtenir une répartition équitable de l'unité centrale entre les différents processus présents en mémoire. Son principe est le suivant : 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.

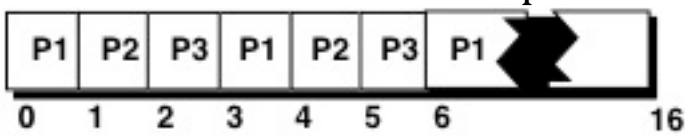
En pratique, la file des processus éligibles est une FIFO. Les nouveaux processus sont placés à la fin de la FIFO. Le système sélectionne le premier processus de la file et arme un temporisateur initialisé à la durée d'un quantum. Le processus peut quitter l'unité centrale avant échéance du temporisateur s'il est terminé ou en attente d'événement. Sinon, à échéance du temporisateur le système reçoit une interruption et la traite en sauvegardant le contexte du processus et en le plaçant à la fin de la file des processus éligibles.

La qualité de cet algorithme va cependant beaucoup dépendre de la taille choisie pour le quantum : elle doit être grande par rapport au temps nécessaire pour le changement de contexte des processus.

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

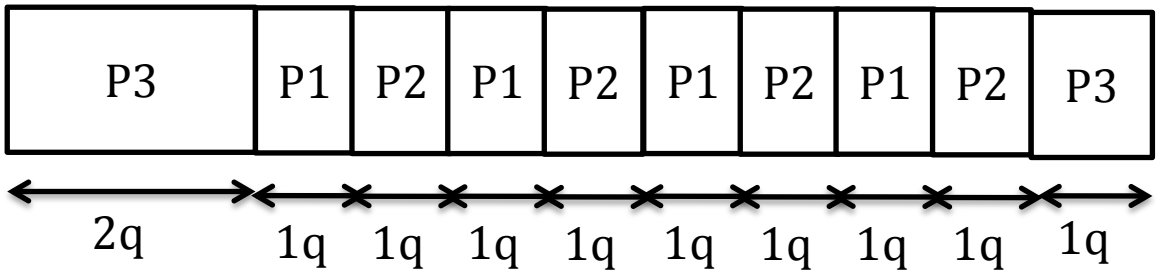


  - Temps moyen de service : 9 = (5+6+16)/ 3, si tous sont prêts au temps 0. (Meilleur que celui donné par l'algorithme FCFS pour le même ordre d'arrivé.)
  - Performances sensibles à la taille choisie pour le quantum.

Algorithme du tourniquet par niveau de priorité (Round Robin within priority)

- Le temps est découpé en quantum, mais les processus sont d'abord élus en fonction de leur priorité ; round robin est utilisé pour des processus de même priorité.

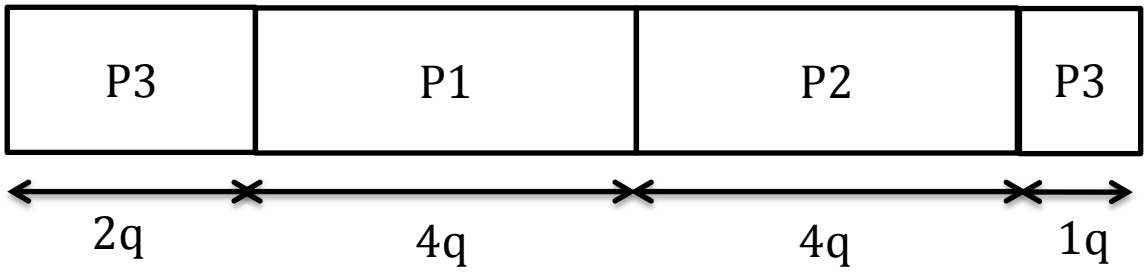
- Exemple : P1, P2, P3, avec Prio(P1)=Prio(P2)>Prio(P3). Date de début de P1 : 2, pour P2 : 2, pour P3 : 0. Durée de P1 : 4 quantum, P2 : 4 q, P3 : 3q.



- Temps de service moyen : (11+8+8)/3=9

Algorithme PAPS par niveau de priorité (FCFS within priority)

- Les processus sont d'abord élus en fonction de leur priorité, puis de leur ordre d'arrivé lorsque des processus ont la même priorité.
- Exemple : P1, P2, P3, avec Prio(P1)=Prio(P2)>Prio(P3). Date de début de P1 : 2, pour P2 : 2, pour P3 : 0. Durée de P1 : 4 quantum, P2 : 4 q, P3 : 3q. On suppose que P1 est arrivé avant P2.



- Tps de service : (11+4+8) = 7,67

Un processus utilisateur démarre avec une priorité de base P_{Base} (P_{Base} = 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 :

$$C = \text{temps cpu consommé}$$

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

$$C = C / 2$$

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

$$P = P_{Base} + C$$

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

$$P = P_{Base} + C + (nice - 20)$$

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

$$P = P_{Base} + f(\text{cpu consommé, temps d'attente}) + (nice - 20)$$

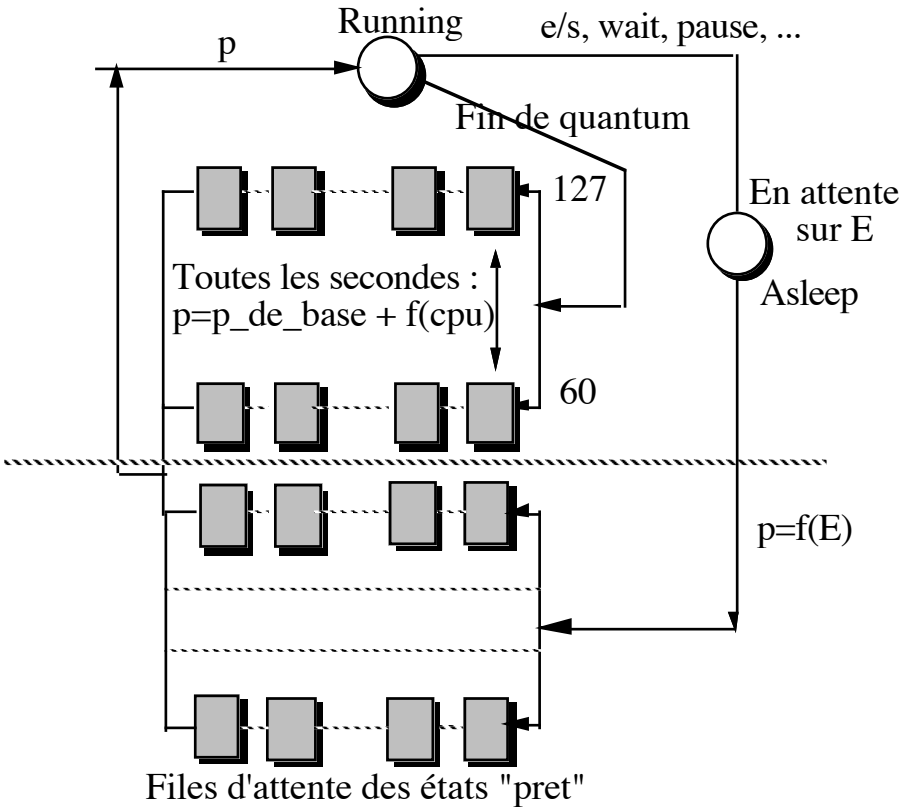
Lors de son réveil après attente sur un événement E (e/s, pause, ...) un processus se voit attribuer une priorité P = f(E), avec interruption éventuelle du processus actif courant, si ce dernier est moins prioritaire. La priorité d'un processus en attente sur un événement dépend donc uniquement du type 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.

Algorithme implanté sous UNIX (tourniquet par priorité avec priorité dynamique)

• Les priorités sont calculées de deux façons différentes:

- Si un processus est arrêté par un événement E (entrée-sortie, fonctions wait, pause ...), il est réactivé avec une priorité qui dépend du type de cet événement.

- La priorité des processus prêts est directement liée à la consommation de temps cpu et au temps qui s'est écoulé depuis leur dernier accès au processeur.



La Gestion des Processus

1. Définitions, exemples

2. Politiques d'allocation du processeur

+ **Synchronisation des processus**

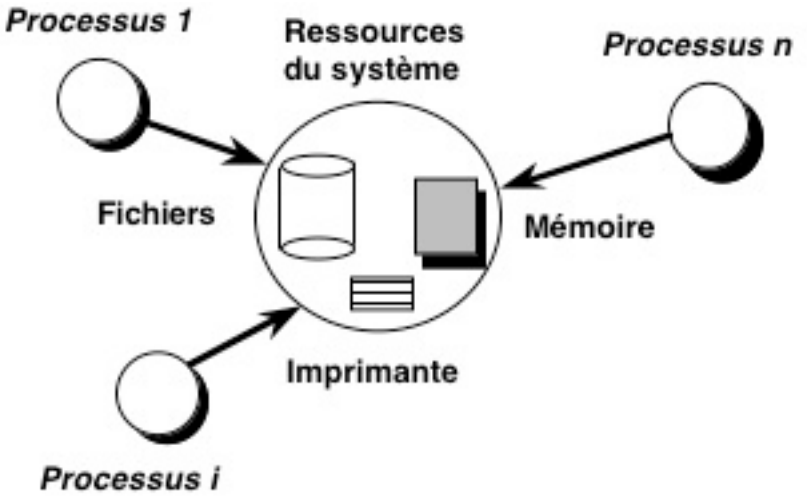
Le nombre de ressources d'un système d'exploitation est limité, il n'offre aux utilisateurs, c'est-à-dire aux des processus, qu'un nombre fini de :

- périphériques d'entrées-sorties : terminaux, imprimantes,
- moyens de mémorisation (disques, mémoire centrale),
- moyens logiciels (fichiers, base de données, ...),

Les ressources allouées à un seul processus, comme l'écran ou le clavier ne posent pas de problème de gestion. Par contre, sera beaucoup plus délicate celle des ressources partagées entre plusieurs processus. Dans ce cas, les processus vont se trouver en situation de *concurrence* (*race*) vis-à-vis des ressources offertes par le système d'exploitation, et se posera alors le problème de la *synchronisation* des actions des différents processus sur les ressources partagées. 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 la séquence d'instructions composant la section critique).

Pourquoi synchroniser les processus ?

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



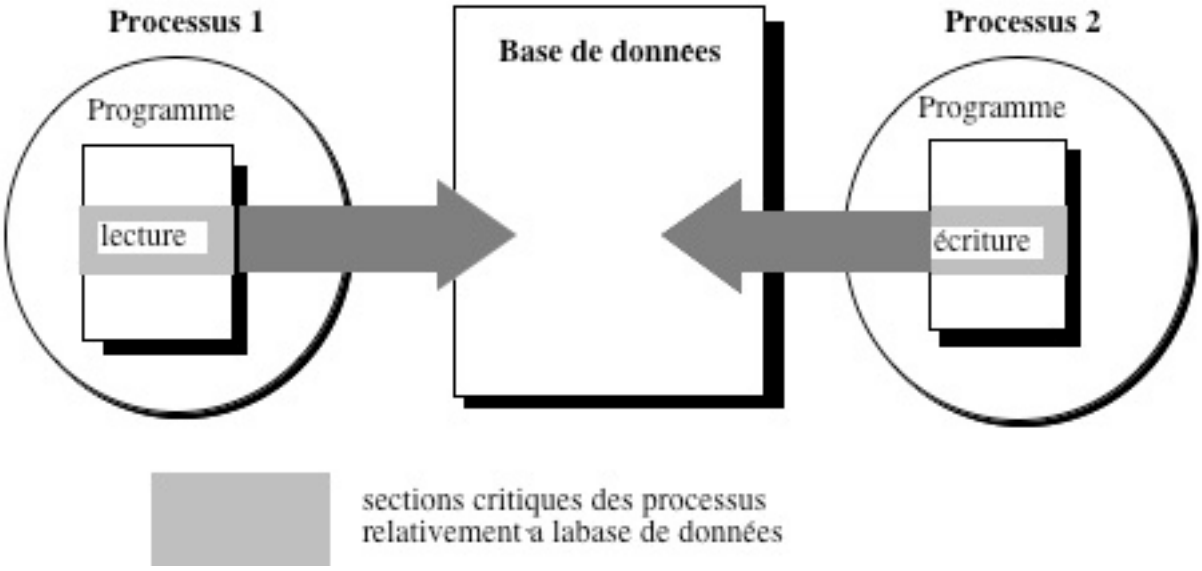
- Ils sont alors en situation de *concurrence* (*race*) vis-à-vis des ressources. Il faut *synchroniser* leurs actions sur les ressources partagées.

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.

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.



Comment les processus vont-ils synchroniser leurs accès respectifs aux sections critiques ?

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).

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.

Pourquoi ces outils de synchronisation ?

• Exemple de solution naive : on considère deux processus P_0 et P_1

• On va utiliser une variable partagée `x` indiquant lequel des deux processus peut entrer en section critique, elle est initialisée à 0 ou 1.

Contrôle d'accès à la section critique pour P_i (i vaut 0 ou 1)

```
...
...
// Pi doit attendre son tour
while (x != i) {}
// Debut de section critique
...
...
// Fin de section critique :
// donner l acces  à l'autre processus.
x = (i+1)%2;
...
```

Problème : Les processus ne peuvent entrer en S.C. qu'à tour de rôle, `tour` impose un ordre (pas d'accès équitable).

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 :

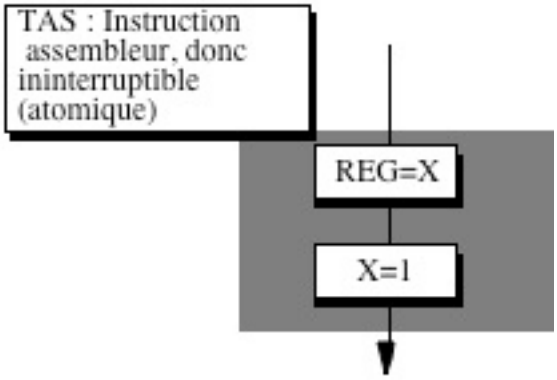
- l'*attente active* (busy waiting) : un processus qui trouve la variable de synchronisation à 1 continuera à la tester jusqu'à ce qu'elle passe à 0. Il reste donc actif, alors qu'il pourrait être suspendu et rangé dans une file d'attente.
- le fait qu'on ne sait pas qui va prendre la main. Ce problème 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.

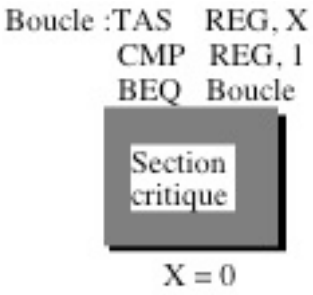
Sur les processeurs SPARC, qui équipent les stations Sun, l'instruction de Test and Set s'appelle `ldstub (load store unsigned byte...)`.

Test And Set

- L'instruction du langage machine dont le nom générique est TAS utilise un registre et une case mémoire. Quand X vaut 1, cela veut dire que la ressource est prise.
- Elle fait deux opérations de façon atomique (l'une est toujours faite avec l'autre) : ranger la valeur actuelle de X dans le registre et mettre X à 1.
- on la note : `TAS REG, X`.



- Utilisation (X est initialisée à zéro) :



Dans l'exemple ci-contre deux processus déroulent le même programme.

Remarque importante :

Autant de quanta passés dans sa section critique par le processus qui a pu y entrer, **autant de quanta passés dans la boucle sur TAS par ceux qui essaient d'entrer dans la leur.** C'est ce qu'on appelle l'attente active.

Soit le scénario suivant :

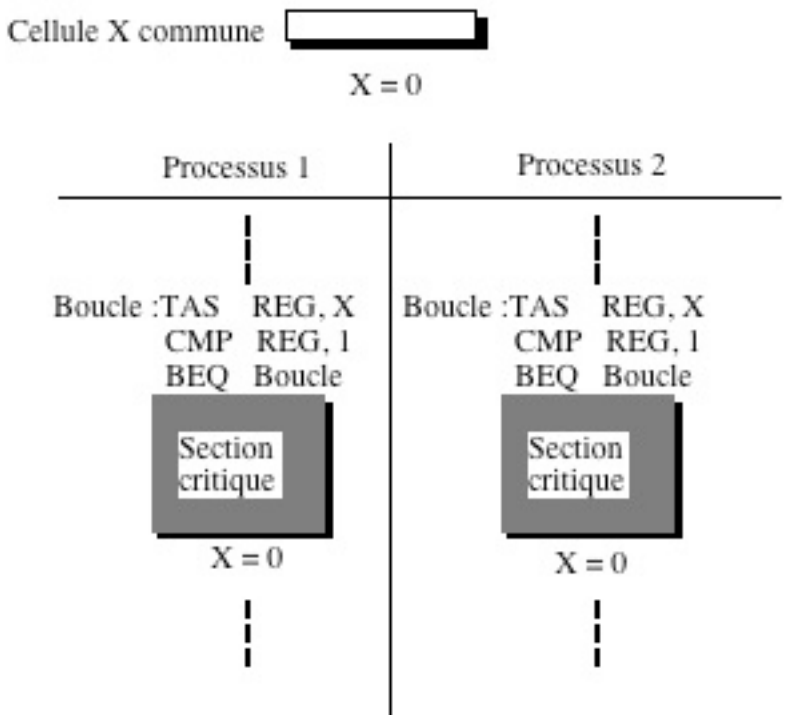
- P1 démarre le premier, c'est-à-dire le S.E lui attribue d'abord le quantum de temps.
- pendant ce quantum, il rentre dans la section critique après avoir fait TAS, X vaut donc **1**.
- la fin de quantum intervient alors que P1 est dans sa section critique
- le S.E attribue un quantum à P2 qui arrive sur TAS et trouve X à un. Il va donc **épuiser le reste du quantum** dans la boucle de l'instruction TAS,
- le S.E redonne le quantum à P1 qui le passe en entier dans sa section critique
- P2 va donc encore **épuiser un autre quantum** dans la boucle de l'instruction TAS,
- le S.E redonne le quantum à P1 qui sort de sa section critique, met **X à 0** et termine son quantum,
- P2 se voit attribuer un quantum et sort de TAS puisque X vaut 0, et il fait passer **X à 1**, il rentre dans sa section critique, fin de son quantum
- si P1 veut maintenant rentrer dans sa section critique, il va être bloqué sur TAS comme l'a été P2.

En fait il s'agit d'une utilisation totalement inefficace du quantum, un gaspillage de la ressource processeur, la plus précieuse de la machine. Nous verrons que les sémaphores permettent d'éviter l'attente active.

D'autre part, il n'est pas sûr que l'accès à la ressource se fera dans l'ordre ou les processus ont été bloqués sur le TAS. Cet accès dépend de l'obtention d'un quantum juste après que X soit revenu à 0.

Exemple d'utilisation de TAS

- 2 processus P1 et P2 exécutent des programmes contenant des sections de code critiques vis-à-vis de l'accès à une certaine ressource.



- Inconvénients de cette technique :
  - *Attente active* (busy waiting)
  - On ne sait pas qui va prendre la main.

Pseudo code (C) du TAS

- Attention, le code ci-dessous à vocation à vous faire comprendre le fonctionnement de TAS ; il n'est en aucun cas l'implémentation réelle de TAS…
- Le test and set en lui même :

```
#define LOCKED 1

int TestAndSet(int* lockPtr) {
    int oldValue;

    // Start of atomic segment
    // The following statements should
    // be interpreted as pseudocode for
    // illustrative purposes only.
    oldValue = *lockPtr;
    *lockPtr = LOCKED;
    // End of atomic segment
    return oldValue;
}
```

- Usage du test and set pour implanter une section critique :

```
volatile int lock = 0; // shared variable

void Critical() {
    while (TestAndSet(&lock) == 1);
    // critical section
    // only one process can be in this section at a time
    lock = 0 // release lock when finished with the critical
             // section
}
```

Les opérations Init, P et 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.

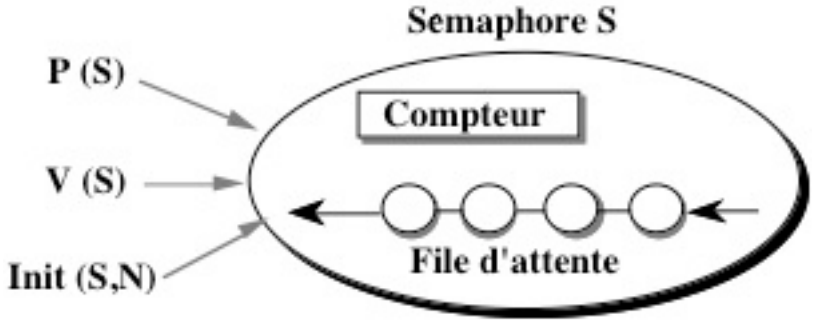
2- sur les états des processus :

- 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.**

SEMAPHORES [Dijkstra 1968]

• 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 (pseudo-code)

- Le système d'exploitation garantit l'atomicité des ces fonctions **:** du point de vue des processus impliqués, elles s'exécutent sans être interrompues. Attention, ce qui suit est du pseudo-code (présente les principe d'implémentation mais n'est pas une réalisation concrète du mécanisme).

```
/******************    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
     }
}
```

Cet exemple reprend celui du TAS, mais cette fois, la synchronisation est faite par sémaphore. On reprend le même scénario, c'est-à-dire que P1 démarre le premier :

- pendant son quantum, il fait P(S), le compteur du sémaphore passe à **zéro**, ce qui n'est pas négatif et P1 peut donc rentrer dans sa section critique,
- la fin de quantum intervient alors que P1 est dans sa section critique
- le S.E attribue un quantum à P2 qui arrive sur P(S) et trouve le compteur à zéro. Ce compteur passe donc à -1, le processus P2 est mis dans **l'état bloqué et rangé dans la file d'attente de S**. Le système choisit alors le processus prêt le plus prioritaire et lui attribue un quantum. Il n'y a **pas** gaspillage de la partie restante du quantum.
- le S.E redonnera un quantum à P1 qui le passe en entier dans sa section critique,
- P2 ne se voit pas proposer de quantum, puisqu'**il n'est pas dans la file d'attente des prêts, mais dans l'état bloqué et dans la file d'attente de S**. Il n'y a plus attente active.
- le S.E redonne le quantum à P1 qui sort de sa section critique et fait V(S). Le compteur de S passe à zéro, on **sort P2 de la file d'attente** et on le met dans la file d'attente des prêts. P1 termine son quantum,
- P2 se voit attribuer un quantum de temps et rentre dans sa section critique, fin de son quantum de temps,
- si P1 veut maintenant rentrer dans sa section critique, il va passer dans **l'état bloqué sur P(S)** comme l'avait fait P2.

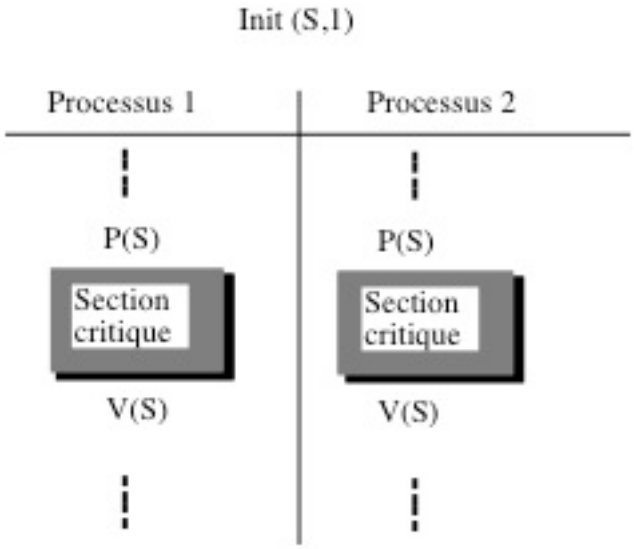
Remarques : il n'y a plus gaspillage de quantums par le processus qui n'a pas accès à sa section critique. Il passe dans l'état bloqué, le S.E ne lui offre donc plus de quantums.

Si P1 et P2 sont seuls sur la machine :

- dans le cas de TAS, P2 ralentit P1 dans sa progression dans sa section critique
- dans le cas sémaphore, P2 est bloqué, les quantums qui lui auraient été attribués sont donnés à P1, qui progresse plus vite, donc P2 sera bloqué moins longtemps !

Verrou

- Définition : un sémaphore dont le compteur est initialisé à un s'appelle un verrou : quand un processus est dans sa section critique, il empêche tous les autres d'entrer dans la leur.



- Remarque : l'exclusion mutuelle est un exemple de compétition entre processus pour l'accès aux ressources, on va voir qu'il peut y avoir des interactions en coopération.

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,

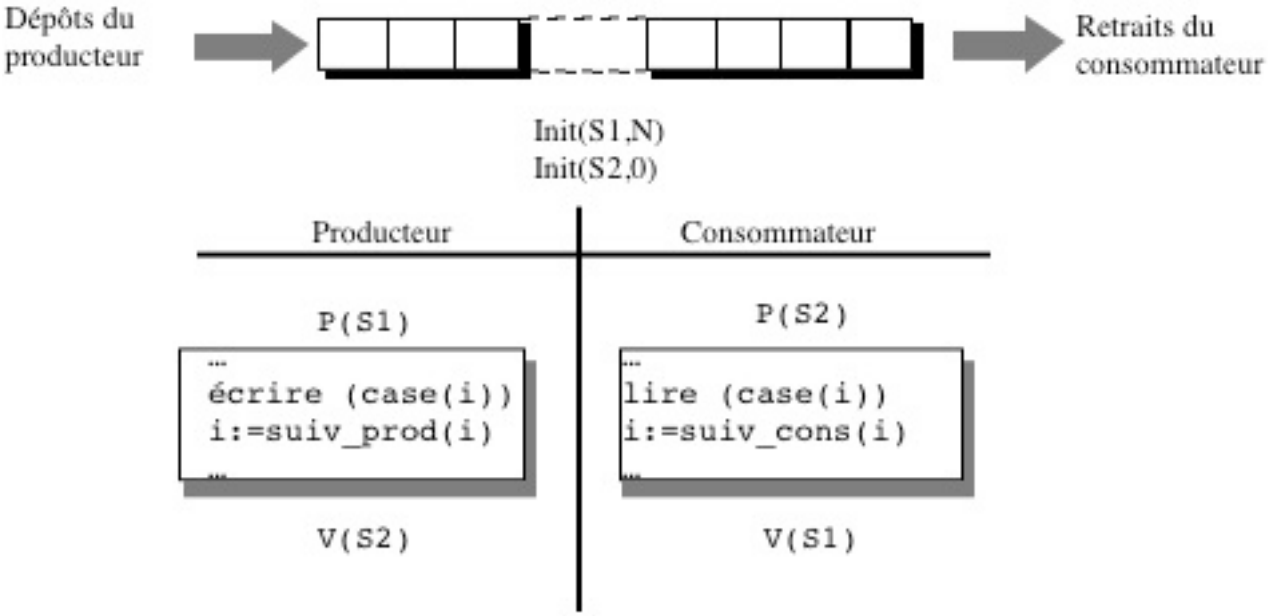
Ce type de synchronisation est mis en œuvre dans le système Unix sous forme de la primitive *pipe*. Celle-ci crée un fichier (appelé pipe, ou *tube* en français) à double accès qui ne peut être lu que si l'on a déjà écrit dedans et dans lequel on ne peut écrire que s'il y a des cases libres.

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*.

Des mécanismes de ce type peuvent être fournis par le langage. On citera Ada et Java qui proposent des outils de synchronisation.

Producteur et consommateur

- Un producteur dépose des messages un par un dans un tampon ; un consommateur retire ces messages un par un. Le producteur et le consommateur doivent synchroniser leurs actions.



- 2 sémaphores sont utilisés :
  - S1 indique les ressources disponibles pour le producteur (nombre de cases vides),
  - S2 indique les ressources disponibles pour le consommateur (nombre de cases pleines).

• Remarques sur le modèle producteur/consommateur:

Les deux processus sont en situation de coopération. Plus les vitesses des processus diffèrent, plus la taille du tampon est grande.

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.

Scénario pour les lecteurs :
Le premier lecteur interdit l'accès aux écrivains, mais permet cet accès à tous les lecteurs qui se présentent. Le dernier libère la ressource.
Scénario pour les écrivains :
Quand un écrivain arrive, il interdit tout autre accès à la ressource, il la libère quand il a terminé les modifications.

```
Init(X, 1)
```

Programme pour les lecteurs	Programme pour les écrivains
`Nb_Lect = Nb_Lect + 1;` `si (Nb_Lect == 1) P(X);` `Consulter` `Nb_Lect = Nb_Lect - 1;` `si (Nb_Lect == 0) V(X);`	`P(X);` `Modifier` `V(X);`

Lecteur et écrivain

- La solution précédente n'est pas correcte : l'accès à la variable **partagée** par les lecteurs `Nb_Lect` n'est pas contrôlé, on ajoute un verrou appelé `XL:`

Programme pour les lecteurs	Programme pour les écrivains
P(XL); ` Nb_Lect = Nb_Lect + 1;` ` si (Nb_Lect == 1) P(X);` **V(XL);** Consulter **P(XL);** ` Nb_Lect = Nb_Lect - 1;` ` si (Nb_Lect == 0) V(X);` **V(XL);**	P(X); Modifier V(X);

- L'inconvénient de cette solution est qu'elle va sans doute provoquer la famine (*starvation*) pour les écrivains : dès qu'un lecteur est arrivé, il permet l'accès à un flux ininterrompu de lecteurs. Un écrivain voit son tour passer après celui du dernier lecteur.

Lecteur et écrivain

- On va modifier le scénario précédent pour qu'un écrivain puisse accéder à la ressource le plus tôt possible, on ajoute un verrou `S:`

Programme pour les lecteurs	Programme pour les écrivains
P(S); /* Si un ecrivain 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 ecrivains suivants */** P(X); Modifier **V(S) /* Permettre l'accès au suivant */** V(X);

- Les lecteurs sont encore prioritaires, mais il n'y a plus famine pour les écrivains : un écrivain peut maintenant interrompre le flot des lecteurs et interdire à ceux qui le suivent de lui prendre son tour.

Rendez-vous

- Un schéma dans lequel on assure que deux processus PA et PB exécutent respectivement les instructions I1 et J1 avant d'exécuter les instructions I2 et J2 s'appelle un rendez-vous :

Processus PA	Processus PB
... I1; Pt de rdv I2; ...	... J1; Pt de rdv J2; ...

Pour assurer ce type de synchronisation, on peut faire :

```
Init (SA, 0);
Init (SB, 0);
```

Processus A	Processus B
I1; P(SB); V(SA); I2;	J1; V(SB); P(SA); J2;

Ou encore :

Processus A	Processus B
I1; V(SA); P(SB); I2;	J1; V(SB); P(SA); J2;

Mais pas (**interblocage** possible) :

Processus A	Processus B
I1; P(SB); V(SA); I2;	J1; P(SA); V(SB); J2;

P1 et P2 sont les **seuls** processus qui utilisent X1 et X2.

Scénario posant un problème :

- P1 fait P(X1), le compteur de X1 passe à zéro, puis il s'arrête (fin de quantum avant P(X2),
- P2 fait P(X2), le compteur de X2 passe à zéro, puis il s'arrête (fin de quantum avant P(X1),
- P1 reprend la main et fait P(X2), le compteur de X2 passe à -1, P1 passe donc bloqué,
- P2 reprend la main et fait P(X1), le compteur de X1 passe à -1, P2 passe donc bloqué,
- P1 et P2 sont tous deux bloqués, ce qui n'est pas grave, et même normal puisque les ressources auxquelles ils veulent faire accès sont déjà prises.

Mais qui peut débloquer P1 ? C'est P2.

Dans quel état est P2 ? Il est bloqué. Par qui ? Par P1, là est le problème : P1 et P2 sont bloqués et ne peuvent être débloqués que l'un par l'autre.

Ils sont interbloqués.

Pour résoudre le problème de l'interblocage on peut demander aux développeurs d'utiliser les sémaphores dans **l'ordre de leur numérotation,** dans notre exemple, le processus 2 est alors bloqué (il doit faire P(X2) APRES P(X1) et il n'y a plus interblocage.

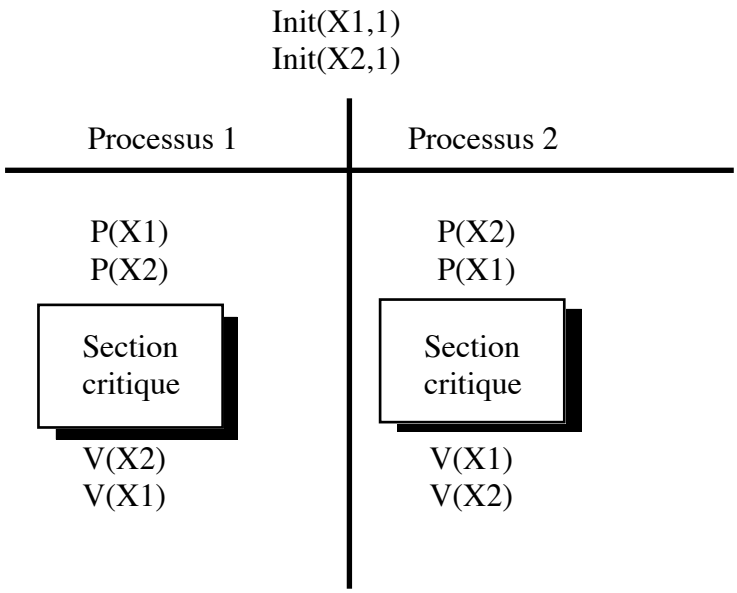
Il faut faire :

Processus P1	Processus P2
P(X1)	P(X1)
P(X2)	P(X2)
Section critique	Section critique
V(X2)	V(X2)
V(X1)	V(X1)

Certains systèmes, comme UNIX, proposent des opérations P et V sur **plusieurs** sémaphores à la fois. On peut ainsi remplacer les deux suites P(X1), P(X2) et P(X1), P(X2) par les deux commandes atomiques P(X1, X2) et V(X1, X2).

Problèmes dans l'utilisation des sémaphores

- **Interblocage** :



- **Famine**, c'est-à-dire l'attente indéfinie dans la file d'attente.

- Attention à la **taille** des sections critiques qui doit être la plus petite possible.

Du bon fonctionnement des outils de gestion de la concurrence

- Deux propriétés à respecter :
  1. sûreté (*safety*), deux processus ne doivent pas se trouver ensemble en section critique. Il faudra assurer l'exclusion mutuelle.
  2. vivacité (*liveness*), qui impose des contraintes de :
     a. progression : il faudra éviter l'interblocage,
     b. équité : pas de famine, attente bornée. Toute demande sera satisfaite, ou encore : tout processus qui veut entre en section critique devra pouvoir le faire
- Les algorithmes sont des compromis entre sûreté et vivacité.

Plan

- Solutions logicielles
  - Divers algorithmes,
  - L'algorithme de Dekker : le première solution,
  - L'algorithme de Peterson : une solution élégante,

Solutions logicielles

Nous allons présenter de façon progressive le premier algorithme résolvant le problème de l'exclusion mutuelle pour deux processus, il s'agit de l'algorithme de Dekker.

Limitations : on suppose que les opérations de lecture, d'écriture et de test d'un mot en mémoire sont atomiques.

Rappel des conditions d'accès à une section critique :

- exclusion mutuelle pour l'accès à la section critique,
- pas d'interblocage,
- pas d'attente infinie pour entrer dans la section critique,
- accès équitable.

Version 1

• On considère deux processus P_i et P_j. On suppose ici :

3. i = 0,

4. j = 1,

• la variable **partagée** `tour` indique lequel des deux processus peut entrer en section critique.

Programme exécuté par le processus P_i
`...` `...` `/* attendre */` `while (tour != i) {}` `/* debut de section critique */` `...` `...` `/* fin de section critique */` `tour = j;` `...`

Problème : les processus ne peuvent entrer en S.C. qu'à tour de rôle, `tour` impose un ordre (pas d'accès équitable).

Version 2

- Il faudrait connaître l'état des processus pour savoir lequel d'entre eux peut entrer en S.C.. On remplace alors la variable unique par un tableau partagé :

`Etat[2]` dont les éléments sont initialisés à zéro , la valeur 1 signifie que P_i **est** en S.C.

Programme exécuté par le processus P_i

```
...
...
/* attendre que l'autre processus sorte. */
while (Etat[j] == 1) {}
Etat[i] = 1;

/* debut de section critique */
...
...
/* fin de section critique */
Etat[i] = 0;
...
...
```

Problème : les deux processus P_i et Pj peuvent être **ensemble** en S.C.

Pour éviter cela, on va positionner `Etat[i]` **avant** d'entrer en S.C..

Version 3

- La valeur 1 dans la case `Etat[i]` signifie que P_i **veut** entrer en S.C..

Programme exécuté par le processus P_i

```
...
...
/* Le processus indique qu'il va entrer en S.C. */
Etat[i] = 1;
/* Si l'autre fait de même, alors attendre */
while(Etat[j] == 1){ }
/* debut section critique */
...
...
/* fin de section critique */
Etat[i] = 0;
...
...
```

Problème :

- L'exclusion mutuelle est garantie, mais **interblocage** possible.

Version 4

Programme exécuté par le processus P_i

```
...
...
/* i veut entrer en S.C. */
Etat[i] = 1;

/* j veut-il entre en section critique ? */
while(Etat[j] == 1) {
     /* oui, le laisser passer */
      Etat[i] =0;
     /* attendre (cf. algo. 2) */
      while (Etat[j] == 1) { };
     /* puis redemander l'accès */
     Etat[i] = 1;
     }

/* debut de section critique */
...
...
/* fin de section critique */
Etat[i] = 0;
...
...
```

Interblocage possible (exécutions pas à pas parallèles et identiques) bien que très peu probable.

Algorithme de Dekker (1)

- On ré-introduit la variable `tour` qui impose un **ordre** pour **briser la symétrie :**

Programme exécuté par le processus P_i

```
...
...
/* i veut entrer en S.C. */
Etat[i] = 1;

while (Etat[j] == 1) {
     if (tour == j) {
          Etat[i] = 0;
          /* attendre que l'autre sorte de SC */
          while(tour == j) { };
      Etat[i] = 1;
     }
}
/* debut de section critique */
...
...
/* fin de section critique */
tour = j;
Etat[i] = 0;
...
...
```

- Ligne ajoutée à la version 4 : `if (tour == j)`

Algorithme de Dekker (2)

• Exclusion mutuelle garantie :

1- P_i ne va en SC que si `Etat[j]=0,`

2- P_i ne teste `Etat[j]` que si `Etat[i]=1`,

donc P_i n'entre en SC qu'avec :

`Etat[i] =1 et Etat[j] =0.`

• Pas d'interblocage :

Si P_i et P_j veulent entrer en SC et que `tour=i` :

1- P_i attend que `Etat[j] = 0`

2- P_j met `Etat[j]` à 0 et attend que `tour` change,

3- P_j entre en SC.

`tour` ne change qu'après passage en section critique.

Algorithme de Peterson (1981)

Programme exécuté par le processus P_i

```
...
/* i veut entrer en S.C. */
Etat[i] = 1;

/* mais il laisse passer j, si j veut entrer */
tour = j;

/* attendre que Pj sorte de SC */
while ((Etat[j] == 1) && (tour == j)) {};
 /*   debut de S.C.   */
...
...
/*   fin de section critique   */
Etat[i] = 0;
...
```

• Exclusion mutuelle garantie, pas d'interblocage et attente limitée, en effet :

- interblocage ssi (ce qui est impossible) :

`(Etat[j] = 1 ET tour = j) ET (Etat[i] = 1 ET tour = i)`

- les deux processus sont ensemble en SC ssi (ce qui est impossible) :

`(Etat[j] = 0 OU tour = i) ET (Etat[i] = 0 OU tour = j)`

• L'initialisation de `tour` est indifférente : suivant l'ordonnancement, `tour` donne le droit d'entrer à l'un ou l'autre des processus.