Université Sultan Moulay Slimane

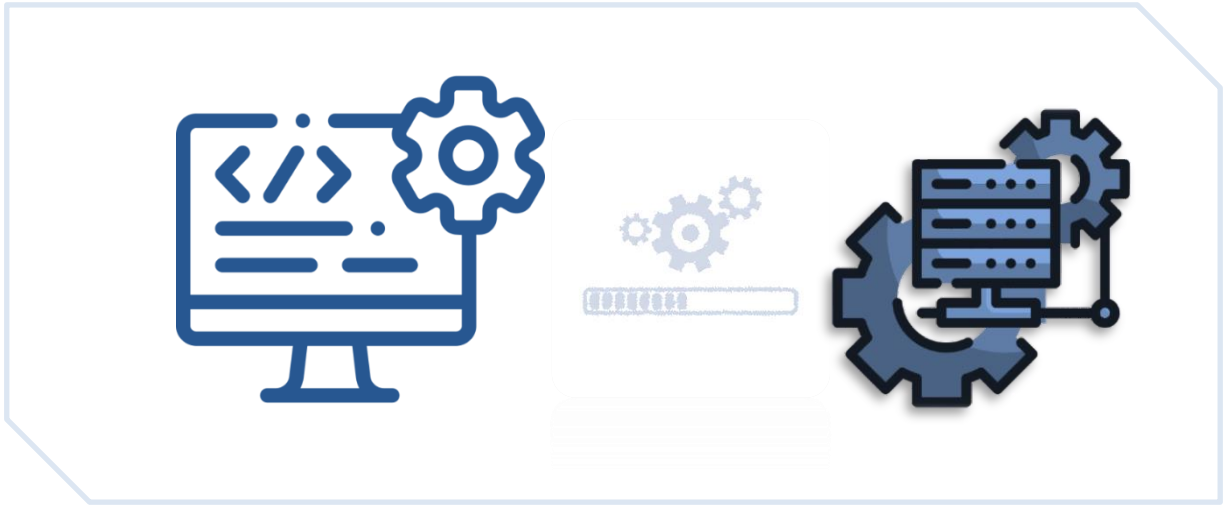
Ecole Supérieure de Technologie de Béni Mellal

Département : ITG
Filière : ARI - *2ème année - S3*
Année universitaire: 2023-2024

Module: Sys. & Réseaux Informatiques Avancés (M11)
Elément: Systèmes d'exploitation avancés (E2)

Support de cours

Systèmes d'exploitation Avancés



Chapitre II: Gestion des processus

Pr. Siham BAKKOURI

Email: s.bakkouri@usms.ma

Plan

- **CONCEPTS DE PROCESSUS ET DE THREADS**
  - → **Définitions et notions sur les processus**
  - → **Etats d'un Processus**
  - → **Bloc de Contrôle de Processus (PCB)**
  - → **Commutation de contexte**
  - → **Concept de Threads**
- **GESTION DES PROCESSUS (*sous linux*)**
- **ORDONNANCEMENT DES PROCESSUS**
- **SYNCHRONISATION ET COMMUNICATION ENTRE PROCESSUS**

Concepts de processus et de threads

Concepts de processus

→ Définition d'un processus

Un processus est **un programme en cours d'exécution** dans un système informatique. Il s'agit d'une entité logique qui est créée par le système d'exploitation et qui est associée à un ensemble de ressources, telles que la mémoire, les fichiers, les ports d'E/S, etc.

- Un processus est l'entité dynamique représentant l'exécution d'un programme sur un processeur.
- Un processus est l'activité résultant de l'exécution d'un programme séquentiel, avec ses données, par un processeur.
- Si un même programme est exécuté plusieurs fois, il correspond à plusieurs processus.
- Un processus peut **communiquer** des informations avec d'autres processus.

Concepts de processus

→ Les caractéristiques du processus

Un processus est caractérisé par:

1. Un numéro d'identification unique (**PID**)
2. Un espace d'adressage (code, données, piles d'exécution)
3. Un état principal (prêt, en cours d'exécution (élu), bloqué, …)
4. Les valeurs des registres lors de la dernière suspension (CO, sommet de Pile...)
5. Une priorité
6. Les ressources allouées (fichiers ouverts, mémoires, périphériques …)
7. Les signaux à capter, à masquer, à ignorer, en attente et les actions associées
8. Autres informations indiquant le processus père, les processus fils, le groupe, les variables d'environnement, les statistiques et les limites d'utilisation des ressources....

Concepts de processus

→ Notions sur les processus

→ Processus père et processus fils

Le processus fils est un processus qui a été créé par un autre processus qui prend le nom de processus père.

→ Identification d'un processus

Un processus sous Linux est identifié par un numéro unique qui s'appelle le numéro d'identification du processus PID (Process IDentifier) et qui lui est attribué par le système à sa création.

→ Temps partagé

On appelle **temps partagé** une exploitation dans laquelle plusieurs processus sont simultanément en cours d'exécution.

→ Swapping (va-et-vient)

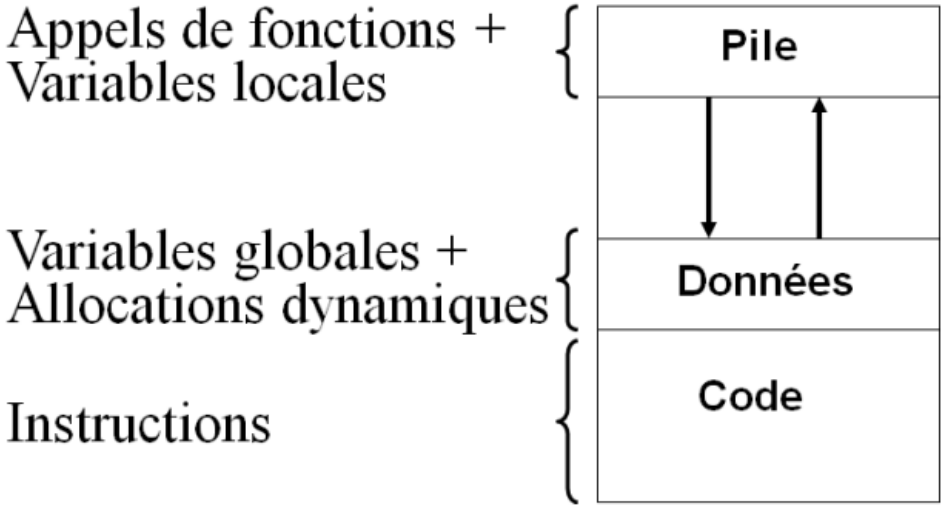
Le **swapping** consiste en la recopie sur disque d'un processus complet ou d'une partie d'un processus ayant perdu le contrôle de l'unité centrale et ne pouvant plus rester en mémoire centrale. La mémoire centrale ainsi libérée est affectée à un processus plus prioritaire.

Concepts de processus

→ Espace de travail d'un processus

Les processus sont composés d'un espace de travail en mémoire formé de 3 segments: **la pile**, **les données** et **le code**

- Le **code** correspond aux instructions, en langage d'assemblage, du programme à exécuter.
- La zone de **données** contient les variables globales ou statiques du programme ainsi que les allocations dynamiques de mémoire.
- Les appels de fonctions, avec leurs paramètres et leurs variables locales, viennent s'empiler sur la **pile**.



Pseudo-parallélisme est parfois utilisé pour référer à des processus multiples s'exécutant sur un seul processeur

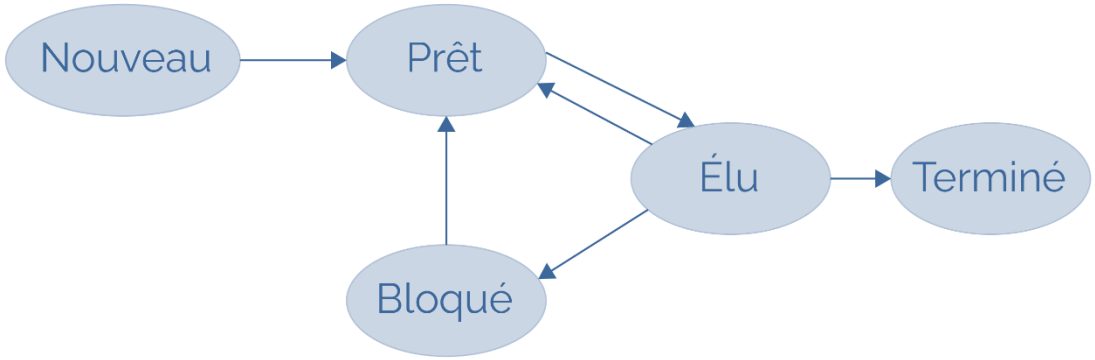
Concepts de processus: Etats de processus

Les processus se partagent le processeur et changent d'états selon qu'ils possèdent ou non le processeur. Ces états correspondent principalement aux états suivants :

- **Actif** (ou élu) Le processus est en cours d'exécution par le processeur.
- **Prêt** :Le processus est en attente du processeur pour s'exécuter.
- **Bloqué** : Le processus est en attente d'une ressource ou d'une opération d'entrée/sortie et ne peut pas continuer à s'exécuter.

Il existe également d'autres états pour la création et la terminaison d'un processus, tels que :

- **Nouveau** : création d'un processus dans le système
- **Fin**: terminaison de l'exécution



Concepts de processus : Etats de processus

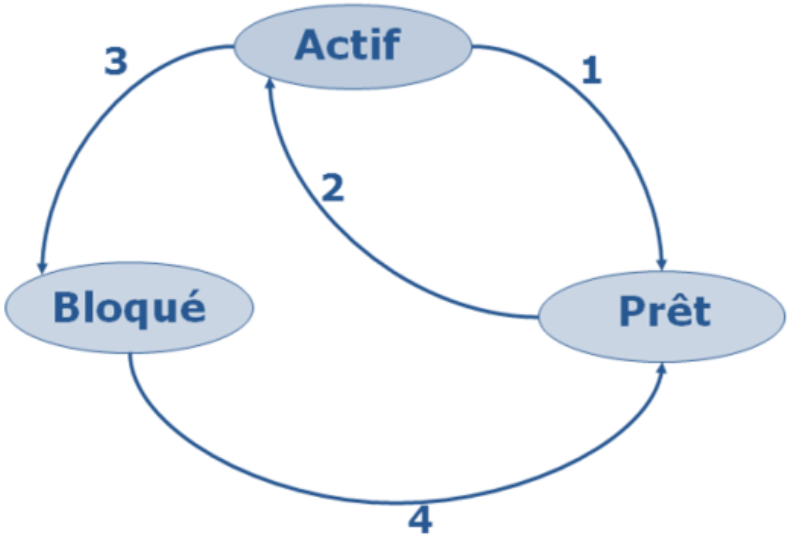
Exemple

- $ ***who | wc -l***

Dans le cas de la commande *who | wc -l*, le processus *wc* se bloque en attente des résultats du processus *who*.

→ **Les différentes transitions d'un état à un autre sont effectuée:**

- **La transition 1** se produit lorsque le processeur est retiré du processus car son quantum a expiré, suite à un appel système ou une interruption.
- **La transition 3** se produit lorsque le processus attend une ressource autre que le processeur (un événement extérieur).
- **La transition 4** se produit lorsque le processus a toutes les ressources qu'il faut (autres que la CPU).
- **La transition 2** a lieu lorsque le processeur lui est alloué (à son tour).

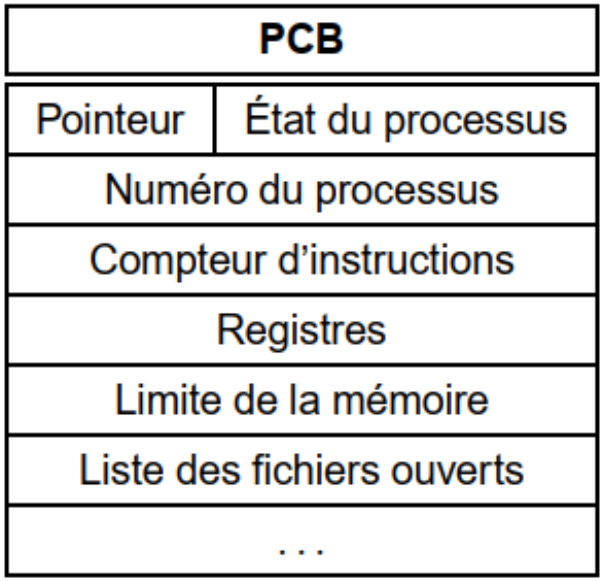


Concepts de processus : Processus et bloc de contrôle (PCB)

Un bloc de contrôle de processus, ou **PCB** (*Process Control Block*), est une structure de données utilisée par les systèmes d'exploitation pour suivre l'état et les informations relatives à un processus.

- **Chaque processus possède un PCB qui contient des informations telles que :**
  - l'identifiant du processus (PID).
  - l'identifiant du processus père (PPID).
  - l'état actuel du processus (prêt, en attente, en cours d'exécution, etc.),
  - la priorité du processus,
  - le temps restant de quantum,
  - les ressources utilisées et demandées par le processus, et bien plus encore.

- Les PCBs sont rangés dans une table de processus. Ces derniers sont créés et peuvent être détruits.
- Les processus peuvent partager de la mémoire, des processus, des fichiers, des I/Os et autres.



Concepts de processus: Commutation de contexte

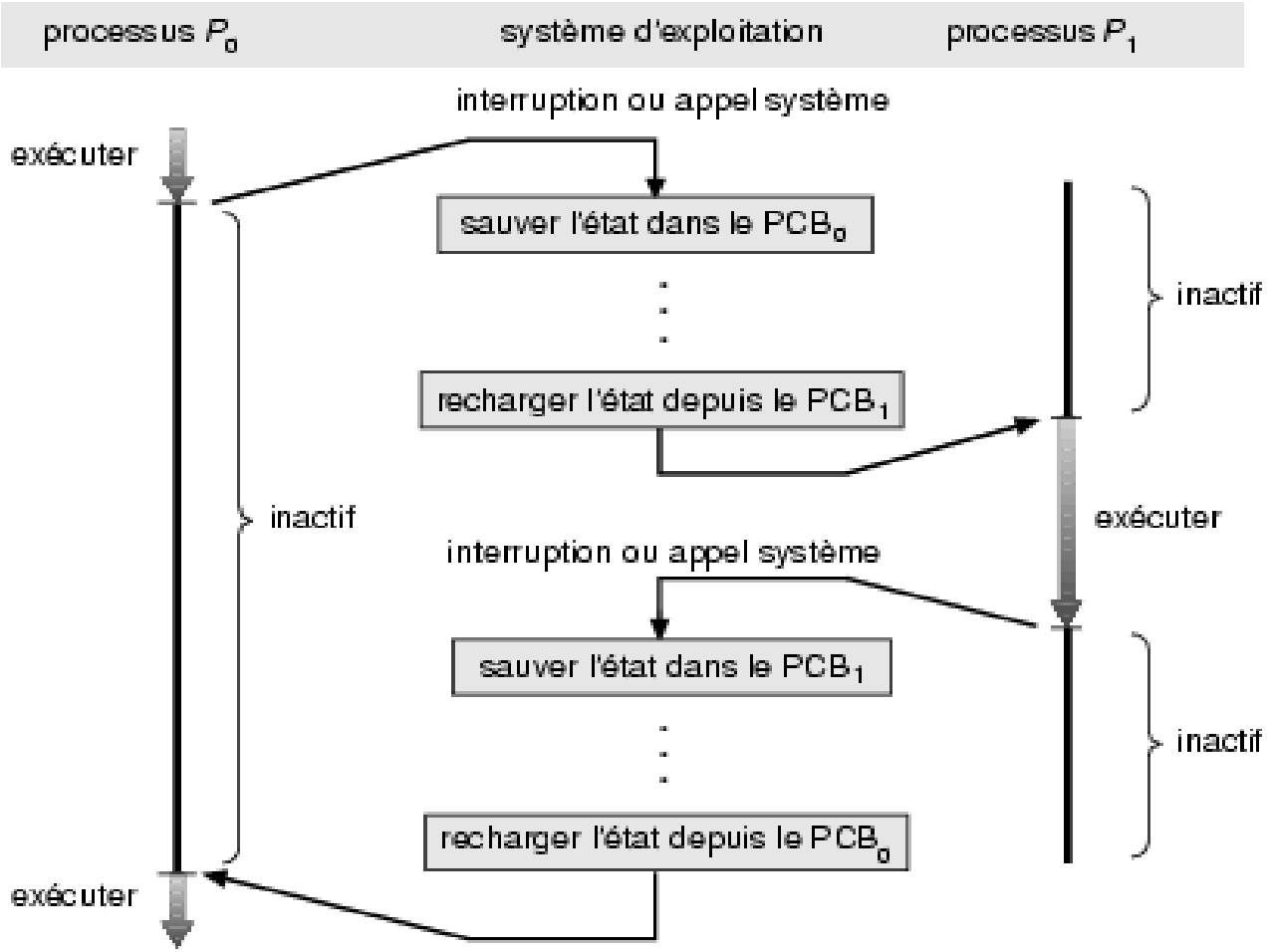
La commutation de processus, également appelée **changement de contexte (**ou context switching), est le mécanisme par lequel le processeur passe d'un processus en cours d'exécution à un autre.

- Cette opération est gérée par le système d'exploitation et consiste à sauvegarder l'état actuel du processus en cours d'exécution, à charger l'état du processus suivant et à reprendre l'exécution à partir de ce point.
- **La commutation de processus** permet au système d'exploitation de garantir une utilisation efficace du processeur et de gérer l'allocation des ressources entre les différents processus en cours d'exécution.

Quand l'UCT passe de l'exécution d'un processus 0 à l'exécution d'un processus 1, il faut :

- mettre à jour le PCB de **0**
- sauvegarder le PCB de **0**
- reprendre le PCB de **1**, qui avait été sauvegardé avant
- remettre les registres d 'UCT, compteur d 'instructions etc. dans la même situation qui est décrite dans le PCB de **1**

Concepts de processus : Commutation de contexte



Concepts de threads

→ Définition d'un processus légers *«thread »*

Les **threads** appelés aussi **processus légers (**ou fils d'exécution) , comme les processus, est un mécanisme permettant a un programme de faire plus d'une chose a la fois (càd. exécution simultanée des parties d'un programme).

- Un processus peut être composé d'un ou de plusieurs processus légers (threads).
- *« Un **thread** est une unité d'exécution rattachée à un processus, chargée d'en exécuter une partie. »*
- Un processus possède un ensemble de ressources (code, fichiers, périphériques…) que ses threads partagent.

Exemple

pour un même document MS-Word, plusieurs threads: Interaction avec le clavier, sauvegarde régulière du travail, contrôle d'orthographe…)

Concepts de threads

Chaque *thread* dispose :

- D'un compteur programme (pour le suivi des instructions à exécuter)
- De registres systèmes (pour les variables de travail en cours)
- D'une pile (pour l'historique de l'exécution)

Avantages des *threads*:

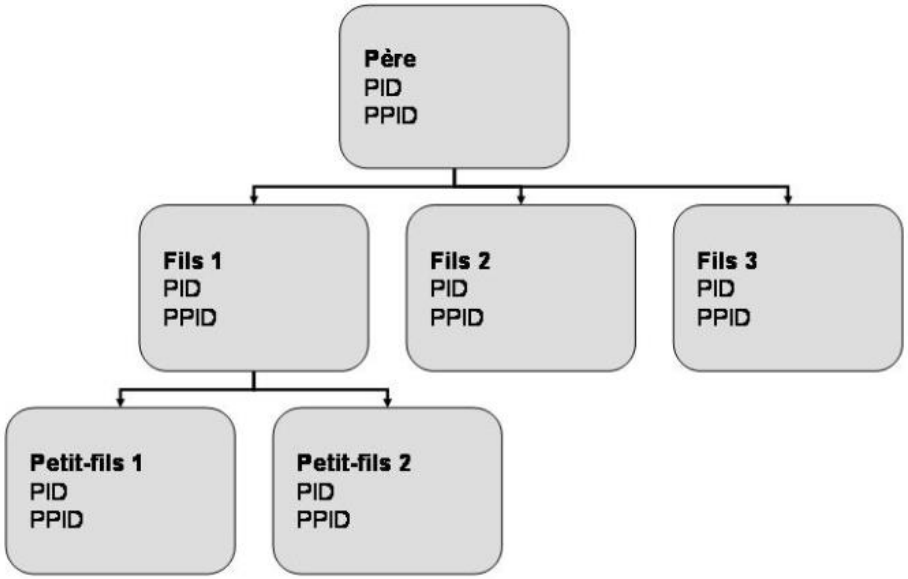
- **Réactivité:** Le processus peut continuer à s'exécuter même si certaines de ses parties sont bloquées (en chargement de fichiers par exemple).
- **Economie d'espace mémoire:** Partage de ressources, surtout la mémoire, entre threads d'un même processus.
- **Economie de temps:** Les threads partagent les ressources du processus auquel ils appartiennent. Ainsi, il est plus économique de créer et de gérer les threads que les processus entre eux.
- **Scalabilité:** Un processus à thread unique ne peut s'exécuter que sur une CPU. Alors qu'un processus à multithreads, peut s'exécuter sur plusieurs CPU (quand elles existent) en même temps.

Gestion des processus

Sous linux

→ Hiérarchie de processus

- Dans certains SE, lorsqu'un processus crée un autre processus, les processus **parent** et enfant continuent d'être associés d'une certaine manière.
- Le processus **enfant** peut lui même créer plusieurs processus, formant un hiérarchie de processus.
- Un processus a un **seul parent** et peut avoir **0 ou plusieurs fils**.



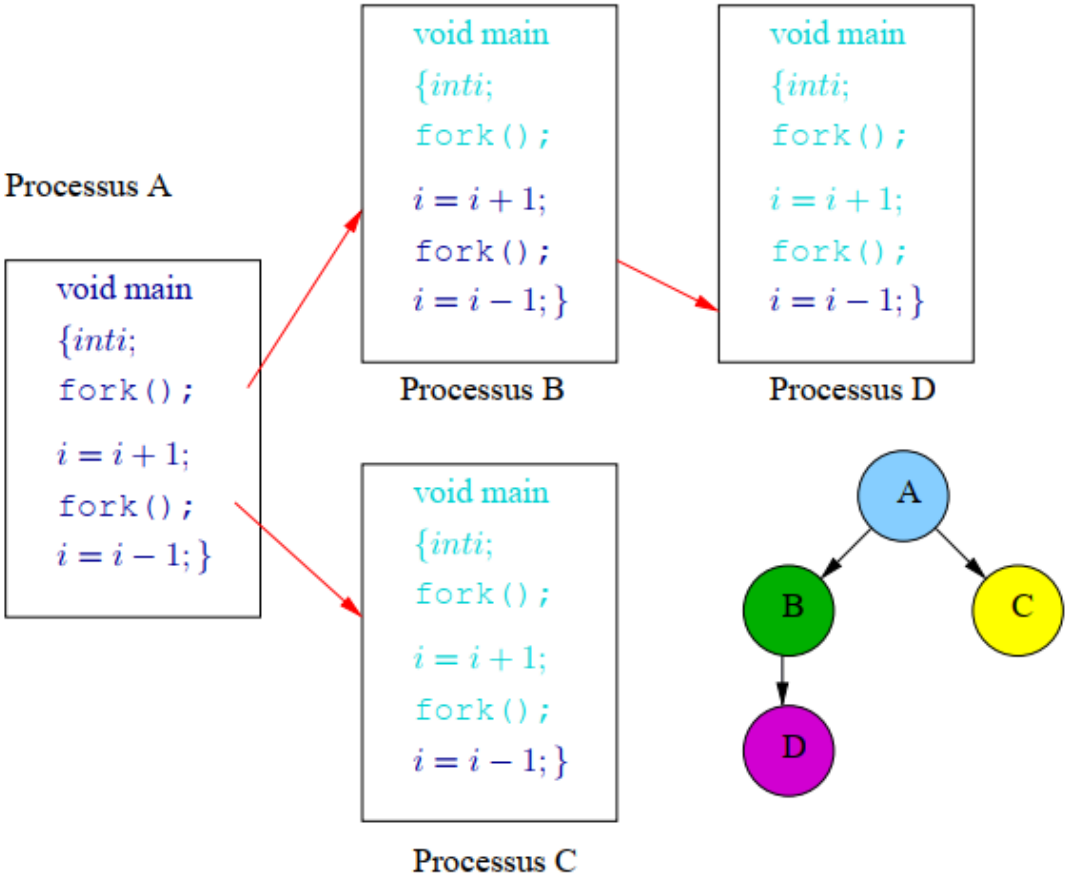
→ Hiérarchie d'un processus

Exemple

→ Le cas de S.E Linux/UNIX:

- Si le ***processus A*** crée le ***processus B***, ***A*** est le parent de ***B***, ***B*** est le fils de ***A*** (***A*** par défaut, exécute le même code que ***B***) ***B*** peut à son tour créer des processus.

- Un processus avec tous ses descendants forment un groupe de processus représenté par un arbre de processus.(fork est le seul appel système de création de processus).



Gestion des processus

→ Création et la destruction des processus

La **création** et la **destruction** des processus sont des opérations fondamentales dans les systèmes d'exploitation.

- **Un processus ne peut être créé** que par un autre au moyen d'un appel système. Il possède un PCB dont les éléments sont initialisés aux éléments de celui de son père (sauf le **PID**, le **PPID** et le temps **UC**). Il hérite également une copie des blocs de données et de code de son père.
  - Les processus sont structurés en arborescence à partir d'un ancêtre créé au démarrage.
- **Les processus sont détruits** en général à la fin de leur exécution.
  - Un processus peut aussi être détruit à la demande d'un autre processus ayant ce droit (comme son père).
  - Lorsqu'il est détruit, son PCB et ses ressources sont libérés. En plus, il faut noter que la destruction du père peut entraîner celle de sa descendance dans le cas de certains systèmes d'exploitation.

→ Gestion des processus sous UNIX/LINUX

Pour la gestion des processus sous Unix, cinq principaux appels système sont prévus. Il s'agit des appels dédiés à :

1. **L'identification** : ***int getpid (void) :***qui retourne l'identité du processus courant alors que **int getppid(void)** qui retourne l'identité du processus père.

2. **La création** : ***int fork (void)*** **:**crée un processus fils, cette fonction retourne :
   - **-1** en cas d'´echec,
   - Dans le processus-fils, **fork**() renvoie **0** alors que dans le processus-père elle renvoie l'identité du fils (**PID**)
   - Le syntaxe :

```
#include <unistd.h>
//#include <sys/types.h>
 pid_t fork(void);
```

→ Gestion des processus sous UNIX/LINUX

3. **La terminaison** : ***void exit (int status) :***termine le processus appelant et met la valeur 'status' dans son PCB.
   - Si le processus a des processus fils lorsque ***exit()*** est appelée, ces derniers ne sont pas modifiés mais leur nom de processus parent est changé en **1**, car leur processus parent se termine.
   - Par convention, un code de retour égal à **0** signifie que le processus s'est terminé correctement, et un code **non nul** (généralement 1) signifie qu'une erreur s'est produite.
   - Le père du processus qui effectue un ***exit()*** reçoit son code retour à travers un appel à **wait()**.

→ **Gestion des processus sous UNIX/LINUX**

4. **L'attente** : ***int wait (int *)*** :provoque une attente de la fin d'un processus fils
   Syntaxe:

```
#include <sys/wait.h>
int wait (int *)
```

   - La fonction wait() prend en paramètre un pointeur vers une variable entière, qui sera remplie avec l'identifiant du processus fils qui s'est terminé. Si le processus n'a pas de fils, la fonction retourne **-1**. (Retour immédiat ou blocage jusqu'à terminaison d'un fils).
   - ***waitpid()*** : cette fonction est similaire à ***wait()***, mais elle permet de spécifier le PID du processus fils à attendre.

→ Gestion des processus sous UNIX/LINUX

5. **Le recouvrement** :la primitive **exec** (avec 6 variantes selon le nombre de paramètres) permet de remplacer le code du processus appelant par un nouveau code donné en paramètre. Elle est souvent utilisée pour lancer un nouveau programme à partir d'un processus existant.

- **execve()** : Remplace le programme en cours d'exécution par un nouveau programme.
- **execl()** : cette fonction est similaire à execve(), mais les arguments sont passés sous forme de liste d'arguments, plutôt que dans un tableau.
- **execvp**() : cette fonction recherche le fichier exécutable spécifié dans les chemins d'accès définis dans la variable d'environnement PATH.
- **execle**() : cette fonction est similaire à execve(), mais elle permet également de spécifier les variables d'environnement à utiliser pour le nouveau programme.
- **execlp()** : cette fonction est similaire à execl(), mais elle recherche le fichier exécutable dans les chemins d'accès définis dans la variable d'environnement PATH.
- **execv()** : cette fonction est similaire à execve(), mais elle prend en entrée les arguments sous forme de tableau.
- **execvp()** : cette fonction est similaire à execv(), mais elle recherche le fichier exécutable dans les chemins d'accès définis dans la variable d'environnement PATH.

→ **Héritage**

Le processus-fils hérite:

- les propriétaires réels et effectifs,
- le répertoire de travail,
- la valeur de nice

Le processus-fils n'hérite pas:

- l'identité de son père,
- les temps d'exécution car ils sont initialises à 0 chez le fils,
- verrous sur les fichiers détenus par le père,
- signaux pendants (arrivées, en attente d'être traites) du père.

Un processus **zombie** est un processus qui a été terminé mais qui n'a pas été correctement supprimé de la table de processus du système d'exploitation.

Relation père/fils

- Le père est toujours prévenu de la fin d'un fils
- Le fils est toujours prévenu de la fin du père mais il faut que le père soit en attente
- Si la fin d'un fils n'est pas traitée par le père ce processus devient un processus **zombie**.

Exercice 1

Écrire un programme qui crée un processus fils à l'aide de la fonction fork(). Le processus fils doit afficher le message "Bonjour, je suis le processus fils" et le processus parent doit afficher le message "Bonjour, je suis le processus parent".

correction

Exercice 2

Écrire un programme qui crée deux processus fils à l'aide de la fonction fork(). Le premier processus fils doit afficher le message "Bonjour, je suis le premier processus fils" et le deuxième processus fils doit afficher le message "Bonjour, je suis le deuxième processus fils". Le processus parent doit afficher le message "Bonjour, je suis le processus parent".

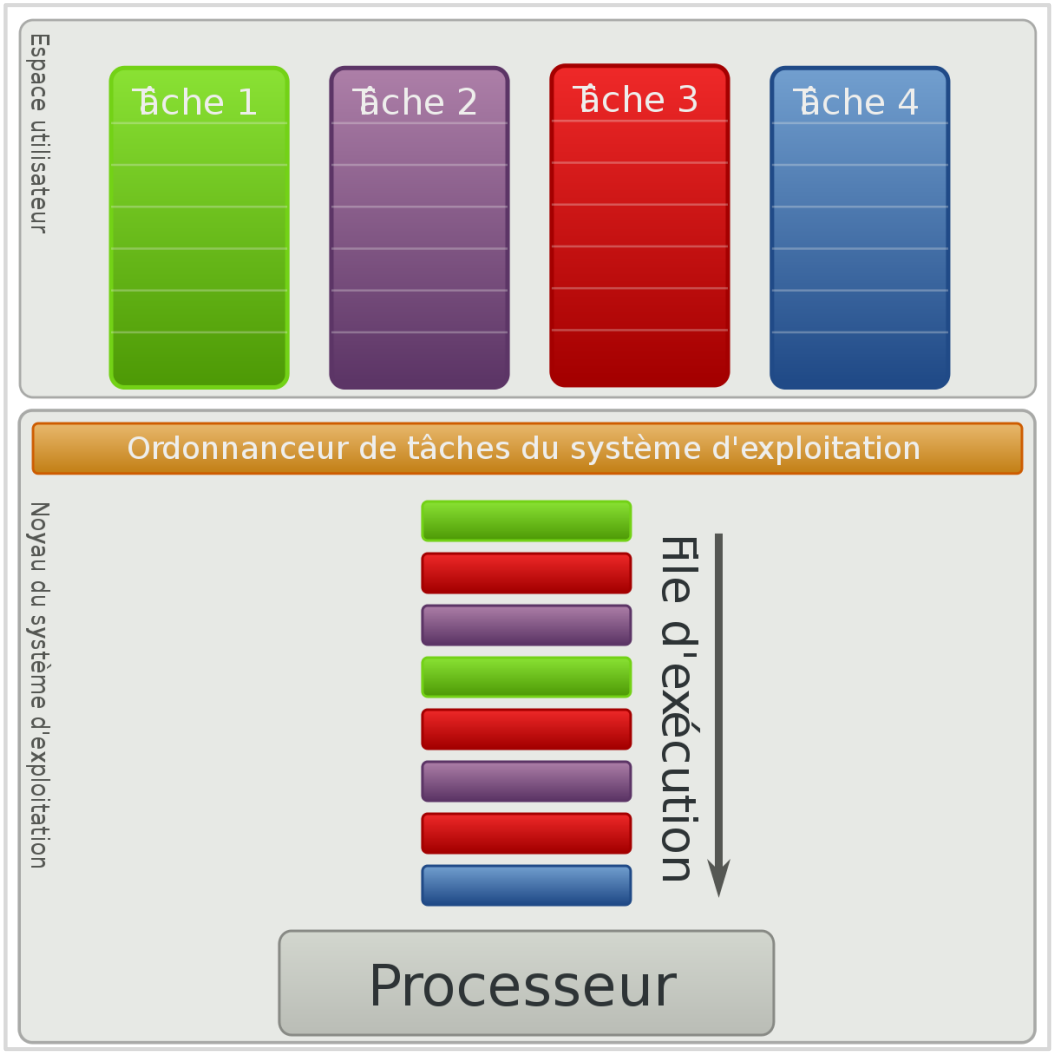
Correction

Ordonnancement des processus

Ordonnancement des processus

→ **Ordonnanceur**

- **Ordonnanceur (Scheduler)**: le module du noyau du système d'exploitation qui choisit les processus qui vont être exécutés par les processeurs d'un ordinateur.



Ordonnancement des processus

→ **Notions utiles:** *les interruptions*

- **Une interruption** est un signal pour arrêter un processus, qui peut avoir plusieurs causes:
  - **Interruptions causées par le programme utilisateur:**

    - Exception: Division par 0, débordement, tentative d'exécuter une instruction protégée, référence au delà de l'espace mémoire du programme

    - Appels Système: demande d'entrée-sortie, demande d'autre service du SE, minuterie établie par le programme lui-même.
  - **Interruptions causées par le SE:**

    o Le processus doit céder l'UCT à un autre processus (Préemption).
  - **Interruptions causées par les périphériques ou par le matériel:**

    o Fin d'une E/S.

Ordonnancement des processus

→ **Notions utiles: *l*es *interruptions***

- La commutation du processeur est réalisée au moyen d'un ordonnanceur et d'un mécanisme **d'interruption**.
- Les interruptions sont dues à des événements internes ou externes et peuvent être de type:
  - Horloge (Quantum épuisé)
  - Disque (E/S)
  - Appel système fait par le processus…
- A chaque type est associée une routine de traitement d'interruption dont l'adresse se trouve dans le vecteur d'interruption.

Ordonnancement des processus

→ **Notions utiles:** *les files d'attente*

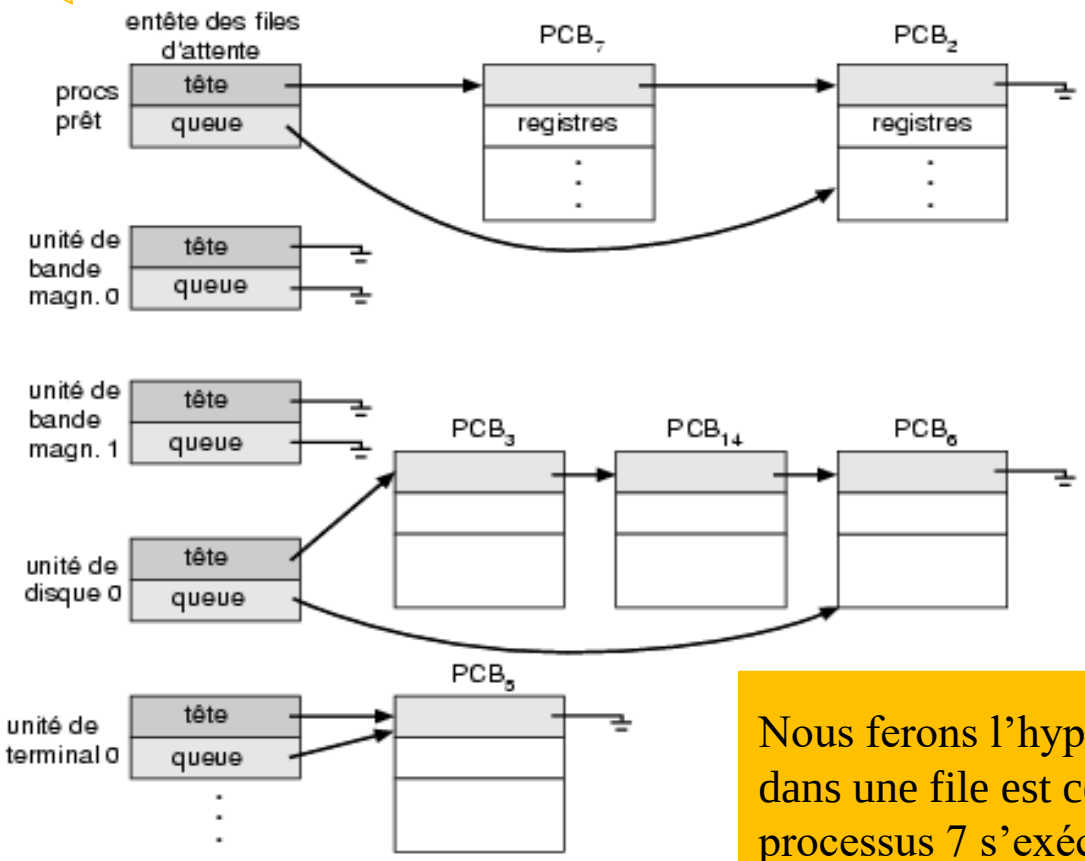
- Les processus qui résident dans la mémoire principale et sont prêts et en attente d'exécution sont conservés sur une liste appelée **File des Processus Prêts** (ou **Ready Queue**).
- **Chaque ressource** a sa propre file de processus en attente.
- Il s'agit d'une liste chaînée qui contient un pointeur vers le PCB (Process Control Block) du processus ainsi qu'un pointeur vers le PCB du processus suivant dans la file.
- En changeant d'état, les processus se déplacent d'une file à l'autre.

Ordonnancement des processus

→ **Notions utiles:** *les files d'attente*

File des prêts

Ce sont les PCBs qui sont dans les files d'attente



Nous ferons l'hypothèse que le premier processus dans une file est celui qui utilise la ressource: ici, processus 7 s'exécute, processus 3 utilise disque 0, etc.

Ordonnancement des processus

→ Ordonnancement de processus

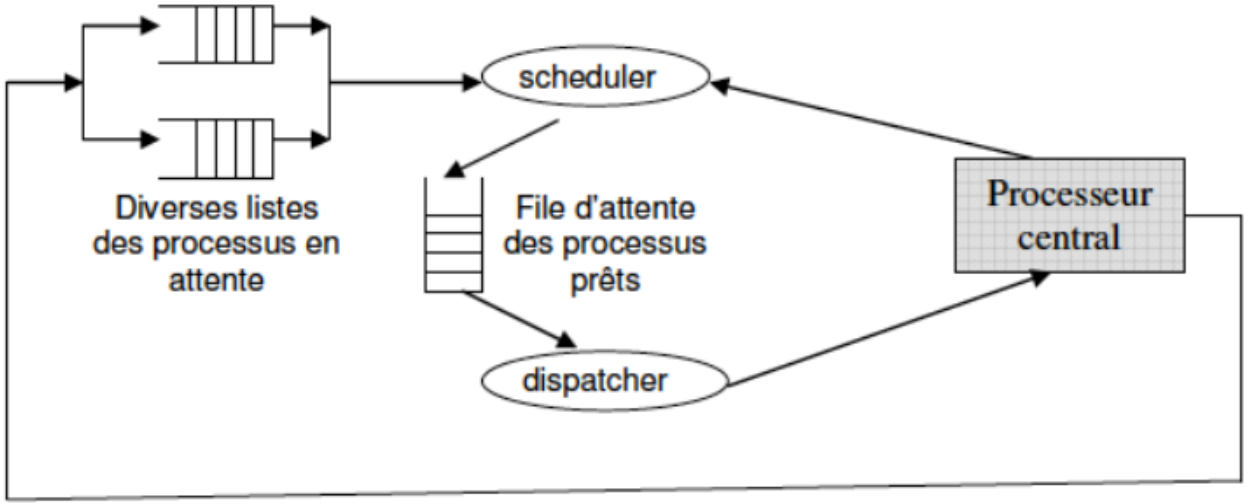
- Dans un système multitâche plusieurs processus sont en cours simultanément, mais le processeur ne peut, à un moment donné, exécuter qu'une instruction (d'un programme) à la fois. Le processeur travaille donc en **temps partagé**.
- L'ordonnanceur (**scheduler**) est le module du SE qui s'occupe de sélectionner le processus suivant à exécuter parmi ceux qui sont prêts.
- Un processus passe entre les diverses **files d'attente** pendant sa durée de vie (file d'attente des processus prêts (attendant pour s'exécuter), fille d'attentes des périphériques, ...).
- Le SE doit sélectionner les processus à partir de ces files d'attente d'une manière quelconque.
- Le processus de sélection est mené à bien par le **scheduler** approprié.

Ordonnancement des processus

→ Ordonnancement de processus

Critères d'ordonnancement

- L'ordre d'arrivée
- Durée d'exécution
- La priorité



Classification des processus

- ***Tributaire des E/S:*** dépense la plupart du temps à effectuer des E/S plutôt que des calculs.
- ***Tributaire de la CPU:*** génère peut fréquemment des requêtes d'E/S, càd, passe plus de temps à effectuer des calculs

Ordonnancement des processus

→ Les types d'ordonnanceurs

L'ordonnancement est à envisager à trois niveaux : à court, à moyen et à long terme.

- **L'ordonnanceur à long terme (***job schedulling ou ordonnanceur de travaux)*
  - Sélectionne le processus qui doit aller dans la file de processus prêts.
  - S'exécute moins fréquemment : peut être lent (seconds, minutes).
  - Contrôle le degré de multiprogrammation (le nombre de processus dans la mémoire).
  - Il est important que le scheduler à long terme réalise un bon mélange de processus tributaires de la CPU et tributaires des E/S.
- **L'ordonnanceur à moyen terme (***permutateur* ou *swapper)* a pour rôle de permuter les processus placés en mémoire avec ceux qui ont été temporairement stockés sur le disque en raison d'un manque de place en mémoire. Toutefois, ces permutations ne peuvent pas être trop fréquentes afin de ne pas gaspiller la bande passante des disques.
- **L'ordonnanceur à court terme** (*dispacher)*, répartiteur ou **ordonnanceur du processeur**, est responsable de choisir le processus auquel le processeur sera alloué et pour quelle durée. Les commutations de processus sont très fréquentes à ce niveau.

→ Les algorithmes d'ordonnancement

Les algorithmes d'ordonnancement (scheduler) peuvent être classés en deux catégories:

1. **Non préemptif** (*sans réquisition*)

L'ordonnanceur sélectionne un processus et le laisse s'exécuter jusqu'à ce qu'il se bloque (soit sur une E/S, soit en attente d'un autre processus) ou qu'il libère volontairement le processeur. Même s'il s'exécute pendant des heures, il ne sera pas suspendu de force. Aucune décision d'ordonnancement n'intervient pendant les interruptions de l'horloge.

1. **Préemptif** (*avec réquisition*)

L'ordonnanceur sélectionne un processus et le laisse s'exécuter pendant un délai déterminé. Si le processus est toujours en cours à l'issue de ce délai, il est suspendu et l'ordonnanceur sélectionne un autre processus à exécuter.

→ Les critères d'évaluation de performances

Les critères permettant de comparer les stratégies et les algorithmes d'ordonnancement:

- **Temps de réponse** (response time): le temps qui s'écoule entre la soumission d'une requête et la première réponse obtenue.
- **Temps de rotation** (turn-around time): le 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 éléments éligibles + temps d'exécution dans le processeur et E/S).
- **Temps d'attente** (waiting time) : le temps passé dans la file d'attente des processus éligibles.
- **Rendement** (throughput) : le nombre de travaux exécutés par unité de temps (débit).

La politique d'ordonnancement (politique d'allocation de CPU) : détermine le choix d'un processus à élire parmi tous ceux qui sont prêts.

→ Les objectifs de l'ordonnanceur

Les objectifs d'un ordonnanceur d'un système sont, entre autres :

- S'assurer que chaque processus en attente d'exécution reçoive sa part de temps processeur.
- Minimiser le temps de réponse ;
- Utiliser le processeur à 100% (ce serait gâcher de la ressource)
- Utiliser d'une manière équilibrée les ressources
- Prendre en compte les priorités
- Être prédictible (et ce n'est pas une mince affaire...)

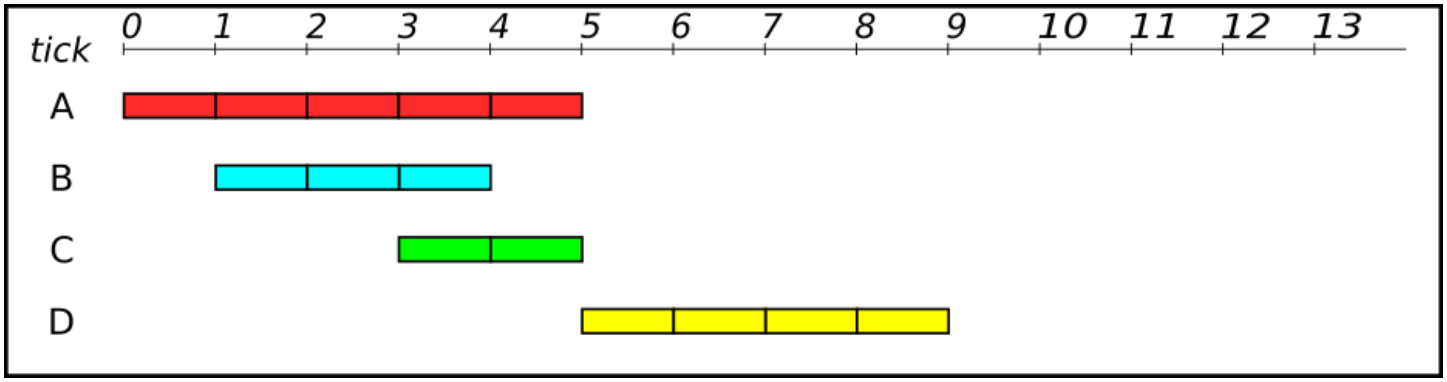
Ces objectifs sont parfois complémentaires, parfois contradictoires :

✓ Augmenter la performance par rapport à l'un d'entre eux peut se faire en détriment d'un autre.

✓ Il est impossible de créer un algorithme qui optimise tous les critères de façon simultané.

Imaginons que le processeur ait à exécuter 4 processus, dont les temps d'exécutions sont différents, et qui se sont présentés à différents instants au processeur :

- le processus A a besoin de 5 cycles d'horloge, et est arrivé au tick 0
- le processus B a besoin de 3 cycles d'horloge, et est arrivé au tick 1
- le processus C a besoin de 2 cycles d'horloge, et est arrivé au tick 3
- le processus D a besoin de 4 cycles d'horloge, et est arrivé au tick 5



Un bon algorithme d'ordonnancement doit:

- Maximiser le taux d'utilisation de l'UC et le débit
- Minimiser le temps moyen de rotation
- Minimiser le temps moyen d'attente
- Minimiser le temps de réponse

Ordonnancement des processus

→ Les algorithmes Non-Préemptifs

Premier-arrivé premier-servis (PAPS)
First Come First Served (FCFS)

- L'algorithme **FCFS** traite les processus dans l'ordre de leur soumission (date d'arrivée) sans aucune considération de leur temps d'exécution.
- L'organisation de la file d'attente des processus prêts est donc tout simplement du "First In First Out (FIFO)".
- L'algorithme FCFC consiste à choisir à un instant donné, le processus qui est depuis le plus longtemps dans la file d'attente, ce qui revient à choisir celui disposant du temps d'arrivée minimal et l'exécuter pendant un temps d'exécution bien définit.
- Ce procédé est répété jusqu'à épuisement des processus dans la file d'attente.

❑ **Avantages:**
- Simple
- Pas de famine (toutes les tâches seront exécutées tôt ou tard)

❑ **Inconvénients:**
- Temps d'attente moyen très important.
- Non adapté aux systèmes interactifs.

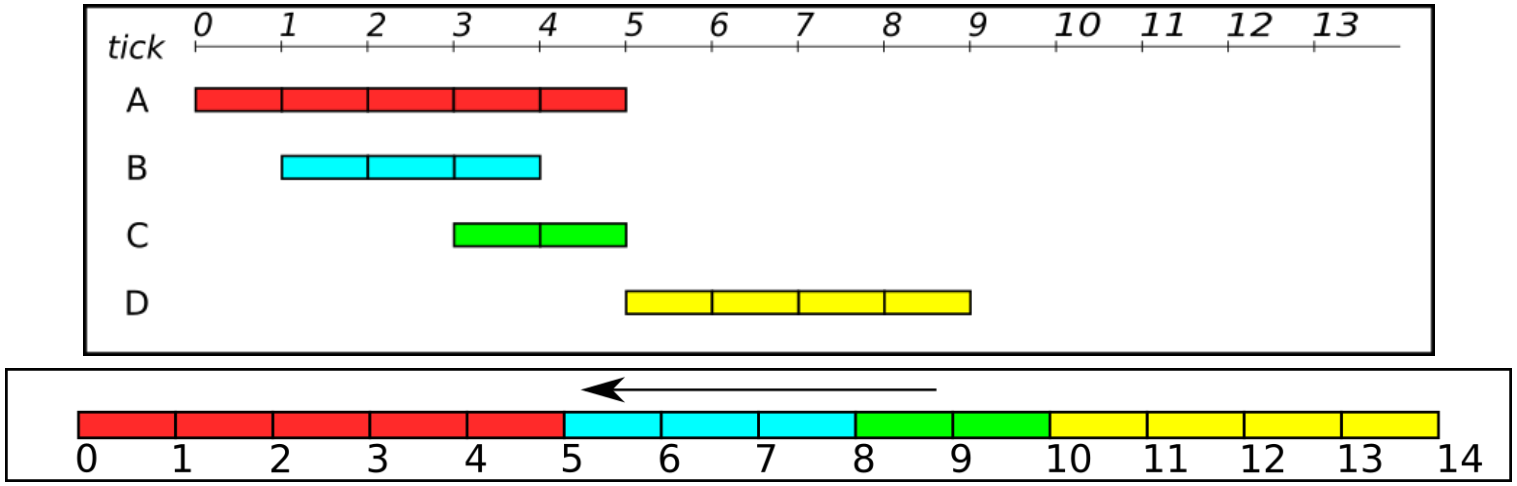
Ordonnancement des processus

→ Les algorithmes Non-Préemptifs

Premier-arrivé premier-servis (PAPS)
First Come First Served (FCFS)

Exemple

Dans cet exemple, chaque processus sera exécuté du début à la fin, sans interruptions. Il faudra 14 cycles d'horloge pour terminer les 4 processus. C'est le principe de la file d'attente pour les imprimantes : peu importe le nombre de pages à imprimer, les documents seront imprimés en entier et dans leur ordre d'arrivée.



→ Les algorithmes Non-Préemptifs

Premier-arrivé premier-servis (PAPS)
First Come First Served (FCFS)

Exemple

Les mesures de performances

- **Le temps de rotation** (de séjour ou cycle UCT) nécessaire à l'exécution de chaque processus, qui correspond à la différence entre le **temps de terminaison** du processus et le **temps d'entrée** dans le processeur :

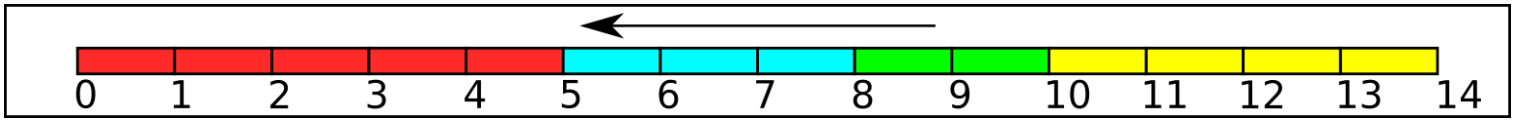
- A est terminé au tick 5, et est entré au tick 0 , donc son temps de séjour est : $\mathbf{t_A = 5 - 0 = 5}$
- B est terminé au tick 8, et est entré au tick 1 d'où : $\mathbf{t_B = 8 - 1 = 7}$
- C est terminé au tick 10, et est entré au tick 3 d'où : $\mathbf{t_C = 10 - 3 = 7}$
- D est terminé au tick 14, et est entré au tick 5 d'où : $\mathbf{t_D = 14 - 5 = 9}$

1. **Le temps de rotation moyen** avec cette méthode est donc : $\frac{5+7+7+9}{4} = \frac{28}{4} = \mathbf{7}$

→ Les algorithmes Non-Préemptifs

Premier-arrivé premier-servis (PAPS)
First Come First Served (FCFS)

Exemple



Les mesures de performances

❑ **Le temps d'attente** est **le temps de rotation** auquel on soustrait **le temps d'exécution**

- **A** a pour temps de rotation 5 et pour temps d'exécution 5. Son temps d'attente est donc **$t'_A = 5 - 5 = 0$** , ce qui signifie que le processus A n'a passé aucun temps en état prêt ou bloqué
- **B** a pour temps de rotation 7 et pour temps d'exécution 3 : **$t'_B = 7 - 3 = 4$**
- **C** a pour temps de rotation 7 et pour temps d'exécution 2 : **$t'_C = 7 - 2 = 5$**
- **D** a pour temps de rotation 9 et pour temps d'exécution 4 : **$t'_D = 9 - 4 = 5$**

2. **Le temps d'attente moyen** avec cette méthode est donc : $\frac{0+4+5+5}{4} = \frac{14}{4} = 3{,}5$
3. **Utilisation UCT**: 14/14=100%
4. **Débit:** 4/14 = 0,28

→ Les algorithmes Non-Préemptifs

Shortest Job First (SJF)

- L'algorithme **SJF** choisit de façon prioritaire les processus ayant le plus court temps d'exécution sans réellement tenir compte de leur date d'arrivée.
- Ce procédé est répété jusqu'à épuisement des processus dans la file d'attente.

❑ Avantages

- Le meilleur pour le temps d'attente moyen (lorsque tous les processus arrivent en même temps.)

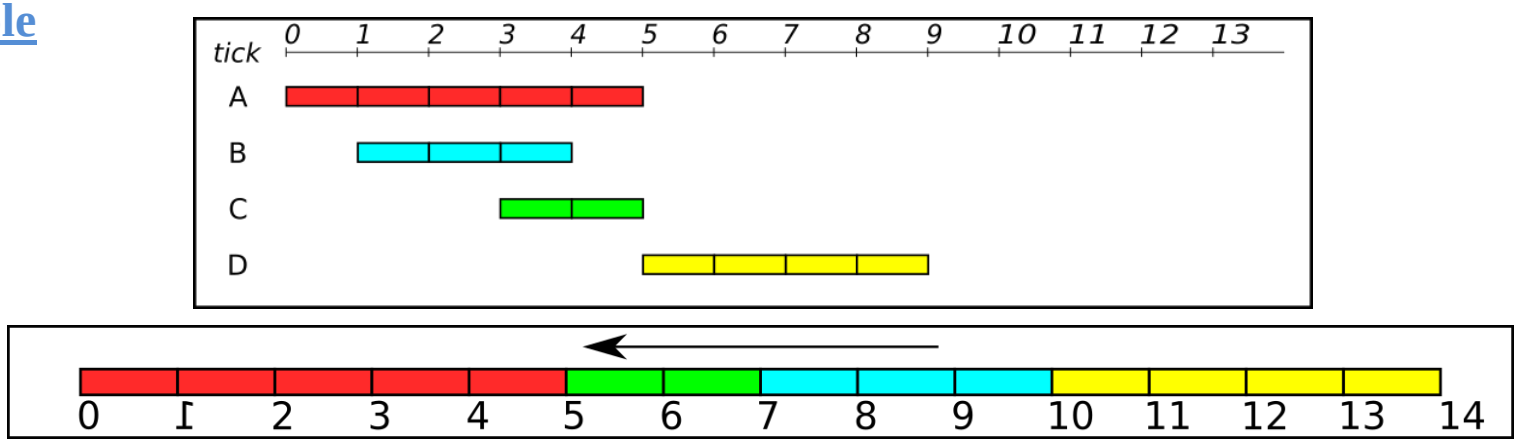
❑ Inconvénients

- Risque de famine: les processus longs peuvent ne jamais s'exécuter
- Nécessite de connaitre à l'avance le temps du cycle UCT (adapté aux traitements par lots où une estimation de la durée du cycle est donnée). Sinon, il devrait être prédit

→ Les algorithmes Non-Préemptifs

Shortest Job First (SJF)

Exemple



- Au tick 0, on exécute le processus A.
- Au tick 5, on a le choix entre les processus B, C et D qui sont dans la file d'attente, donc on exécute C, qui est le plus rapide.
- Au tick 7 on exécute B.
- On termine par D.

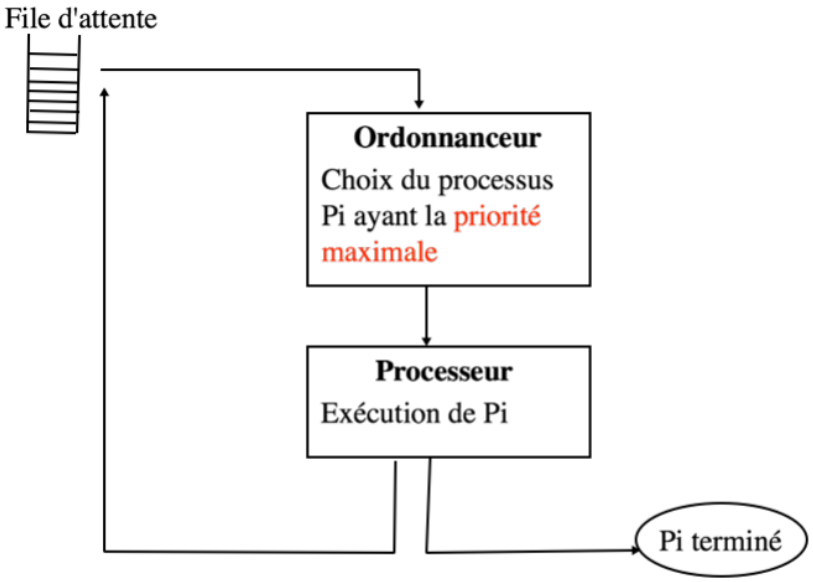
→ **Calculer le temps de séjour moyen et le temps d'attente moyen**

→ Les algorithmes Non-Préemptifs

Algorithme basé sur la priorité
Highest Priority First ou haute priorité d'abord (HPF)

- Les algorithmes fondés sur les priorités attribuées par le système d'exploitation aux processus choisissent les processus les plus prioritaires sans prise en considération d'une manière générale des données durée d'exécution et date d'arrivée des processus.

- Le principe de cet algorithme consiste à associer à chaque processus une priorité. Le processeur est alloué au processus ayant la plus grande priorité. Ce procédé est répété jusqu'à épuisement des processus se trouvant dans la file d'attente.



→ Les algorithmes Non-Préemptifs

Exercice

Considérons les processus suivants, avec leur temps d'arrivée (en unité de temps), temps CPU (en unité de temps) et priorité (plus le nombre est élevé, plus la priorité est importante) :

Processus	Temps d'arrivée	Temps CPU
A	0	7
B	4	3
C	5	5
D	7	3

1. Quel est le temps moyen de séjour (de rotation) et d'attente pour l'algorithme SJF (Shortest Job First) ?
2. Quel est le temps moyen de séjour et d'attente pour l'algorithme FCFS (First-Come, First-Served) ?

Ordonnancement des processus

→ Les algorithmes Préemptifs

Shortest Remaining Time (SRT)

- L'algorithme **SRT** est la version préemptive de l'algorithme **SJF**, qui privilégie l'exécution du processus avec le temps d'exécution restant le plus court.
- Un processus arrive dans la file de processus, l'ordonnanceur compare la valeur espérée pour ce processus contre la valeur du processus actuellement en exécution. Si le temps du nouveau processus est plus petit, il rentre en exécution immédiatement.

❑ Avantages

- Plus efficace que SJF, car le temps d'attente moyen optimal est garantit quelque soit le moment d'arrivée des processus

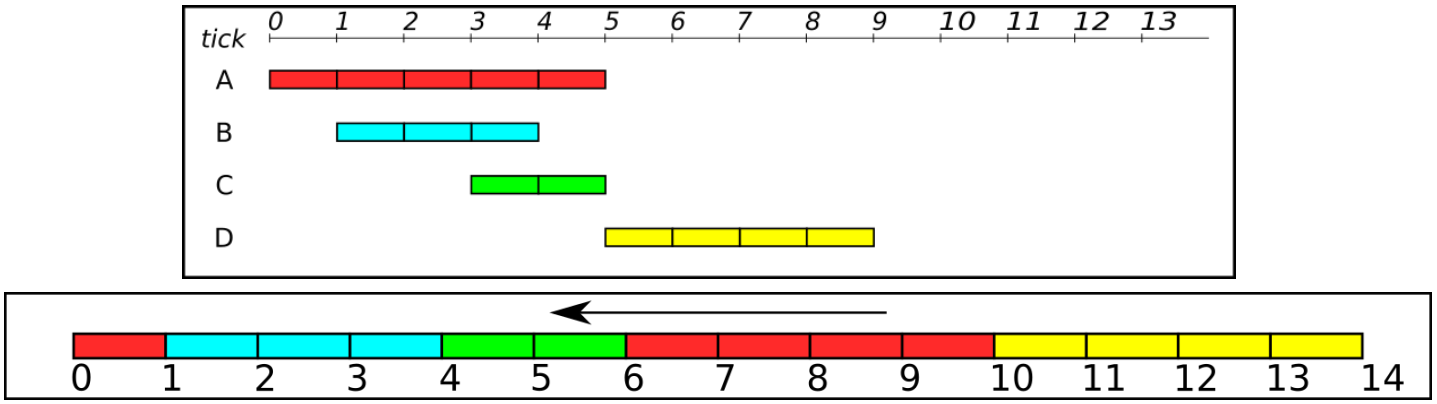
❑ Inconvénients

- Risque de famine
- Besoin de connaitre la durée du cycle UCT à l'avance

→ Les algorithmes Préemptifs

Shortest Remaining Time (SRT)

Exemple



- Au tick 0, le processus A est exécuté.
- Au tick 1, le processus B est estimé plus court que la partie restante de A (3 contre 4), donc A est suspendu et B est exécuté.
- Au tick 3, il reste 4 temps au processus A, 1 seul pour B, et 2 pour C, donc on termine B
- Au tick 4, on exécute C, qui est le plus court.
- Au tick 6, il reste 4 temps pour A et D, un choix est fait : comme A est déjà en cours, on le termine en priorité.

→ **Calculer le temps de séjour moyen et le temps d'attente moyen.**

→ Les algorithmes Préemptifs

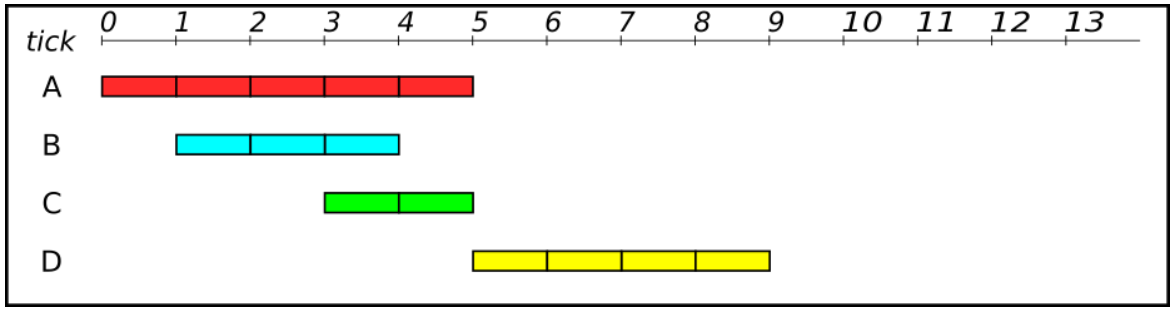
Algorithme du tourniquet ou Round Robin(RR)

- Le **Round Robin (RR)** un algorithme ancien, simple, fiable et très utilisé. Il mémorise dans une file du type FIFO la liste des processus prêts, c'est-à-dire en attente d'exécution.
- A chaque processus est allouée une tranche de temps, appelée **quantum** (généralement entre 10 et 100 ms.), pour s'exécuter.
- Le système alloue le processeur au processus en tête de file pour un quantum de temps donné.
  - → Si le processus se bloque ou se termine avant la fin de son quantum, le processeur est immédiatement alloué à un autre processus, celui en tête de file.
  - → Si le processus n'est pas terminé à la fin de son quantum, son exécution est temporairement suspendue, et le processeur est alloué à un autre processus.
  - → Le processus suspendu sera ensuite placé à la fin de la file d'attente, et le système allouera le processeur à un autre processus.

→ Les algorithmes Préemptifs

Algorithme du tourniquet ou Round Robin(RR)

Exemple



- **Si le quantum de temps est de 3 tick**
  - A est entré au tick 0 , et est terminé au tick 8
  - B est entré au tick 1 , et est terminé au tick 6
  - C est entré au tick 3 , et est terminé au tick 10
  - D est entré au tick 5 , et est terminé au tick 14

→ **Tracer le diagramme de Gantt, et calculer le temps moyen de séjour et le temps moyen d'attente.**

Ordonnancement des processus

→ Les algorithmes Préemptifs

Algorithme basé sur la priorité
Highest Priority First ou haute priorité d'abord (HPF)

Le principe de cet algorithme consiste à associer à chaque processus une priorité.

• Le processeur est alloué au processus ayant la plus grande priorité pendant un quantum de temps Q.

• Ce procédé est répété jusqu'à épuisement des processus se trouvant dans la file d'attente.

• C'est une généralisation des algorithmes basés sur les priorités **avec réquisition**.

• Les priorités peuvent être dynamiques. Elles sont, dans ce cas recalculé après chaque quantum de temps.

→ Les algorithmes Préemptifs

Exercice

Considérez les quatre processus suivants, prêts à être exécutés dans un système d'exploitation :

Processus	Temps d'arrivée	Temps CPU
A	0	10
B	2	5
C	5	13

Donner le diagramme de Gantt et calculer le temps moyen de séjour et le temps moyen d'attente pour les algorithmes :

1. SRT
2. RR avec un Quantum = 8
3. RR avec un Quantum = 4

Ordonnancement des processus

→ synthèse : La politique d'allocation de CPU

- La politique d'ordonnancement, également connue sous le nom de politique d'allocation de CPU, consiste à choisir parmi tous les processus prêts celui qui doit être exécuté en premier.
- La politique d'allocation de CPU peut être mise en œuvre selon différentes approches, chacune ayant ses avantages et ses inconvénients.
- **Il existe différentes politiques d'allocation :**

→ avec ou sans priorité

- **Sans priorité** : *FCFS*, SJF, SRT, RR…
- **Avec priorité** :
  - La priorité peut être fixe ou dynamique
  - il peut y avoir préemption, ou non

→ Préemptif ou non- préemptif

- **Non –Préemptif**: *FCFS*, SJF, HPF (non-préemptif).
- **Préemptif** : SRT, RR, HPF (préemptif).

Ordonnancement des processus

→ synthèse : Critères d'ordonnancement

- Maximiser **l'utilisation UCT** : pourcentage d'utilisation de l'UCT
- Maximiser le **débit** (throughput) : nombre de processus qui se terminent dans l'unité de temps
- Minimiser le **temps de rotation** ou temps de séjour (turnaround time) : temps de terminaison moins temps d'arrivée
- Minimiser le **temps d'attente** (waiting time) : temps de rotation (de séjour) moins le temps d'exécution
- Minimiser le **temps de réponse** (response time) =>pour les systèmes interactifs : le temps entre une demande et la réponse.

Synchronisation et communication entre processus

Synchronisation et communication entre processus

→ Introduction

- Les processus étant des entités autonomes et indépendantes, ils peuvent se trouver en **conflit** pour l'accès à certaines ressources communes.
- Cela nécessite la mise en place de mécanismes de **synchronisation** pour gérer ces conflits. Par ailleurs, bien que les processus soient généralement indépendants, cela ne doit pas empêcher la **communication** entre eux.
- Les conséquences de ces mécanismes sont le blocage des processus en attente d'une ressource momentanément indisponible. Deux problèmes en découlent : l'interblocage et la famine.

→ La synchronisation: Notion d'exclusion mutuelle entre processus

Problématiques (1/2)

- Considérons un compte bancaire, dont le montant est mémorisé dans un emplacement donné **A** sur disque. Le programme qui consiste à ajouter 100 à ce compte pourrait être le suivant, où N est une variable locale du programme:

```
lire (N, A);
N := N + 100;
écrire (N, A);
```

- Si deux processus différents consistent en l'exécution de ce même programme, le processeur sera alloué à chacun d'eux dans un ordre quelconque. En particulier, on peut imaginer que l'ordre soit le suivant:

```
processus P1
lire (N, A);


N := N + 100;
écrire (N, A);
```

```
processus P2

        lire (N, A);
        N := N + 100;
        écrire (N, A);
```

→ La synchronisation: Notion d'exclusion mutuelle

Problématiques (2/2)

N étant une variable locale du programme, implique qu'il en existe un exemplaire pour chacun des deux processus.

→ Il s'ensuit que, si la valeur initiale du compte est 1000, on constate qu'après ces exécutions, il est de 1100 au lieu de 1200.

→ Les deux processus ne sont pas en fait totalement indépendants. Ils partagent la ressource commune qu'est la valeur du compte bancaire à l'adresse **A** sur disque.

→ Cette ressource doit être à un seul point d'accès, c'est donc **une ressource critique,** et les processus sont en **exclusion mutuelle** sur cette ressource critique.

L'exclusion mutuelle est un mécanisme de synchronisation qui permet de garantir qu'à un instant donné, un seul processus accède à une ressource donnée.

→ Concepts fondamentaux

- **Section Critique**

Une section critique (SC) est un ensemble d'instruction d'un programme qui peuvent engendrer des résultats imprévisibles lorsqu'elles sont exécutées simultanément par des processus différents.
→ D'une manière générale on peut dire qu'un ensemble d'instructions peut constituer une section critique (SC) s'il y a des variables partagées ; et non dans l'absolu. Autrement dit, l'existence de section critique implique l'utilisation de variable partagées , mais l'inverse n'est pas vrai.

- **Ressource critique**

Une ressource critique désigne tout objet, qu'il s'agisse d'une variable, d'une table, d'un fichier, d'un périphérique, ou autre, pouvant être sujet à un accès concurrent par plusieurs processus.

Par exemple: deux processus qui tentent d'envoyer chacun, simultanément, un fichier sur l'imprimante ; le périphérique imprimante devient ressource critique pour eux.

→ Concepts fondamentaux: Exclusion mutuelle

- Deux processus ne peuvent pas utiliser la section critique (SC) en même temps.
- Pour contrôler l'accès et la sortie de la SC, les étapes suivantes doivent être suivies :
    1. Réserver l'accès si la SC est libre ou attendre
    2. Utiliser la SC.
    3. Signaler la sortie de la SC aux autres processus.
- **L'exclusion mutuelle** des processus est un mécanisme qui permet de garantir qu'un seul processus à la fois peut accéder à une ressource partagée. Cela évite que plusieurs processus tentent de modifier ou d'utiliser une ressource simultanément.
- **L'exclusion mutuelle** est essentielle pour éviter les conflits et les incohérences lors de l'accès à des ressources partagées.

→ Règle de la synchronisation

Les règles de synchronisation sont les suivantes :

- **Exclusion mutuelle**
- **Pas de famine** : aucun processus ne doit attendre trop longtemps pour entrer en section critique.
- **Pas d'interblocage** : aucun processus suspendu en dehors des sections critiques ne doit bloquer les autres processus
- **Aucune hypothèse ne doit être faite sur les vitesses relatives des processus**. Cela signifie que les processus peuvent être exécutés à des vitesses différentes, et que le système doit être conçu pour prendre en compte cette variation de vitesse.

→ Les mécanismes de synchronisation

Les solutions courantes pour assurer la **synchronisation** des processus sont les techniques et méthodes pour garantir un accès sécurisé et cohérent aux ressources partagées par plusieurs processus en même temps. Ces solutions incluent :

- **Masquage des interruptions**
- **Solutions avec attente active *(busy waiting)*:**
  - → Verrous
  - → Test and Set Lock (TSL)
  - → Alternance
  - → Alternance de Peterson
- **Solutions sans attente active :**
  - → Les sémaphores
  - → sleep et wakeup
  - → Les moniteurs
  - → La communication par message

→ Les mécanismes de synchronisation

1. Masquage des interruptions

- La première solution de **masquage des interruptions** consiste à masquer les interruptions avant d'entrer en section critique et de les restaurer après.
- Cette solution présente des limites. Elle ne permet pas la commutation du processeur puisque l'interruption horloge est ignorée. Elle est inadaptée aux systèmes multiprocesseurs puisqu'un processus concurrent peut accéder à la section critique s'il est exécuté par un autre processeur.
- Elle est aussi dangereuse vu qu'il y a risque que le système soit bloqué en cas d'oubli de restauration des interruptions.
- Elle est inéquitable car tous les processus se trouvent en attente et ne peuvent prendre leur part de CPU même s'ils ne sont pas concernés et ne posent pas de problème d'accès concurrent.
- Cette solution peut également bloquer un processus prioritaire.

→ Les mécanismes de synchronisation

2. Solutions avec attente active : Les verrous

- **Un verrou** (en anglais "lock") est un objet système qui permet de contrôler l'accès à une ressource partagée entre plusieurs processus. Cette ressource peut être considérée comme une ressource logicielle.
- Les verrous sont utilisés pour éviter les conflits d'accès concurrents à la ressource partagée.

→ **Deux opérations** sont définies pour manipuler un verrou :

- ***verrouiller*** (ou "lock" en anglais) : cette opération permet à un processus d'acquérir le verrou s'il est disponible. Si le verrou n'est pas disponible, le processus est mis en attente jusqu'à ce que le verrou soit libéré par le processus qui le détient déjà.
- ***déverrouiller*** (ou "unlock" en anglais) : cette opération permet à un processus de libérer le verrou qu'il possède. Si d'autres processus étaient en attente de ce verrou, l'un d'entre eux est réactivé et obtient le verrou libéré.

→ L'utilisation de verrous permet de synchroniser les accès concurrents à une ressource partagée et ainsi d'éviter les problèmes de concurrence. Cependant, il est important de bien gérer les verrous pour éviter les situations de blocage ou de deadlock.

→ Les mécanismes de synchronisation

2. Solutions avec attente active : Les verrous

- **Exemple:**

Algorithme

```
while (verrou == 1) ; /*Attente active*/
verrou=1 ; /*Verrouillage*/
section_critique();
verrou=0 ; /*Déverrouillage*/
```

Si le verrou est déjà pris par un autre processus, le processus en attente entre dans une boucle d'attente active, qui consiste à vérifier périodiquement si le verrou est disponible.

→ Les mécanismes de synchronisation

2. Solutions avec attente active : TSL

- **TSL** (***Test and Set Lock***) est une solution avec attente active pour assurer la synchronisation des processus. Elle consiste en l'utilisation d'une variable **binaire** partagée (le verrou/lock).
- Cette variable peut avoir deux états : verrouillé (lock) ou déverrouillé (unlock).
- Lorsqu'un processus souhaite accéder à une ressource partagée, il doit d'abord **tenter de verrouiller le verrou** à l'aide de l'instruction **TSL**.

  → Si le verrou est **unlock**, le processus le verrouille et entre en section critique.

  → Si le verrou est **lock**, le processus est mis en attente active (loop d'attente) jusqu'à ce que le verrou soit libéré par un autre processus.

  → Une fois le processus sorti de la section critique, il déverrouille le verrou en affectant à la variable binaire la valeur **unlock**.

→ Les mécanismes de synchronisation

2. Solutions avec attente active : Alternance

- **L'alternance** est une solution avec attente active qui consiste à utiliser une variable partagée (par exemple un booléen) pour permettre à deux processus d'entrer en section critique alternativement.
- Chaque processus vérifie périodiquement la valeur de la variable partagée et attend si elle est occupée par l'autre processus. Lorsqu'un processus termine sa section critique, il change la valeur de la variable partagée pour permettre à l'autre processus d'entrer.
- **Exemple**

→ Algorithme d'utilisation pour N processus

```
While (1) {
while (tour != monNumero) ; /*Attente active*/
Section_critique();
tour = (monNumero +1) % N ; /*Au suivant*/
Suite du programme }
```

→ Les mécanismes de synchronisation

2. Solutions avec attente active : Alternance de Paterson

- La solution avec attente active de l'alternance de Peterson est un algorithme de synchronisation qui permet à deux processus d'entrer en section critique en alternance, en utilisant des variables partagées telles que "**turn**" et "**flag**".
- L'idée de base est que chaque processus doit attendre que l'autre processus soit prêt à entrer en section critique avant de prendre le contrôle de la ressource partagée.

→ Les mécanismes de synchronisation

2. Solutions avec attente active : Alternance de Paterson

- **Exemple:**

```
int turn;                                  flag[0] = false;
boolean flag[2];                           // Processus 1
// Processus 0                             flag[1] = true;
flag[0] = true;                            turn = 0;
turn = 1;                                  while (flag[0] && turn == 0) {
while (flag[1] && turn == 1) {               // Attendre }
  // Attendre }                            // Entrer en section critique
// Entrer en section critique              ...
...                                        // Sortir de la section
// Sortir de la section                    critique
critique                                   flag[1] = false;
```

- Dans cet exemple, les processus 0 et 1 utilisent les variables partagées "turn" et "flag" pour se synchroniser et entrer en section critique en alternance.
- Chaque processus attend que l'autre processus soit prêt à entrer en section critique avant de prendre le contrôle de la ressource partagée. Si l'autre processus n'est pas prêt ou si ce n'est pas son tour, le processus en attente continue d'attendre.

→ Les mécanismes de synchronisation

3. Solutions sans attente active : Les sémaphores

- Un ***sémaphore*** est un mécanisme proposé par E.W. Dijkstra en 1965, et qui est un peu plus général que le verrou. Il se présente comme un distributeur de jetons, mais le nombre de jetons est fixe et non renouvelable: les processus doivent restituer leur jeton après utilisation.
- Lorsqu'il n'y a qu'un seul jeton en circulation, cela revient à utiliser un **verrou**. Les deux opérations sur un sémaphore sont traditionnellement notées ***P(s)*** et ***V(s)***.

  → ***P(s)*** ou ***down(s)*** permet à un processus d'obtenir un jeton, s'il y en a de disponibles. Si aucun n'est disponible, le processus est bloqué.

  → ***V(s)*** ou ***up(s)*** permet à un processus de restituer un jeton. Si des processus étaient en attente de jeton, l'un d'entre eux est réactivé et le reçoit.

→ Les mécanismes de synchronisation

3. Solutions sans attente active : Les sémaphores

- La fonction principale d'un sémaphore est de contrôler l'état d'une ressource partagée afin que les opérations effectuées sur celle-ci soient synchronisées.
- Une valeur de jetons est attribuée pour indiquer le nombre d'opérations qui peuvent utiliser la ressource simultanément.
- Lorsqu'un processus souhaite utiliser la ressource, il réduit la valeur du sémaphore associée à cette ressource.

→ Si la valeur est égale à zéro, cela signifie qu'aucun jeton n'est disponible et que le processus doit attendre jusqu'à ce qu'un nouveau jeton soit ajouté au sémaphore.

→ Les mécanismes de synchronisation

3. Solutions sans attente active : Les sémaphores

- **Exemple:**
- Notons qu'un sémaphore peut être vu comme un couple constitué d'un entier, encore appelé le **niveau du sémaphore** et d'une **file d'attente de processus**.
  - Le niveau est le nombre de jetons encore disponibles. Il est évident que si le niveau est positif, la file d'attente est vide.
  - Parfois on utilise les valeurs négatives du niveau pour représenter le "déficit" en jetons, c'est à dire le nombre de processus en attente de jeton.
- À la création d'un sémaphore, il faut décider du nombre de jetons dont il dispose. On voit que si une ressource est à un seul point d'accès (critique), le sémaphore doit avoir initialement 1 jeton, qui sera attribué successivement à chacun des processus demandeurs.
  - → Si une ressource est à ***n*** points d'accès, c'est-à-dire, peut être utilisée par au plus ***n*** processus à la fois, il suffit d'un sémaphore initialisé avec ***n*** jetons.
  - → Dans les deux cas, un processus qui cherche à utiliser la ressource, demande d'abord un jeton, puis utilise la ressource lorsqu'il a obtenu le jeton et enfin rend le jeton lorsqu'il n'a plus besoin de la ressource.

→ Les mécanismes de synchronisation

3. Solutions sans attente active : Sleep et Wakeup

- La solution sans attente active "***sleep*** et ***wakeup***" permet à un processus de se mettre en sommeil pendant un certain temps sans occuper activement le processeur.
- Lorsqu'un processus a besoin d'accéder à une ressource partagée, il peut vérifier si la ressource est libre. Si la ressource est occupée, le processus peut appeler la fonction "***sleep***" pour se mettre en attente.
- Une fois que la ressource est disponible, un autre processus peut appeler la fonction "***wakeup***" pour réveiller le processus en attente et lui donner accès à la ressource. Cette solution est efficace car elle évite l'attente active et permet à d'autres processus d'utiliser le processeur pendant que le processus en attente est en sommeil.

→ Les mécanismes de synchronisation

3. Solutions sans attente active : Les moniteurs

- Les moniteurs ont un ensemble de procédures qui sont accessibles uniquement à travers un verrou de moniteur, qui garantit que seul un processus à la fois peut accéder aux procédures du moniteur.

  → Lorsqu'un processus souhaite accéder à la ressource partagée, il doit d'abord acquérir le verrou de moniteur. Si la ressource n'est pas disponible, le processus est mis en attente, mais contrairement à l'attente active, cette attente est passive et ne consomme pas de ressources du processeur.

  → Lorsque la ressource devient disponible, le processus en attente est réveillé et le verrou de moniteur est libéré. Les autres processus en attente peuvent ensuite tenter d'acquérir le verrou de moniteur à leur tour.

→ Les mécanismes de synchronisation

3. Solutions sans attente active : La communication par message

- La communication par message est une solution sans attente active pour la synchronisation de processus. Au lieu d'attendre l'accès à une ressource partagée, les processus communiquent entre eux via l'échange de messages pour coordonner leurs actions.
- Dans cet algorithme, chaque processus dispose de sa propre mémoire et les communications sont gérées par le système d'exploitation qui s'assure que les messages sont bien acheminés vers les destinataires appropriés. Cette méthode est utilisée dans les systèmes distribués où les processus s'exécutent sur différentes machines et communiquent via le réseau.
- La communication par message est considérée comme une solution plus efficace et moins sujette aux problèmes de concurrence que les solutions avec attente active telles que les verrous.

→ La communication inter-processus

❑ Définition

La communication entre les processus (***IPC*** en anglais pour Inter-Process Communication) est une technique qui permet à différents processus de communiquer et de partager des ressources dans un système informatique.

→ L'IPC est un élément important dans la conception de systèmes d'exploitation multitâches et multiprocesseurs, car il permet aux différents processus de travailler ensemble pour accomplir des tâches plus complexes.

Synchronisation et communication entre processus

→ La communication entre les processus

Il existe différentes **méthodes de communication** entre les processus, notamment :

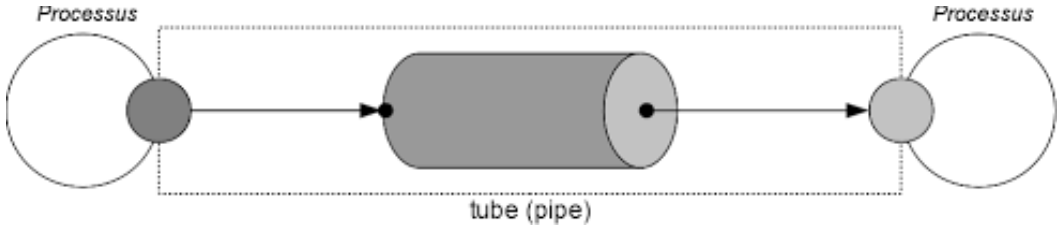
1. **Les tubes (les pipes)** : est un canal unidirectionnel de communication qui permet à un processus d'écrire des données dans le tube, tandis qu'un autre processus peut les lire à partir du tube.
2. **Les signaux** : Les processus peuvent envoyer des signaux à d'autres processus pour leur indiquer certains événements ou déclencher des actions.
3. **Les files d'attente de messages** : Les processus peuvent envoyer et recevoir des messages via des files d'attente de messages partagées.
4. **Les mémoires partagées** : Les processus peuvent partager des zones de mémoire communes pour échanger des informations.
5. **Les sockets** : Les processus peuvent communiquer via des sockets, qui permettent des communications réseau entre des machines différentes.

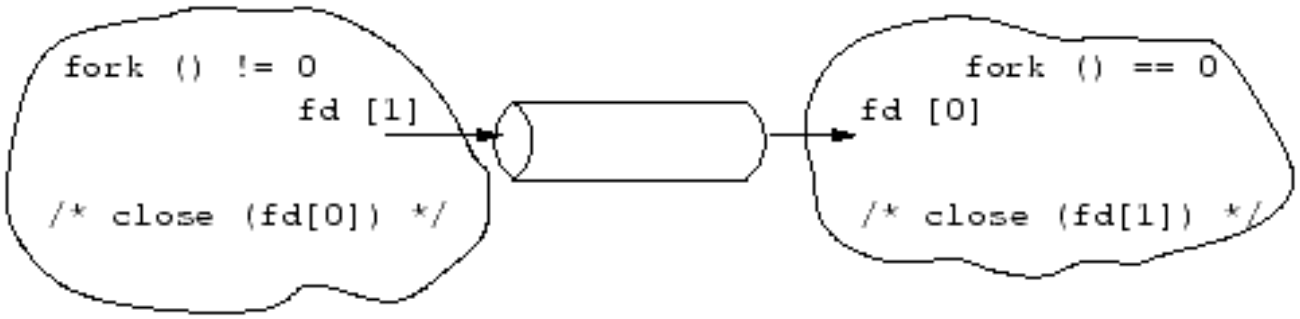
→ La communication par tubes sous Linux

Un tube (pipe en anglais) est un moyen de transmission de données d'un processus à un autre.

→ Il existe deux types de tube:

- les tubes **anonymes**, encore appeler tubes **ordinaires**
- les tubes **nommés**





→ La communication par tubes sous Linux

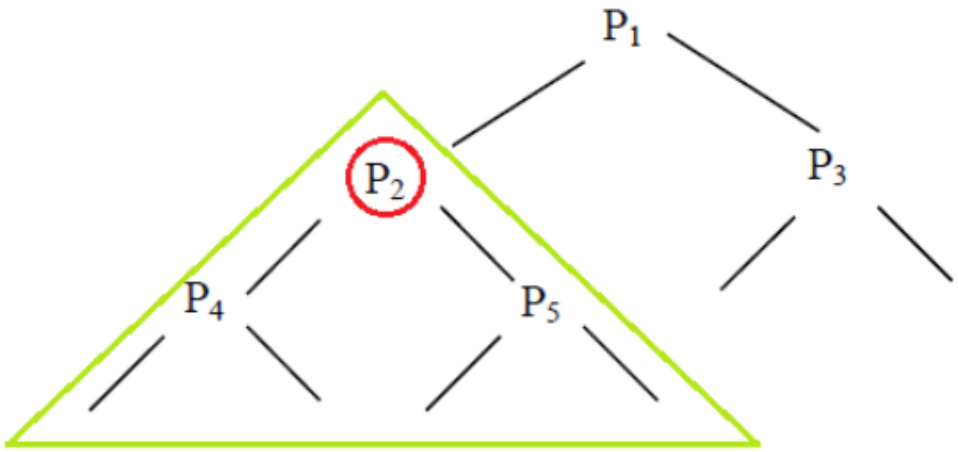
Les principales caractéristiques d'un pipe ordinaire sont les suivantes :

- Un pipe est de nature **FIFO** (First In First Out).
- Un pipe est **unidirectionnel**, c'est-à-dire qu'un processus peut soit lire soit écrire dans un pipe, mais pas les deux en même temps.
- Les données transmises à travers un pipe sont **éphémères**, c'est-à-dire qu'elles disparaissent après avoir été lues et que le pipe lui-même est détruit après que le dernier processus à y avoir accédé l'ait fermé.
- La **taille** d'un pipe est limitée, généralement **de 4k à 20k**, en fonction du matériel.
- Un pipe est un objet de type fichier associé à deux descripteurs de fichiers et deux entrées dans la table des fichiers ouverts.
- La communication via un pipe ordinaire n'a lieu qu'entre des processus de la même famille, c'est-à-dire des processus qui ont hérité des descripteurs de fichier du même processus père.

→ La communication par tubes sous Linux

→ Les tubes ordinaires

Exemple:



- Les pipes crées par le processus P2 ne permettent la communication qu'entre les processus **P2**, **P4**, **P5** et leurs descendances.
- Les processus **P1** et **P3** ne peuvent pas accéder à ces pipes.

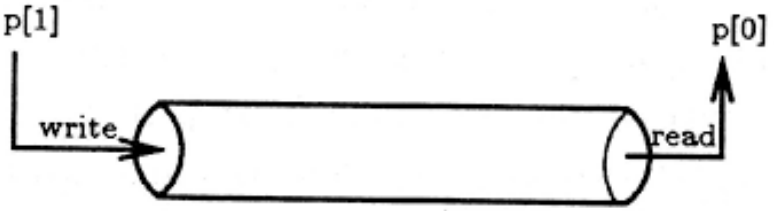
→ La communication par tubes sous Linux

Création d'un tube ordinaire, lecture et écriture

- **Primitive pipe() :** permet de créer un canal de communication (pipe) unidirectionnel entre deux processus.
  Le prototype:

```
#include <unistd.h>
int pipe(int p[2]);
```

- Crée un pipe (tube) et lui associe deux descripteurs de fichier dans le tableau de deux entiers p.
- Par définition **p[0]** est le descripteur pour ***lire*** (sortie du pipe) et **p[1]** celui pour ***écrire*** (entrée du pipe).
- La valeur retour de **pipe()** est **0** en cas de succès, **-1** sinon (trop de descripteurs actifs, ou de fichiers ouverts, etc...).



→ La communication par tubes sous Linux

- **Primitive read():** permet de lire dans un pipe en utilisant le descripteur **p[0]** retourné par **pipe()**
  Le prototype:

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

  → **fd** : le descripteur de fichier à partir duquel la lecture doit être effectuée.
  → **buf** : le tampon dans lequel les données lues doivent être stockées.
  → **count** : le nombre maximal d'octets à lire.

- **Primitive write() :** permet d'écrire dans un pipe. L'écriture est faite en utilisant le descripteur **p[1]**
  Le prototype:

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

  → **fd** : le descripteur de fichier dans lequel l'écriture doit être effectuée.
  → **buf** : le tampon à partir duquel les données doivent être écrites.
  → **count** : le nombre d'octets à écrire.

- La fermeture est effectuée par **close()** .

Exemple

Dans cet exemple, le processus parent envoie un message au processus enfant via le **pipe** et le processus enfant affiche le message reçu

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
void main() {
    int fd[2];
    pid_t pid;
    char message[] = "Bonjour, enfant !\n";
    char buffer[100];
     // Créer le pipe
    pipe(fd);
    pid = fork();
    if (pipe(fd) == -1 || pid == -1 ) {
        printf("Erreur");
        exit(1);
    }
    if (pid == 0) { // processus enfant
        close(fd[1]); //Fermeture du descripteur de fichier d'écriture
        read(fd[0], buffer, sizeof(buffer));// Lecture de la chaîne de caractères
        printf("Message reçu par le processus enfant : %s", buffer);
        close(fd[0]); //Fermeture du descripteur de fichier de lecture
    } else { // processus parent
        close(fd[0]); //Fermeture du descripteur de fichier de lecture
        write(fd[1], message, strlen(message));
        close(fd[1]); //Fermeture du descripteur de fichier d'écriture
    }
}
```

→ La communication par tubes sous Linux

→ Les tubes nommés

- Un **pipe nommé** a les mêmes caractéristiques qu'un pipe ordinaire. En plus, il a un nom sous Unix comme un fichier ordinaire.
- Il est crée de la même manière qu'un fichier spécial avec **mknod** ou **mkfifo**, Ensuite, il peut être ouvert avec la primitive **open**.
- Il est accessible par les processus n'ayant pas de lien de de parenté.
- Il subit les mêmes règles qu'un fichier ordinaire. Donc, comme son nom l'indique, le pipe dispose d'un nom dans le système de fichier. Il suffit qu'un processus l'appelle par son nom, et il donne droit au processus appelant de lire ou écrire en son intérieur.

→ Un pipe nommé est crée par la commande **mkfifo** ou **mknod**, ou par l'appel système **mknod()** ou **mkfifo()**.

→ La communication par signaux sous Linux

- **Un signal** correspond à une interruption logicielle: c'est un moyen de communication soit entre deux processus utilisateurs, soit entre le système d'exploitation et un processus.
- Il est d´défini par un code numérique dans le fichier: **/usr/include/sys/signal.h** ou dans **/usr/include/bits/signum.h**
- La communication par signaux entre deux processus se fait en deux étapes:
  - L'envoi du signal par le processus **émetteur**.
  - La prise en compte du signal par le processus **récepteur**.
- L'**émission** du signal est volontaire et donc prévisible dans le déroulement chronologique du processus émetteur : c'est pour lui un évènement **synchrone**.
- La **réception** de ce signal par le processus récepteur peut arriver à n'importe quel moment, c'est donc un évènement **asynchrone**.

→ La communication par signaux sous Linux

→ Notions sur les signaux

- Contrairement aux autres mécanismes de communication inter-processus, les signaux ne **transportent pas de données**, mais indiquent des actions à effectuer dans certaines circonstances.
- Un processus ne peut connaître l'identité du processus qui lui a envoyé le signal.
- Les signaux sont identifiés par un **numéro entier** et un **nom symbolique** décrit dans **signal.h**.
- La commande: `$ kill –l` donne la liste des signaux d'un système Unix.

→ La communication par signaux sous Linux

→ Types de signaux

Tous les signaux ont une routine de service, ou une action par défaut. Cette action peut être du type :

- **A** : terminaison du processus
- **B** : ignorer le signal
- **C** : Créer un fichier **`core`**
- **D** : Stopper le processus
- **E** : La procédure de service ne peut pas être modifiée
- **F** : Le signal ne peut pas être ignoré

- Le noyau UNIX/Linux admet **64 signaux** différents.

  → Les signaux **1** à **31** correspondent aux signaux **classiques.**
  → Les numéros **32** à **63** correspondent aux signaux **temps réel.**

→ La communication par signaux sous Linux

→ Liste des signaux

La liste de quelques signaux classiques (on peut utiliser la commande **$ kill –l** pour les lister)

Numéro	Nom	Description
1	**SIGHUP**	Instruction (HANG UP) - Fin de session
2	**SIGINT**	Interruption du clavier (frappe de « Ctrl+C »)
3	**SIGQUIT**	Caractère « quit » frappé depuis le clavier (Frappe de « Ctrl \ »)
4	**SIGILL**	Instruction illégale
5	**SIGTRAP**	Trace trap (Point d'arrêt pour le débogage)
6	**SIGIOT /SIGABRT**	Terminaison anormale
7	**SIGBUS**	Erreur de bus
8	**SIGFPE**	Erreur mathématique virgule flottante
9	**SIGKILL**	Terminaison forcée du processus
10	**SIGUSR1**	Signal utilisateur 1

La liste de tous les signaux classiques se trouve dans l'annexe.

→ La communication par signaux sous Linux

- Un signal peut être envoyé:
  1. Lors de la constations d'une anomalie matérielle.
  2. Suite à la frappe d'une combinaison de touches (par ex. ***CTRL-C*** pour envoyer le signal **SIGINT**).
  3. Par un autre processus utilisant la commande **kill** ou l'appel système **kill()**.

- Lors de la réception d'un signal, il y a **trois actions** par défaut suivant le signal :

  → **Ignorer** le signal

  → **Terminer** le programme ou terminer en créant un fichier **`core`**.

  → **Un processus peut changer son comportement** par défaut lors de la réception d'un signal en déroutant le signal, c'est-à-dire en indiquant la fonction à exécuter lors de la réception du signal.

Le fichier **`core`** est un fichier de débogage généré automatiquement lorsqu'un programme/processus se termine de manière anormale ou rencontre une erreur fatale.

→ La communication par signaux sous Linux

→ Emission d'un signal: kill()

- La primitive **kill():** permet l'émission du signal de numéro **sig** à destination du processus de numéro **pid**. Prototype:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

- Cet appel système prend deux arguments: Le **second** est toujours le numéro du signal à délivrer ou la constante définissant ce signal dans **`signal.h`**. Le **premier argument** indique à quel(s) processus le signal doit être délivré :
  - pid>0 : le signal est délivré au processus ayant comme identifiant pid.
  - pid==0 : le signal est destiné à tous les processus de même groupe que le processus émetteur à l'exception des processus spéciaux : init, swapper, sched . . .
  - pid==-1. le signal est délivré à tous les processus sauf les processus spéciaux
  - pid<-1. le signal est envoyé à tous les processus dont l'identificateur de groupe de processus est égal à la valeur absolue de pid: abs(pid).
- **Kill** retourne un **0** s'il n'y a pas d'erreur et **-1** dans le cas d'erreur.

→ La communication par signaux sous Linux

→ Traitement d'un signal: signal()

❑ La primitive **signal() :** permet d'attacher un gestionnaire de traitement à un signal.
Prototype:

```
#include <signal.h>
void(*signal(int sig,void(*func)(int)))(int) ;
```

- **sig** : représente le numéro ou le nom du signal.
- **func** : c'est la fonction à exécuter, à l'arrivé du signal.
- Le **Handler** (la procédure) défini par ***func** est attaché au signal **sig**, désigné par le nom qui lui est associé.
- L'argument **func** peut prendre trois valeurs distinctes :
  - **func = SIG_DFL**: l'action par défaut est attaché au signal **sig**.
  - **func = SIG_IGN**: alors le signal **sig** est ignoré, elle est utilisé pour les processus en arrière plan.
  - **func** est un pointeur vers une procédure **Handler** définie dans le code utilisateur. La prise en compte du signal **sig** par le processus entraine l'exécution de cette procédure.
- En cas d'échec, la fonction signal() renvoie la valeur **SIG_ERR**.

→ La communication par signaux sous Linux

→ Attente d'un signal

- L'appel système **pause()** suspend l'appelant jusqu'au prochain signal.
  Prototype:

```
#include <unistd.h>
int pause (void) ;
```

- L'appel système **sleep(v)** suspend l'appelant jusqu'au prochain signal ou l'expiration du délai (v en secondes).
  Prototype:

```
#include <unistd.h>
void sleep (int );
```

→ La communication par signaux sous Linux

→ Armer une temporisation

❑ La primitive **alarm()** permet à un processus d'armer une temporisation.

- A l'issue de cette temporisation le signal **SIGALRM** est délivré au processus. Le comportement par défaut est d'arrêter l'exécution du processus.

  Le prototype de la fonction est :

```
#include <unistd.h>
Unsigned int alarm (unsigned int nb_sec);
```

- Une temporisation d'une durée égale à ***nb_sec*** seconds est armée.

Remarque: L'opération **alarm(0)** annule une temporisation précédemment armée.

→ La communication par signaux sous Linux

→ Traitement d'un signal: signal()

❑ Exemple 1

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void handle_signal(int signal) {
    printf("Signal %d reçu.\n", signal);
}

int main() {
    signal(SIGINT, handle_signal);
    printf("En attente du signal SIGINT...\n");
    while(1) {
        sleep(1);
    }
    return 0;
}
```

la fonction signal() est utilisée pour spécifier la fonction **handle_signal()** comme fonction de traitement de signal pour **SIGINT**.

Lorsque le processus reçoit le signal **SIGINT** (par exemple en appuyant sur Ctrl+C dans la console), la fonction **handle_signal()** est exécutée pour traiter le signal.

Le programme continue ensuite de s'exécuter normalement jusqu'à ce qu'il soit interrompu ou qu'il se termine.

→ La communication par signaux sous Linux

→ Emission d'un signal: kill()

❑ Exemple 2

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

void signal_handler(int signo) {
    if (signo == SIGUSR1) {
        printf("Le processus fils a reçu le signal SIGUSR1.\n");
    }
}
int main() {
    pid_t pid;
    pid = fork();
    if (pid < 0) {
        printf("Erreur lors de la création du processus fils");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        signal(SIGUSR1, signal_handler);  // Définir le gestionnaire de signal pour le fils
        printf("Le processus fils est en attente du signal SIGUSR1...\n");
        while (1) { }
    } else { // Code du processus parent
        printf("Le processus parent envoie le signal SIGUSR1 au processus fils.\n");
        kill(pid, SIGUSR1);  // Envoyer le signal SIGUSR1 au fils
        printf("Signal SIGUSR1 envoyé au processus fils.\n");
        // Attendre la fin du processus fils (facultatif)
        wait(NULL);
    }
    return 0;
}
```

Le processus fils attend indéfiniment jusqu'à ce qu'il reçoive un signal.

Le processus parent envoie ensuite le signal **SIGUSR1** au processus fils en utilisant la fonction **kill**().

Exercice 3

Écrire un programme en C qui crée un tube anonyme, puis crée deux processus enfants.

- Le premier processus doit **lire** un message à partir du tube et l'afficher.
- Le deuxième processus doit **envoyer** un message au tube.
- Le processus parent doit attendre que les processus enfants se terminent avant de se terminer lui-même.

Exercice 4

Écrivez un programme en C qui utilise la fonction signal pour capturer le signal **SIGINT** (généré par la combinaison de touches ***Ctrl+C***) et afficher un message d'erreur lorsque ce signal est reçu.

ANNEXES

ANNEXE : Gestion des processus

Exercice 1

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

int main() {
    pid_t pid;

    pid = fork(); // création d'un nouveau processus fils

    if (pid == 0) { // code exécuté par le processus fils
        printf("Bonjour, je suis le processus fils (pid=%d)\n", getpid());
    } else if (pid > 0) { // code exécuté par le processus parent
        printf("Bonjour, je suis le processus parent (pid=%d)\n", getpid());
    } else { // gestion d'erreur si la création du processus fils a échoué
        printf("Erreur de création du processus fils\n");
    }
    return 0;
}
```

ANNEXE : Gestion des processus

Exercice 2

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

int main() {
    pid_t pid1, pid2;

    pid1 = fork(); // création du premier processus fils

    if (pid1 == 0) { // code exécuté par le premier processus fils
        printf("Bonjour, je suis le premier processus fils (pid=%d)\n", getpid());
    } else if (pid1 > 0) { // code exécuté par le processus parent
        pid2 = fork(); // création du deuxième processus fils

        if (pid2 == 0) { // code exécuté par le deuxième processus fils
            printf("Bonjour, je suis le deuxième processus fils (pid=%d)\n", getpid());
        } else if (pid2 > 0) { // code exécuté par le processus parent
            printf("Bonjour, je suis le processus parent (pid=%d)\n", getpid());
        } else { // gestion d'erreur si la création du deuxième processus fils a échoué
            printf("Erreur de création du deuxième processus fils\n");
        }
    } else { // gestion d'erreur si la création du premier processus fils a échoué
        printf("Erreur de création du premier processus fils\n");
    }
    return 0;
}
```

ANNEXE

→ Liste des signaux classiques

Numéro	Nom	Description
1	**SIGHUP**	Instruction (HANG UP) - Fin de session
2	**SIGINT**	Interruption du clavier (frappe de « Ctrl+C »)
3	**SIGQUIT**	Caractère « quit » frappé depuis le clavier (Frappe de « Ctrl \ »)
4	**SIGILL**	Instruction illégale
5	**SIGTRAP**	Trace trap (Point d'arrêt pour le débogage)
6	**SIGIOT /SIGABRT**	Terminaison anormale
7	**SIGBUS**	Erreur de bus
8	**SIGFPE**	Erreur mathématique virgule flottante
9	**SIGKILL**	Terminaison forcée du processus
10	**SIGUSR1**	Signal utilisateur 1
11	**SIGSEGV**	Violation de mémoire (Référence mémoire invalide)
12	**SIGUSR2**	Signal utilisateur 2

ANNEXE

→ Liste des signaux classiques

Numéro	Nom	Description
13	**SIGPIPE**	Ecriture dans un tube sans lecteur
14	**SIGALRM**	Horloge temps réel, fin de temporisation
15	**SIGTERM**	Signal de terminaison
16	**SIGSTKFLT**	Stack Fault (Erreur de pile du coprocesseur)
17	**SIGCHLD** ou **SIGCLD**	Processus fils terminé
18	**SIGCONT**	Demande de reprise du processus
19	**SIGSTOP**	Stoppe l'exécution d'un processus
20	**SIGTSTP**	Demande de suspension depuis le clavier
21	**SIGTTIN**	lecture terminal en arrière-plan
22	**SIGTTOU**	écriture terminal en arrière-plan

ANNEXE

→ Liste des signaux classiques

Numéro	Nom	Description
23	**SIGURG**	évènement urgent sur socket
24	**SIGXCPU**	temps maximum CPU écoulé
25	**SIGXFSZ**	taille maximale de fichier atteinte
26	**SIGVTALRM**	alarme horloge virtuelle
27	**SIGPROF**	Profiling alarm clock
28	**SIGWINCH**	changement de taille de fenêtre
29	**SIGPOLL (System V)**	occurrence d'un évènement attendu
30	**SIGPWR**	Chute d'alimentation
31	**SIGSYS**	Erreur d'appel système
32	**SIGUNUSED**	Non utilisé