Page1 Gst mémoire :

mémoire : tout composant électronique capable de stocker des données.

La mémoire est assemblage de cellules repérées par leur numéro, ou adresse.

Les caractéristiques d'une mémoire : La capacité, représentant le volume global d'informations (en bits) que la mémoire peut stocker ; **Le temps d'accès** : correspondant à l'intervalle de temps entre la demande de lecture/écriture et la disponibilité de la donnée ; **Le temps de cycle**, représentant l'intervalle de temps minimum entre deux accès successifs ; **Le débit**, définissant le volume d'information échangé par unité de temps, exprimé en bits par seconde ; **La non volatilité** caractérisant l'aptitude d'une mémoire à conserver les données lorsqu'elle n'est plus alimentée électriquement.

Tyoe de memoires : **Mémoire centrale** : Mémoire primaire = Mémoire vive = RAM : volatile, c'est-à-dire quelle ne peut stocker les informations que lorsque L'ordinateur est allumé , peut être lue et modifiée à volonté.Permettant de mémoriser temporairement les données lors de l'exécution des programmes.

Mémoire de masse : (appelée également *mémoire physique ou mémoire externe*) **permettant de stocker des informations à long terme, y compris lors de l'arrêt de l'ordinateur.**

Elle correspond: - Aux dispositifs de stockage magnétiques, tels que le disque dur,

- Aux dispositifs de stockage optique, correspondant par exemple aux CD-ROM ou aux DVD-ROM, Ainsi qu'aux mémoires mortes, tels que la ROM. **Mémoire physique** est constituée d'un ensemble de mots mémoire contigus désignés chacun par une adresse physique.

Système multiprogrammé : syst permet de lancer plusieurs programmes à fois comme Windows . Dans un système multiprogrammé, trois problèmes sont à résoudre vis-à-vis de la mémoire centrale : 1.Il faut définir un espace d'adressage indépendant pour chaque processus 2.Il faut protéger les espaces d'adressage des processus les uns vis-à-vis des autres 3.Il faut allouer de la mémoire physique à chaque espace d'adressage.

Les méthodes d'allocation mémoire peuvent être divisées en deux grandes familles : pour **la première famille**, un programme est un ensemble de mots contigus insécables(non divisés). Pour **la seconde famille**, un programme est un ensemble de mots contigus sécable, c'est-à-dire que le programme peut être divisé en plus petits morceaux, chaque morceau étant lui-même un ensemble de mots contigus. Chaque morceau peut alors être alloué de manière indépendante. On trouve ici les mécanismes de **segmentation** et de **pagination**.

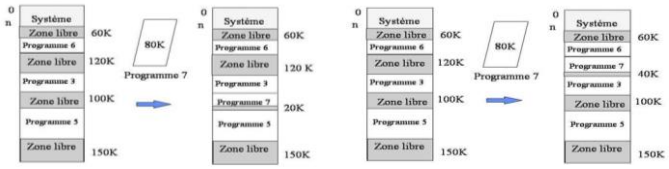
Allocation mémoire d'un seul tenant : Dans cette méthode d'allocation, le programme est considéré comme un espace d'adressage insécable. **La mémoire physique** est découpée en zones disjointes de taille variable, adaptables à la taille des programmes : ces zones sont appelés des partitions .

Initialement, la mémoire centrale est uniquement occupée par les procédures du système d'exploitation.

La zone réservée aux programmes utilisateurs est vide et constitue une unique zone libre.

Au fur et à mesure des chargements de programmes, la zone libre va se réduire et à l'instant t, elle n'occupe plus qu'une fraction de la mémoire central.

Allocation en partitions variables : Lorsque l'exécution des programmes se termine (ici P2 et P4), la mémoire est libérée : il se crée alors pour chaque zone libérée, une nouvelle zone libre. Finalement, la mémoire centrale se retrouve constituée d'un ensemble de zones allouées et de zones libres réparties dans toute la mémoire. Les zones libres sont organisées en une liste chaînée de zones libres repérée par une tête de liste.



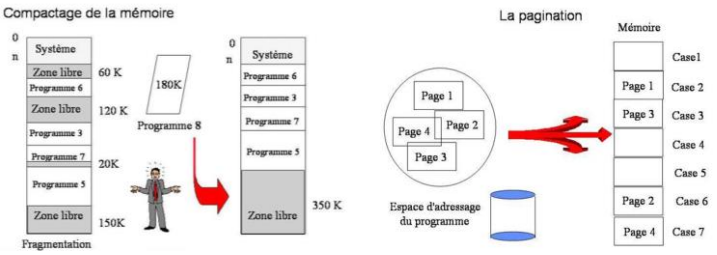
Stratégie First Fit (prog chargé à 100K) **Stratégie Best Fit (prog chargé à 120K)**

1ere Famille : a. Stratégie First Fit : Dans ce contexte, charger un nouveau programme consiste à trouver une zone libre suffisamment grande pour pouvoir y placer le programme. Une première stratégie pour trouver et choisir cette zone libre est de prendre la première zone libre suffisamment grande trouvée au cours du parcours de la mémoire . **b.Stratégie Best Fit :** Une seconde stratégie pour trouver et choisir cette zone libre est de prendre la zone libre dont la taille est la plus proche de celle du programme à allouer, donc celle engendrant le plus petit trou résiduel : c'est la stratégie Best Fit.

Problème : Au fur et à mesure des opérations d'allocations et de désallocations, la mémoire centrale devient composée d'un ensemble de zones occupées et de zones libres éparpillées dans toute l'étendue de la mémoire centrale. Ces zones libres peuvent devenir trop petites pour permettre l'allocation de nouveaux programmes (problème de fragmentation de la mémoire).

Solution : Fragmentation et compactage de la mémoire centrale : la mémoire centrale comporte 4 zones libres mais aucune d'elles n'est assez grande pour contenir un programme 8 de 180K.

Pourtant l'ensemble des 3 zones libres forme un espace de 120 + 20 + 150+ 60 = 350K suffisant pour le programme 8. Pour permettre l'allocation du programme 8, il faut donc réunir l'ensemble des zones libres pour ne former plus qu'une zone libre suffisante : c'est l'opération de compactage de la mémoire centrale **.**



L'allocation en mémoire centrale d'un seul tenant **(1ère famille)** souffre donc de 2 défauts : - Elle nécessite une opération de compactage de la mémoire qui est une opération très coûteuse

- Elle exige d'allouer le programme en une zone d'un seul tenant.

- Une solution est de diviser le programme en portions de taille fixe et égales à l'unité d'allocation de la mémoire centrale.

2ème famille : a.Pagination : mécanisme d'allocation que le programme est découpé en pages. - l'espace d'adressage du programme est découpé **en morceaux de même de taille** : **la page**. - L'espace de la mémoire physique est lui-même découpé en morceaux de même taille : la case. - La taille d'une case est égale à la taille d'une page. Dans ce contexte, charger un programme en mémoire centrale consiste à placer les pages dans n'importe quelle case disponible.

Traduction de l'adresse paginée vers l'adresse physique : effectuée **par la MMU (Memory Management Unit)** qui est chargée de faire cette conversion. Il faut donc savoir pour toute page, dans quelle case de la mémoire centrale celle-ci a été placée : cette correspondance s'effectue grâce à une structure particulière appelée la table de pages. Dans une première approche, la table des pages est une table contenant autant d'entrées que de pages dans l'espace d'adressage d'un processus. Chaque processus a sa propre table des pages. Chaque entrée de table st couple < n°de page, n°de case physique dans laquelle la page est chargée >.

Le processus a **4 pages** dans son espace d'adressage, donc la table des pages a 4 entrées.Chaque entrée établit l'équivalence n°de page, n°de case relativement au schéma de la mémoire centrale.

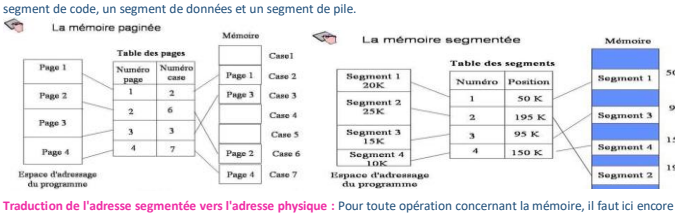
Puisque **chaque processus dispose de sa propre table des pages**, chaque **opération de commutation de contexte** (passage d'un programme à un autre) se traduit également par un **changement de table des pages**, de manière à ce que la "table active" corresponde à celle du **processus élu**.

b.Segmentation : Pour le programmeur, un programme est généralement constitué des données manipulées par ce programme, d'un programme principal, de procédures séparées et d'une pile d'exécution.

La pagination constitue un découpage de l'espace d'adressage du processus qui ne correspond pas à l'image que le programmeur a de son programme.

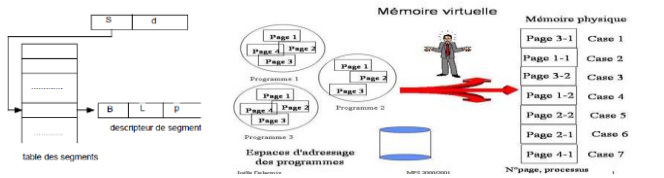
La segmentation est un découpage de l'espace d'adressage qui cherche à conserver cette vue du programmeur. Ainsi, lors de la compilation, le compilateur associe un segment à chaque morceau

du programme compilé. **Difference** : Un segment est un ensemble d'emplacements mémoire consécutifs non sécable. A la différence des pages, les segments d'un même espace d'adressage peuvent être de taille différente. En général, on trouvera un segment de code, un segment de données et un segment de pile.



Traduction de l'adresse segmentée vers l'adresse physique : Pour toute opération concernant la mémoire, il faut ici encore convertir l'adresse segmentée générée au niveau du processeur en une adresse physique équivalente. C'est la **MMU (Memory Management Unit)** qui est chargée de faire cette conversion. Il faut donc connaitre pour tout segment, l'adresse d'implantation dans la mémoire centrale du segment : cette correspondance s'effectue grâce à une structure particulière appelée la **table des segments** .

Table des segments : table contenant autant d'entrées que de segments dans l'espace d'adressage d'un processus. Chaque entrée de la table est un couple < n°de segment, adresse d'implantation du segment >.



B : adresse de base (adresse physique de début du segment) **L :** longueur du segment **p :** protection du segm

Problème : On peut remarquer qu'une fois les programmes 1 et 2 chargés dans la mémoire, toutes les cases sont occupées : le programme 3 ne peut pas être chargé.

Solution : La technique du recouvrement (swapping) permet de stocker temporairement sur **disque** des **images de processus** afin de libérer de la MC pour d'autres processus. **Le va et vient :** ou *swap, se comporte exactement comme la mémoire vive, à la* différence près qu'on ne peut y exécuter des processus (pour exécuter un processus sur le *swap, il faut* le charger en mémoire vive). Le swap contient les processus inactifs.(en attente attandant leurs tours pour s'executer) .

Gst Processus : **Un processus** est une instance d'un programme en train de s'exécuter.

Il est représenté au niveau du système d'exploitation par: **Pile d'exécution , Son code, Ses données,**

Etat du processus… Avec pile d'exécution sert à garder la trace de l'endroit où chaque fonction doit retourner à la fin de son exécution. Un processus est connu du système par une structure de données appelée Descripteur de Processus(DdP).

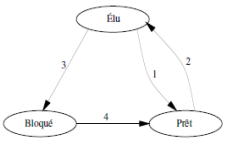
Le rôle d'un descripteur est de permettre la manipulation simultanée de plusieurs processus par le système. Il mémorise : numéro du processus , priorité , état , contexte (copie des registres du processeur…) , pointeurs sur diverses tables (pagination mémoire par ex.) pointeurs sur le code et les données du processus concerné .

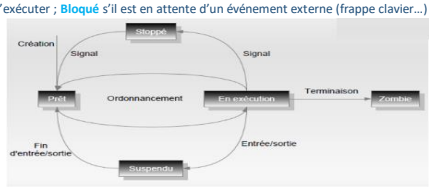
Relations entre processus : Un processus est créé par d'autres processus (sauf le premier). Dans le cas de système à temps partagé, tous les processus progressent dans le temps mais un seul s'exécute à la fois.

Exécution d'un processus : La vitesse d'exécution d'un processus donné ne serait pas nécessairement la même si l'ensemble des processus est exécuté à nouveau, car l'état du système au lancement ainsi que le traitement d'événements peut conduire à des ordonnancements très différents.

Etat d'un processus en détails : Elu s'il est en cours d'exécution sur le processeur.

N.B: Dans le cas d'une machine multiprocesseurs, plusieurs processus peuvent être élus en même temps, mais il y a toujours au plus autant de processus élus que de processeurs. **Prêt** même s'il est suspendu en faveur d'un autre. Un processus est prêt s'il ne lui manque que la ressource processeur pour s'exécuter ; **Bloqué** s'il est en attente d'un événement externe (frappe clavier…)



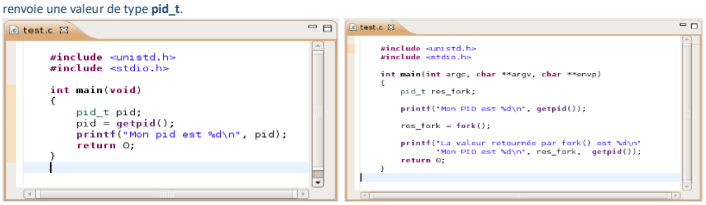


1 Le processus a épuisé le laps de temps attribué. L'ordonnanceur, lorsque le temps est écoulé, élit un autre processus parmi les processus prêts**. 2** L'ordonnanceur élit ce processus parmi les processus prêts**. 3** Le processus s'endort en attente d'un événement externe (décompte d'horloge, attente de données,…) L'ordonnanceur, appelé de façon explicite par le processus courant lorsque celui-ci ne peut progresser dans le traitement d'un appel système, élit un autre processus parmi les processus prêts.

4 L'événement attendu par le processus se produit. C'est leSE qui gère son traitement (et pas le processus, puisqu'il est bloqué) de façon asynchrone, en interrompant le déroulement du processus actuellement élu pour traiter les données reçues, et faire passer le processus en attente de l'état *bloqué* à l'état *prêt*.

Etats de l'état bloqué en détails : Zombie: Le processus a terminé son exécution mais il est toujours référencé dans le système. **Suspendu** : Le processus est temporairement en attente d'une ressource, par exemple il attend la fin d'une entrée / sortie. **Stoppé**: Le processus a été suspendu par une intervention extérieure, son exécution est complètement interrompue et ses ressources sont libérée.

Identification par le PID : Le premier processus du système, **init**, ainsi que quelques autres sont créés directement par le noyau au démarrage. Pour créer un nouveau processus il faut appeler l'appel-système **fork(),** qui va **dupliquer** le processus appelant. Chaque processus a son **PID unique** .Pour connaître son propre PID, on utilise l'appel-système **getpid() , qui ne** prend pas d'argument et renvoie une valeur de type **pid_t**.



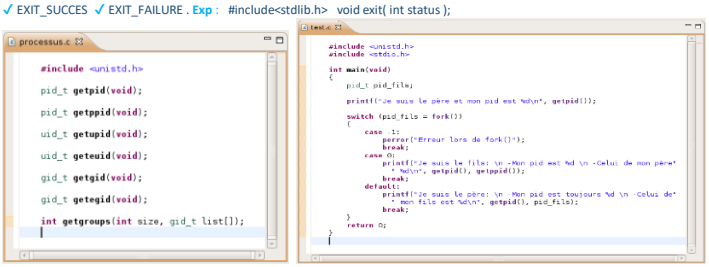
Lors de l'appel de fork, le processus courant est **dupliqué**. Au retour de fork, le père et le fils exécutent le même code. Quand c'est le père qui s'exécute **fork()** retourne 0 quand c'est le fils elle retourne le PID fils.

getpid : retourne l'identificateur unique du processus courant. **getppid** : retourne l'identificateur unique du père du processus courant. **getuid**: retourne l'identificateur d'utilisateur réel du processus courant.

geteuid : retourne l'identificateur d'utilisateur effectif du processus courant. **getgid** : retourne l'identificateur de groupe réel du processus courant.**getegid** : retourne l'identificateur de groupe effectif du processus courant. **getgroups** : retourne le nombre total de groupes associés au processus courant.

En cas d'erreur : En cas d'échec fork retourne la valeur -1 et le code d'erreur est stocké dans la variable errno définie dans <errno.h>. **EAGAIN** : Le nombre maximal de processus a été atteint pour l'utilisateur ou le système. **ENOMEM** : Le noyau ne dispose pas d'assez de mémoire pour créer un nouveau processus.

Terminaison du processus : Une façon de terminer un programme normalement est d'utiliser la fonction exit(). Les marcos : ✓ EXIT_SUCCES ✓ EXIT_FAILURE . **Exp :** #include<stdlib.h> void exit(int status);



Attente de la fin d'un fils : ✓ La fonction *wait() permet à un processus de se* mettre en attente jusqu'à la terminaison d'un de ses fils et de récupérer les informations sur l'état de terminaison de celui-ci. **Exp :**

#include<sys/wait.h> pid_t wait(int *stat_loc) ;

Un processus est suspendu jusqu'à la terminaison d'un de ses fils : Si ce dernier **n'a pas de fils,** wait() **retourne -1. S'il existe un/des fils à l'état zombie**, wait() retourne directement le *pid d'un de ces fils.*

Lorsqu'un fils se termine, le processus père est réveillé et la fonction retourne le *pid de ce fils.*

✓ La fonction *waitpid() permet de se mettre en* attente de la fin du fils dont le PID est indiqué par le paramètre *pid.* Exp :
#include<sys/wait.h> pid_t waitpid (pid_t pid, int *status , int options) ;
-Si la valeur pour le paramètre *pid est -1, waitpid() se comporte* comme *wait().*
-Si *pid est positif, il spécifie le numéro du processus fils à* attendre.
-Si le *pid est nul, il spécifie tout les processus fils dont le numéro* de groupe de processus est égal à celui du processus appelant.
-Si *pid est inférieur à -1, il spécifie tout processus fils dont le* numéro de groupe de processus est égal à la valeur absolue de *pid.*

L'état du processus fils est retourné dans la variable dont l'adresse est passée dans le paramètre ***status.***

Il faut utiliser les macros suivantes pour analyser les circonstances de la fin du fils :

✓ **WIFEXITED(status) est vraie si le processus s'est terminé de son propre chef en invoquant** exit() ou en revenant de main(). On peut obtenir le code de retour du processus fils, c'est-à- dire la valeur transmise à exit(), en appelant **WEXITSTATUS(status).**

✓ **WIFSIGNALED(status) indique que le fils s'est terminé à cause d'un signal**. Le numéro du signal ayant tué le processus fils est disponible en utilisant la macro **WTERMSIG(status).**

✓ **WIFSTOPPED(status) indique que le fils est stoppé temporairement. Le numéro du signal** ayant stoppé le processus fils est accessible en utilisant **WSTOPSIG(status).**

```c
int main(int argc, char **argv, char **envp) {
    pid_t pid_fils, res;
    int info;
    printf("Je suis le père et mon PID est %d\n", getpid());
    switch (pid_fils = fork()) {
        case -1:
            perror("Erreur lors du fork");
            exit(EXIT_FAILURE);
        case 0:
            printf("Je suis le fils, mon PID est %d et celui de mon père %d\n", getpid(), getppid());
            sleep(2); exit(2);
        default:
            if ((res = waitpid(pid_fils, &info, 0)) == -1 ) {
                perror("Erreur lors de waitpid");
                exit(EXIT_FAILURE);
            }
            else {
                printf("Je suis le père, le PID de mon fils était %d\n",res);
                if ( WIFEXITED(info) )
                    printf("Mon fils a renvoyé la valeur %d\n", WEXITSTATUS(info));
                else
                    printf("Mon fils s'est arrêté à cause d'un signal\n");
            }
    }
    return EXIT_SUCCESS;
}
```

Chap : Ordonnencement : La fct d'ordonnancement gère le partage du processeur entre les différents processus en attente pour s'exécuter, c-a-d entre les différents processus qui sont dans l'état **prêt**.

- **Le système d'ordonnancement** gère: une file des processus **prêts** et une file des processus **bloqués**.
- La file des processus prêts est classée selon une politique d'ordonnancement qui assure que le processus en tête de file est le prochain processus à élire au regard de la politique mise en œuvre.
- C'est le **répartiteur** qui élit le processus en tête de file, c'est-à-dire qui lui alloue un processeur libre. Une fois élu par le répartiteur, le processus s'exécute. ✓ S'il quitte le processeur, sur préemption, il réintègre la file des processus prêts . ✓ S'il quitte le processeur sur un blocage, il intègre la file des processus bloqués.
- **L'ordonnanceur** est un programme système dont le rôle est d'allouer le processeur à un processus prêt.
- **Algorithmes d'ordonnancement : FCFS *(First Come First Serve)*, premier arrivé premier servi :** le Premier Arrivé est le Premier Servi PAPS ou le plus court d'abord (SJF Short Job First). Il lui alloue le processeur jusqu'à ce qu'il se termine ou il se bloque (en attente d'un événement). *Il n'y a pas de réquisition.*
- **SJF *(Short Job First)*, plus court d'abord :** Si l'ordonnanceur fonctionne selon la stratégie SJF, il choisit parmi le lot de processus à exécuter, le plus court d'abord (plus petit temps d'exécution).
- **SRT *(Shortest Remaining Time)*, plus petit temps de séjour :** 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.
- **RR *(Round Robin)*, algorithme circulaire :** L'algorithme du **tourniquet, circulaire** ou **round robin**
- ✓ Il mémorise dans une file du type FIFO (First In First Out) la liste des processus prêts, c'est à dire, en attente d'exécution. ✓ Il alloue le processeur au processus en tête de file, pendant un quantum de temps.
- ✓ 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 ne se termine pas au bout de son quantum, son exécution est suspendue. ✓ Le processeur est alloué à un autre processus (celui en tête de file).
- Le processus suspendu est inséré en queue de file. Les processus qui arrivent ou qui passent de l'état bloqué à l'état prêt sont insérés en queue de file.
- **PRI *(Priorités, sans évolution)* :** L'ordonnanceur à priorité attribue à chaque processus une priorité. Le choix du processus à élire dépend des priorités des processus prêts. Les processus de même priorité sont regroupés dans une file du type FIFO. Il y a autant de files qu'il y a de niveaux de priorité. L'ordonnanceur choisit le processus le plus prioritaire qui se trouve en tête de file. En général, les processus de même priorité sont ordonnancés selon l algorithme du tourniquet.
- **Pour construire un ordonnanceur**, il suffit de disposer : ✓ **d'un mécanisme d'interruption** ✓ **d'un mécanisme de commutation de contexte** ✓ **et d'une horloge.** ➔ On choisit un **intervalle de temps** (quantum) et on génère une **interruption à** la fin de chaque quantum. Cette interruption provoque une commutation de contexte qui redonne le contrôle à l'ordonnanceur.
- **Mécanisme d'interruption : L'interruption**: ✓est un signal ✓est la conséquence d'un évènement qui peut être interne au processus et résultant de son exécution, ou bien extérieur et indépendant de cette exécution.
- Dans un système multitâche, interruptions: ✓causées par les périphériques (fin d'exécution de requête)
- ✓ causées par signaux d'horloge ✓évènements extérieurs ✓ déroutement en cas d'erreur (accès illégal à la mémoire, division par zéro ...) ✓ interruptions provoquées par instruction spéciale
- **Exemple : interruption disque** : interruption venant d'un contrôleur de disques : ✓mettre le processus actif à l'état prêt ✓déterminer la cause de l'interruption (p. ex : fin de lecture) ✓trouver le processus demandeur (qui est bloqué) ✓ lui transférer les données reçues ✓le remettre à l'état prêt ✓activer un des processus prêts
- **Exemple : interruption horloge :** quantum de temps épuisé (ordonnancement avec réquisition, preemptive scheduling) : ✓remettre le processus actif à l'état prêt ✓activer un des processus prêts
- **Priorités des niveaux d'interruption :** S'il existe plusieurs niveaux d'interruption, on peut avoir à un moment donné, plusieurs indicateurs d'interruption positionnés en même temps.
- D'où l'utilisation de priorités sur les niveaux d'interruption.
- Si dans une interruption, une seconde se produit, ..., les processus ne progressent plus ; on fait des commutations de mots d'état en cascade. D'où le masquage et le désarmement de niveaux d'interruption.
- En masquant un niveau d'interruption, on retarde la prise en compte des interruptions de ce niveau.
- En désarmant un niveau d'interruption on annule l'effet des interruptions correspondantes. On peut réarmer un niveau désarmé.
- **Chap : communication inter-processus :** La solution à la résolution des problèmes d'accès concurrents consiste à interdire la modification de données partagées à plus d'un processus à la fois, c'est-à-dire définir un mécanisme d'**exclusion mutuelle** sur des portions spécifique du code, appelés **sections critiques**
- **Section critique :** On appelle **section critique la partie d'un programme où se produit le conflit** d'accès
- **Exclusion mutuelle : mécanisme qui empêche les autres processus d'accéder à des** données partagées si celles-ci sont en train d'être utilisées par un processus .
- On peut formaliser le comportement des sections critiques au moyen de quatre conditions suivantes afin d'éviter le conflit d'accès et garantir le bon fonctionnement du SE: ✓ Deux processus ne peuvent être simultanément dans la même section critique
  ✓ Aucune hypothèse n'est faite sur les vitesses relatives des processus, ni sur le nombre de processeurs
  ✓ Aucun processus suspendu en dehors d'une section critique ne peut bloquer les autres
  ✓ Aucun processus ne doit attendre trop longtemps avant d'entrer en section critique
- **mécanismes de l'exclusion mutuelle : Masquage des interruptions :** Masquer les interruptions avant d'entrer dans une section critique et de les restaurer à la sortie. ✓Si le masquage se fait par **le processus entrant** à la section critique donc il ne pourra pas être suspendu au profit d'un autre processus, puisque l'inhibition (le masquage) des interruptions empêche l'ordonnanceur de s'exécuter. ✓Si le SE fait le masquage, cette méthode ne sera pas efficace pour les systèmes multi-processeurs; puisque les processus s'exécutant sur les autres processeurs peuvent toujours entrer en section critique **Inconvénient:** pour le cas du masquage des interruptions. Lorsqu'un processus entre en section critique, tous les autres processus sont bloqués, même si la section critique ne concerne qu'un seul processus. **Solution ?**
- **Variables de verrouillage :** ➔ Pour cela, on déclare une variable par section critiques, qui joue le rôle de **verrou**. La variable est mise à 1 par le processus entrant dans la section critique considérée et remise à 0 par le processus lorsqu'il quitte.
  While(verrou==1) /*Attenre*/ Verrou=1; Section_critique(); Verrou=0;
- **Variable d'alternance :** Le problème dans l'algorithme précédent est dans le cas où deux processus modifient la même variable en même temps.➔ Pour éviter cela on peut mettre en œuvre une variable tour, qui définit quel processus a le droit d'entrer en section critique. Mais, cette méthode ne respecte pas les deuxième et troisième règles des sections critiques.
- **Variables d'alternances de Peterson :** La solution proposée par Peterson améliore l'approche de variable d'alternances en permettant au processus dont ce n'est pas le tour de rentrer quand-même en section critique, lorsque l'autre processus n'est pas encore prêt à le faire
- **Problèmes classiques de synchronisation : Les producteurs / consommateurs :** Des activités productrices produisent des informations consommées par des activités consommatrices.
- L'exécution concourante des producteurs et des consommateurs repose sur l'utilisation d'un ensemble de N *buffers intermédiaires remplis par* les producteurs et vidés par les consommateurs.
- **Exp** : Le processus clavier produit des caractères qui sont consommés par le processus d'affichage à l'écran.
- **Les lecteurs / rédacteurs** Un objet est partagé entre plusieurs activités concurrentes. Certaines activités (les lecteurs) ne modifient pas le contenu de l'objet contrairement à d'autres (les écrivains).
- Les lecteurs peuvent donc accéder simultanément au fichier. Un écrivain au contraire doit accéder seul au fichier. Si le fichier est disponible, lecteur et rédacteur ont la même priorité.
- Supposons qu'une base de données ait des lecteurs et des rédacteurs, et qu'il faille programmer les lecteurs et les rédacteurs de cette base de données.
- Les contraintes sont les suivantes : plusieurs lecteurs doivent pouvoir lire la base de données en même temps ; si un rédacteur est en train de modifier la base de données, aucun autre utilisateur (ni rédacteur, ni même lecteur) ne doit pouvoir y accéder.
- **Les primitives sleep et wakeup :** sont deux appels système. La primitive **sleep** endort le processus qui l'appelle, en rendant la main à l'ordonnanceur alors que **wakeup** permet à un processus de réveiller un processus endormi dont l'identifiant lui est passé en paramètre .Ces appels système peuvent servir à mettre en œuvre des mécanismes de type **producteur-consommateur**
- **Sémaphore :** Un sémaphore S est une variable entière qui n'est accessible qu'au travers de 3 opérations Init, P et V. La valeur d'un semaphore est le nombre d'unités de ressource (exemple : imprimantes...) libres. S'il n'y a qu'une ressource, un semaphore binaire avec les valeurs 0 ou 1 est utilisé.

- **Init(S, valeur)** est seulement utilise pour initialiser le sémaphore S avec une valeur. Cette opération ne doit être réalisée qu'une seule et unique fois.
- L'opération **P (Puis-je?)**est en attente jusqu'à ce qu'une ressource soit disponible, ressource qui sera immédiatement allouée au processus courant. **V (Vas-y)** est l'opération inverse; elle rend une ressource disponible à nouveau après que le processus a terminé de l'utiliser.
- **Chap : Signaux : Signal** : un mécanisme de communication entre le noyau et les processus ou entre processus. **Un signal** est représenté dans le système par un nom de la forme ***SIGXXX***.
- Il existe un nombre déterminé de signaux et divisés en deux catégories, **les signaux classiques** ou standars leurs numéros definis de **1 à 31** et **les signaux temps-réel** définies de ***SIGRTMIN* : 32** à ***SIGRTMAX* : 64**
- ***NSIG*** **:** Le nombre total de signaux existants dans le système est indiqué par la constante *définie dans le fichier d'en-tête <signal.h>.*
- **Les signaux classiques : Exps :** ***SIGTERM*** *: Terminaison du processus .* ***SIGKILL*** *: Terminaison irrévocable.*
- ***SIGINT*** *: Terminaison à partir du clavier (généralement ctrl-c) .* ***SIGQUIT*** *: Terminaison clavier (ctrl-\ )*
- ***SIGSTOP*** *: Arrêt temporaire du processus (passage à l'état stoppé).*
- ***SIGSTP*** *: Arrêt temporaire du processus à partir du clavier (ctrl-z).*
- ***SIGCONT*** *: Reprise de l'exécution.* ***SIGSEGV*** *: Référence mémoire invalide.* ***SIGCHLD*** *: Fils stoppé ou terminé.*
- **Les signaux temps-réel :** Ils sont réservés à l'utilisateur, ce sont des extensions de ***SIGUSR1*** *et* ***SIGUSR2.***
  Ils ne sont pas représentés par des constantes, mais directement par leurs valeurs, qui s'entendent de *SIGRTMIN et SIGRTMAX.*
- **Classiques VS Temps-réel :** Quand plusieurs **signaux classiques** d'un même type sont envoyés à un processus sans qu'il puisse les prendre en compte le noyau ne conserve qu'un exemplaire des signaux envoyés et ne délivrera donc qu'un seul signal au processus Si plusieurs **signaux temps-réel** sont en attente d'être délivrés à un processus, lorsque le processus pourra les traiter, ils lui seront délivrés en ordre croissant du numéro de signal. Il y a donc une **priorité** pour **les signaux temps-réel** qui n'existe pas pour les signaux classiques.
- **Affichage des signaux :** Pour obtenir le libellé d'un signal dont le numéro est connu, il existe plusieurs méthodes : Il existe un tableau contenant la chaîne de caractère à chaque signal. Il est défini dans le fichier *<signal.h> :* ***char *sys_siglist[num_signal]*** les deux fonctions de la bibliothèque C : ***strsignal()*** *et* ***psignal().***
- #include<signal.h> #include<string.h> **char* strsignal (int sig) ;** **void psignal (int sig ,const char* s) ;**
- **L'appel système *kill()*** *est utilisé pour envoyer un* signal à un processus ou à un groupe de processus.
- ***NB****: L'appel système* ***kill()*** *est particulièrement mal nommé car il ne tue que rarement l'application cible.*
- #include<signal.h> **int kill (pid_t pid , int sig) ;**
- Le premier argument représente l'identité du destinataire. Le deuxième indique le signal à émettre.
- Si ***pid est positif****, le signal sig est envoyé au processus identifié par pid.* Si ***pid est nul****, le signal sig est envoyé aux processus du groupe du* processus émetteur. Si ***pid est égal à -1****, le signal sig est envoyé à tous les processus, sauf le* premier (processus *init).*
- Si ***pid est inférieur à -1****, le signal sig est envoyé au groupe du* processus identifié par la valeur *-pid.*
- Si le signal envoyé est nul, l'appel ne génère pas de signal mais contrôle l'existence du processus destinataire.
- Si l'appel se déroule correctement, la valeur 0 est renvoyée, sinon, le valeur -1 est renvoyée si une erreur s'est produite et la variable *errno* contient l'une des valeurs suivantes :
- ***EINVAL*** *: Le numéro du signal est invalide.* ***ESRCH*** *: Le processus ou groupe de processus cible n'existe* pas.
- ***EPERM*** *: Le processus appelant n'a pas la permission d'envoyer* un signal au processus destinataire car ils n'appartiennent pas au même propriétaire.
- Un processus appartenant au super utilisateur a bien sûr le droit d'envoyer un signal à tous les autres processus (sauf *init afin de* protéger le système, aucun processus ne peut envoyer de signal qui provoquerait la terminaison du processus ***init.***
- **raise() :** Il existe aussi une fonction ANSI-C permettant à un processus de s'envoyer un signal Cette fonction est équivalente à *kill(getpid(), sig). En* cas d'erreur, une valeur non nulle est renvoyée. #include<signal.h> **int raise (int sig) ;**
- **L'appel système *killpg()*** *est utilisé pour envoyer un* signal à un groupe de processus. Le premier argument de l'appel représente le groupe vers lequel le signal est émis. Si sa valeur est nulle, alors le signal est envoyé aux processus du groupe du processus émetteur. Sinon, le fonctionnement est similaire à celui de *kill().* #include<signal.h> **int killpg (int pgrp , int sig) ;**

```c
int main()
{
  pid_t pid_fils;
  int info_fin;

  switch ( pid_fils = (long)fork() ){
      case -1:
        perror("Erreur lors de fork");
        return EXIT_FAILURE;
      case 0:
        while ( 1 ){
          printf("Le processus fils est vivant\n"); sleep(1);
        }
      default:
        sleep(5);
        printf("Le processus père envoie le signal SIGKILL à son fils\n");
        if ( kill(pid_fils, SIGKILL) ){
          perror("Erreur lors de kill");
          return EXIT_FAILURE;
        }
        if ( waitpid(pid_fils, &info_fin, 0) != -1 )
          if ( WIFSIGNALED(info_fin) )
              printf("Le processus s'est terminé à cause du signal %s\n", sys_siglist[WTERMSIG(info_fin)]);
  }
  return EXIT_SUCCESS;
}
```

- **Réception d'un signal : L'appel système *signal()*** permet d'installer un gestionnaire de signaux en une seule instruction.
  #include<signal.h> **void (*signal (int signum, void (*handler) (int))) (int);**
- Le second argument est un pointeur sur la fonction de déroutement (gestionnaire) ou bien l'une des deux constantes suivantes :
- ***SIG_ING*** *: Pour ignorer le signal.* ***SIG_DFL*** *: Pour repositionner le comportement par défaut.*
- La fonction de déroutement prend en paramètre un entier correspond au signal qui l'a activée et ne renvoie aucun argument.
- Lors de l'arrivée d'un signal, le système interrompt l'exécution normal du processus pour exécuter la fonction de déroutement. Il passe la valeur du signal émis à cette fonction.
- ***pause()*** *:* Un processus peut se mettre en attente de l'arrivée d'un signal grâce à l'appel système Cet appel suspend le processus appelant jusqu'a l'arrivée d'un signal quelconque.La valeur de retour de l'appel est toujours -1, et la variable *errno contient la valeur EINTR.* **EINTR :** *L'appel système a été interrompu par la réception d'un* signal. #include < unistd.h > int pause (void)
- **Les groupes de signaux :**

```c
void deroute(int sig);

int main() {
    sigset_t masque, pendante;
    if(signal(SIGINT, deroute) == SIG_ERR) {
        perror("signal SIGINT non déroutable"); exit(1);
    }
    sigemptyset(&masque);
    sigaddset(&masque, SIGINT);
    if(sigprocmask(SIG_BLOCK, &masque, NULL) < 0) {
        perror("SIG_BLOCK dans sigprocmask"); exit(1);
    }
    sleep(5);
    if(sigpending(&pendante) < 0) {
        perror("sigpending"); exit(1);
    }
    if(sigismember(&pendante, SIGINT))
        printf("Signal SIGINT pendant \n");
    if(sigprocmask(SIG_UNBLOCK, &masque, NULL) < 0) {
        perror("SIG_UNBLOCK dans sigprocmask"); exit(1);
    }
    printf("Signal SIGINT débloqué\n");
    return 0;
}

void deroute(int sig) {
    printf("Signal recu : %s\n", strsignal(sig));
}
```

```c
#include <signal.h>
int sigemptyset(sigset_t * set);
int sigfillset(sigset_t * set);
int sigaddset(sigset_t * set, const int signum);
int sigdelset(sigset_t * set, const int signum);
int sigismember(const sigset_t * set, const int signum);
```

- ✓La première crée un ensemble vide. ✓ La seconde crée un ensemble qui contient tous les signaux.
- ✓ Les deux fonctions suivantes ajoutent un signal à un ensemble ou réalisent l'opération inverse.
- ✓ Le retrait permet de tester l'appartenance d'un signal à un ensemble, il renvoi 1 si le signal appartient à l'ensemble, et 0 sinon.
- ✓ Le seul cas d'erreur possible pour l'ensemble de ces fonctions consiste à fournir un signal invalide. La valeur de retour est alors -1 et la variable *errno contient la valeur EINVAL.* ✓*Les objets sigset_t défini dans <signal.h> représentent des ensembles de signaux.*
- **Le blocage des signaux :** Chaque processus possède un ensemble nommé masque de signaux, contenant la liste des signaux bloqués. int sigprocmask (**int** *how*, **const sigset_t** **set*, **sigset_t** **oldset*);
- Le 1er argument indique la façon dont l'ensemble de signaux passé en second argument doit être traité. Il prend l'une des valeurs suivantes : ***SIG_BLOCK*** *: Les signaux bloqués sont l'union de l'ensemble courant et de* l'ensemble *set.* ***SIG_UNBLOCK*** *: Les signaux de l'ensemble set sont retirés de la liste des* signaux bloqués. ***SIG_SETMASK*** *: Les signaux bloqués sont ceux de l'ensemble set.*
- Le 3eme argument de l'appel contient l'ensemble des signaux défini avant l'appel à *sigprocmask().*
- La valeur de retour de l'appel est 0 en cas de succès, et -1 sinon. Dans ce cas, la variable *errno prend l'une des valeurs suivantes :*
- ***EINVAL*** *: valeur de how est incorrect.* ***EFAULT*** *: set ou oldset contient une adresse invalide.* ***EINTR*** *: L'appel système a été interrompu.*
  Le blocage des signaux : ***sigpending*** **:** *Le processus peut demander au système la liste des signaux en attente par l'appel système sigpending().* #include<signal.h> *int sigpending(sigset_t *set);* Le paramètre *set contient à l'issue de l'appel, la liste* des signaux pendants du processus appelant. Cet appel retourne 0 quand l'appel réussit et -1 en cas d'échec. Le seul cas d'erreur possible pour cet appel est celui ou l'adresse du paramètre *set est invalide. La variable* errno contient alors la valeur *EFAULT.*