NOTES SE CM

CHAPITRE 1 : GENERALITES

- Généralités sur les processus

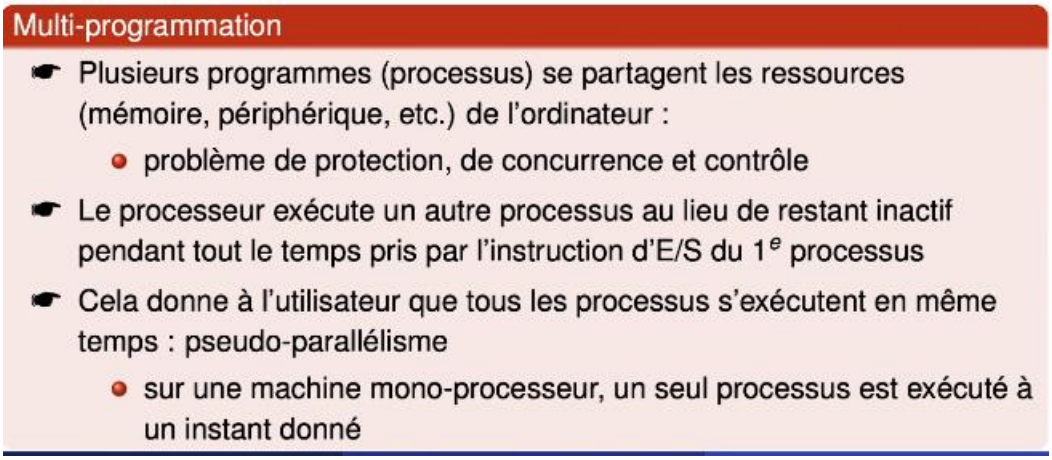
1- Partage de l'unité centrale

NB : Un processus est une **instance d'exécution d'un programme** lancé par le système ou par l'utilisateur.

L'UC est composé de 2 unités principales : le **processeur** (CPU) accompagné de la **mémoire principale** (RAM par exemple)**.** Souvent la mémoire principale est associée à une **mémoire secondaire** (stockage des données de manière permanente). A cela s'ajoutent les dispositifs d'entrée/sortie.

Les systèmes sont conçus selon 2 approches.

- La **monoprogrammation** : elle consiste au fait que dans le système, il n'y a qu'**un seul programme** transformé en processus dans la mémoire centrale, il aura donc toute la mémoire centrale comme espace pour l'exécution du processus. Un souci est que le processus monopolisera le CPU et donc les autres activités du système sont suspendues, un autre étant que si le processus est dépendant d'un autre, il se posera le problème de la lenteur d'exécution et de travail excessif du CPU. D'où la seconde approche.
- La **multiprogrammation** : elle consiste au fait que plusieurs programmes peuvent être transformés en processus **en même temps**. L'exécution se fera selon l'algorithme d'ordonnancement du système (cf. chapitre suivant). Ainsi lorsqu'un processus devra exécuter autre chose (instruction d'entrée/sortie par exemple), il laissera de la place aux autres processus. De plus, si un programme est prioritaire par rapport à un autre en cours d'exécution, le CPU peut suspendre ce dernier et prendre le programme en priorité. Cela soulève néanmoins le problème de la sécurité, les processus ne doivent pas violer par eux-mêmes les espaces mémoires des autres processus.



NB : Entrée de données par la carte réseau : Les données arrivant par la carte réseau arrivent par les **micro-contrôleurs**, et tant que les données n'arrivent pas en intégralité, le CPU ne peut pas les traiter.

2- Processus

Un processus peut être issu d'un programme lancé par l'utilisateur (processus utilisateur), stocké sous disque (mémoire secondaire), il peut aussi être issu du noyau (processus système). **Un processus système est prioritaire par rapport au processus utilisateur. Les priorités des processus utilisateurs entre eux dépendent de la priorité des utilisateurs. Les priorités des processus système entre eux sont fixés par rapport aux algorithmes d'ordonnancement et de priorité.**

- Processus = suite d'instructions. On peut l'exécuter et l'interrompre :
  - Peut se trouver dans plusieurs états (actif, suspendu, terminé, en attente d'un événement)

3- Rôle du noyau dans la gestion de processus

- Création, suppression et interruption
- Ordonnancement des processus : *exécution équitable entre processus tout en privilégiant les processus systèmes*
- Synchronisation des processus :
  - Choisir le processus à exécuter à un instant donné
  - Choisir le moment où interrompre un processus
  - Choisir le processus qu'il faut exécuter ensuite (le suivant)
  - Spécifier les ressources dont à besoin (et qu'il faut affecter à) un processus
- Gestion des conflits d'accès ressources partagées
- Protection des processus d'un utilisateur contre les actions d'un autre processus
- etc.

NB1 : Le système de fichiers est géré par le noyau contrairement **au Shell** qui ne l'est pas**.** Le noyau n'est constitué que de peu de fonctionnalités (E/S en amont, les pilotes en aval, les sous-systèmes entre les deux [gestions de fichiers, gestions de processus] cf. chapitre 2 DUT1).

NB2 : Tout dispositif d'entrée/sortie distinct de la composition principale d'un ordinateur est équipé d'un contrôleur qui permet la synchronisation de l'appareil avec la machine.

4- Cycle de vie d'un processus

Rappelons qu'un processus est créé après lancement d'un programme. Lors de la création du processus, il est à l'état **« prêt »**. Après cela il attend d'être pris par le processeur (ou un cœur). L'entité qui gère les processus est l'**ordonnanceur**. Et dans le cas d'un système multiprogrammé, plusieurs processus peuvent être à l'état « prêt ». Lorsque le processus démarre, il sera à l'état **élu/choisi/en exécution**. Cela peut être dû à 2 possibilités : soit le processeur (ou un cœur) est libre et le processus en question va s'exécuter, soit le processus est prioritaire par rapport aux autres processus.

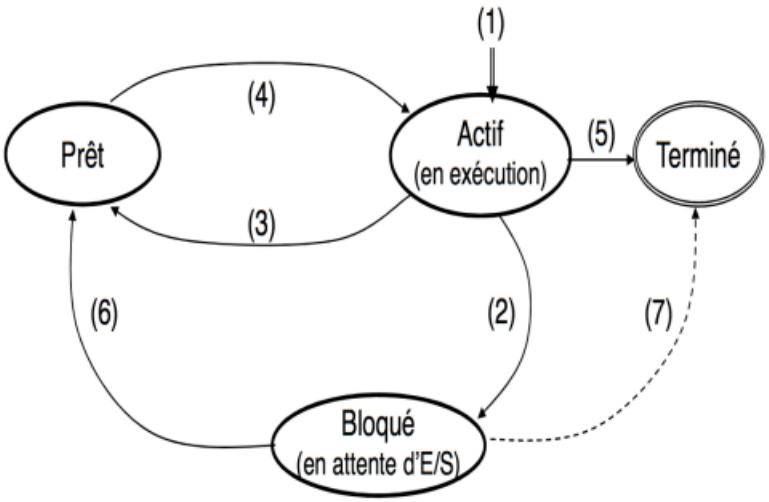
A l'exécution, on a 2 cas de figures :

- Le processus se termine et il est à l'état **« terminé »**
- Le processus est interrompu et dans ce cas il est à l'état « **suspendu »** Cela peut être dû à deux causes :
    - Un processus prioritaire vient prendre sa place
    - La durée accordée à son exécution est dépassée et le processus n'a pas encore atteint l'état « terminé » (principe du quantum alloué)

  Dans ce cas-ci le processus interrompu repasse dans la file d'attente le temps que le processeur (ou un cœur) se libère.

  NB : Le nombre de passages de ce genre de processus est donné par la formule suivante : $t/x + 1$ (t pour le temps nécessaire à son exécution, x pour le temps qui lui est accordé (**quantum alloué**), le « +1 » est appliqué dans le cas où t n'est pas divisible par x).

Un processus peut passer de l'état « élu » à l'état « bloqué » **sans possibilité de retour en arrière, et devra (dans ce cas) impérativement repartir sur la liste d'attente**. Cela est dû au fait que le processus a besoin d'une ressource non disponible. Dans le cas où l'état bloqué est définitif, le processus passera à l'état « terminé »



5- Structures de données utilisée lors de la gestion de processus

Pour la gestion de processus, le noyau utilise le **PCB (Process Control Bloc)** appelé aussi **bloc contexte**, créé lors de chaque création de processus. Chaque processus possède un PCB (à l'équivalence d'un CNI par citoyen). Il contient l'ID du processus, son état, son registre de mémoires, son compteur ordinal (pointeur d'instructions) etc.

La 2e structure utilisée par le noyau est la **table des processus.** C'est une table où chaque ligne contient les informations d'un processus. Elle contient l'ID du processus parent, les fichiers ouverts, l'occupation mémoire etc.

NB : Le pointeur de processus présent sur chaque ligne pointe vers le PCB de ce dernier, évitant ainsi la duplication des données.

NB : La table des processus est stockée dans l'espace mémoire du noyau, donc il s'agit d'une table protégée et non accessible par les processus. De plus, cette table est commune à tous les processus contrairement à un PCB qui est spécifique à chacun.

6- Structuration de l'espace mémoire d'un processus

Chaque processus contient son espace mémoire dans la mémoire centrale (RAM).

Cet espace est divisé en 4 segments (délimité de 0 à n-1 (n étant le nombre d'octets)) :

- Le **segment de code** : Il est toujours au début de l'espace mémoire et contient les instructions à exécuter. C'est un segment en lecture seule et tous les processus enfants lisent les instructions du processeur parent dans ce segment. (Si le programme est plus grand en termes d'espace que le segment de code, le noyau divisera le programme en portions réduites, ce qui explique souvent les lenteurs des processus ou les chargements répétitifs [loading…])
- Le **segment de données** : Il se trouve juste après le segment de code. Il contient les données initialisées et celles qui ne le sont pas.
- Le **Tas** : C'est une structure arborescente utilisée souvent pour les programmes dynamiques. Il est utilisé pour stocker les données produites par le processus en cours d'exécution.
- La **pile** : C'est une structure linéaire qui a 2 fonctions : empiler des fonctions et les dépiler. Elle est placée en fin d'espace mémoire et est **renversée** (pour éviter de déborder de l'espace mémoire alloué au processus). Elle contient les différentes fonctions du programme. Si lors de l'exécution, le programme appelle une fonction, la pile dépile cette dernière pour l'exécuter (si la fonction se trouve en bas de la pile, cette dernière va dépiler toute fonction se trouvant au-dessus de la fonction concernée jusqu'à l'atteindre et rempile les autres fonctions)

NB : L'espace mémoire d'un processus n'est pas une structure linéaire. Il est une chaine reliant les différents segments par les pointeurs d'indexation.

7- Interruption d'un processus :

Interruption

- Dans le cas des transitions « **actif** ⟼ **bloqué** » et « **actif** ⟼ **prêt** » on parle d'**interruption (IT)**

D'où viennent les IT

- Quand le processus atteint une instruction d'E/S
- Le quantum attribué au processus est écoulé
- Un processus plus urgent doit être exécuté
- Un processus nécessite une ressource (matérielle ou logicielle) ou une donnée (un résultat calculé par un autre processus, ou un ensemble d'instructions qui ne sont pas encore chargées en mémoire) détenue par un autre processus (elle n'est pas encore disponible)

Une interruption de processus est toujours causée par un signal d'interruption (IT). Une IT est stockée dans le **gestionnaire d'IT.** A chaque interruption on associe une fonction. Cette fonction permet par exemple de sauvegarder l'état d'un processus, son compteur ordinal et sa progression en attendant son éventuel passage suivant.

On a 2 types d'interruptions :

- Interruption par un évènement interne : causé par le processus lui-même. Il a 2 formes :
  - Un appel système (quand le processus a une fonction d'interruption par exemple comme pause () ou autres)
  - Un déroutement (lorsque le processus fait quelque chose d'imprévu) comme :
    - Une violation mémoire (le processus qui déborde)
    - Une instruction non prévue ou non autorisée

    Dans ces 2 cas, le noyau tue le processus.
- Interruption par un évènement externe causée par l'utilisateur ou par un autre processus.

NB : Interruptions matérielle (IRQ) et logicielle

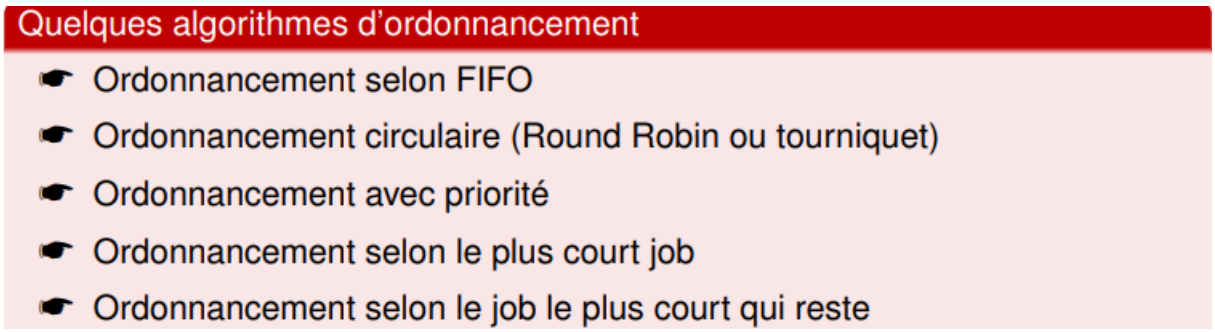
Traitement d'une IT

- Deux sortes d'interruptions : Matérielles et Logicielles
  - **IT Matérielles (IRQ)** : générées par les périphériques. Parviennent au processeur par l'intermédiaire d'un contrôleur d'IT
  - **IT Logicielles** : des IT internes, c'est le processus qui appelle cette IT à l'aide du numéro d'IT.
    - : Par exemple, pour appeler une IT DOS, appeler l'IT N°21H
  - Si plusieurs interruptions arrivent au même temps, alors celle qui a le plus petit numéro qui a la plus grande priorité :
    - Exemple : IRQ horloge système = 0, IRQ port parallèle = 7

CHAPITRE 2 : ORDONNANCEMENT DE PROCESSUS

L'ordonnancement concerne la gestion de processus en fonction des priorités des temps et des critères en fonction des circonstances du système. Il concerne principalement le classement des processus dans la file d'attente. Pour se faire, nous faisons recours à des **algorithmes d'ordonnancement**. Ils sont divisés en 2 familles : les **préemptifs** et les **non préemptifs**.

Un algorithme préemptif, contrairement à celui non préemptif, est un algorithme qui pose des contraintes à l'utilisation du processeur, donc c'est un algorithme qui peut réquisitionner le processeur. La condition de réquisition peut être la contrainte de temps (quantum alloué), les priorités des utilisateurs ou processus etc.



L'ordonnancement est implanté pour améliorer la productivité, et donc le processeur devra travailler à chaque fois qu'un processus est à l'état « prêt ». Une autre raison est l'équité. Le processeur ne gère pas que les processus, il gère également les autres ressources de la machine, ce qui fait que l'ordonnancement peut également suspendre des processus pour gérer les autres ressources de la machine.

Critères d'ordonnancement :

- Le nombre de processus par unité de temps :
  - Ce critère dépend cependant de la longueur des processus.
- La durée de rotation :
  - Délai moyen entre l'admission du processus et la fin de son exécution.
- Le temps d'attente :
  - Temps moyen qu'un processus passe à attendre
  - Il s'obtient en soustrayant la durée d'exécution du processus de sa durée de rotation
  - Cette durée est indépendante de la durée d'exécution du processus lui-même.
- Le temps de réponse :
  - Vitesse de réaction aux interventions extérieures. Les programmes d'avant-plan doivent pour cela avoir priorité sur les tâches de fond

NB : La durée de rotation est la durée pour laquelle le processus passe de l'état « prêt » à l'état « terminé » : T(sortie) – T(arrivée).

Durée moyenne de processus : $\frac{\sum_1^n Dr(Pi)}{n}$ (Dr : Durée de rotation, n : nombre de processus).

Pour une exécution équitable des processus, on attribue à chacun d'entre eux un poids (tel que la somme des poids est égale à 1) et dans ce cas la durée moyenne de processus est : $\frac{\sum_1^n \propto i \times Dr\ (Pi)}{n}$ avec $\propto i$ le poids du processus Pi.

Le temps d'attente d'un processus est calculé en prenant la durée de rotation du processus où l'on diminue le temps d'exécution (temps de service prévisionnel).

Ce temps équivaut aux temps passés au niveau de la file d'attente et les temps où il y avait des opération d'entrée/sortie (temps de blocage).

- La prévisibilité :
  - Un système qui d'habitude réagit rapidement aux commandes mais qui parfois prend un temps beaucoup plus long sera perçu comme moins stable que s'il répondait à chaque fois dans un temps comparable même s'il est globalement plus lent.
  - Le système semblera aussi plus convivial s'il respecte l'idée parfois fausse que les utilisateurs se font de la complexité des tâches.

ORDONNANCEMENT NON PREEMPTIF :

1- FIFO (First Come (In), First Out)

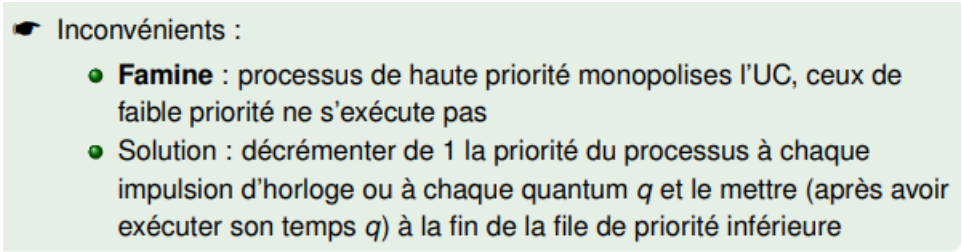
C'est un algorithme où le premier processus qui arrive est le premier à être exécuté. Donc la file d'attente se fait en fonction de l'ordre d'arrivée des processus. De ce fait, le processeur ne perd pas de temps et le principe est simple. Les inconvénients en sont que les processus prioritaires ne seront pas « prioritaires » même s'il s'agit d'un processus urgent, de plus un processus peut monopoliser le processeur à cause de son temps de service qui est long.

2- SJF (Shorted Job First)

C'est un algorithme qui trie les processus par ordre croissant du temps d'exécution. Et lors de l'exécution d'un processus, le processus ne peut pas être retiré du processeur, même si un processus de temps de service plus court que celui en exécution arrive. L'inconvénient de cet algorithme est qu'il n'est pas possible de connaître à l'avance le temps de service d'un processus, à moins qu'il ait déjà été exécuté auparavant.

3- Ordonnancement avec priorité

C'est un algorithme qui trie les processus selon les priorités qu'ils ont. Les processus de plus haute priorité s'exécutent en premier.



ORDONNANCEMENT PREEMPTIF :

1- Round Robin – Algorithme du tourniquet

C'est un algorithme qui trie les processus par ordre d'arrivée. En revanche, si le processus dépasse le quantum de temps alloué par l'ordonnanceur et qu'il n'a pas encore atteint l'état « terminé », le processus est remis dans la file d'attente et le second processus démarre. Et ainsi de suite… La condition de réquisition ici est donc la contrainte du **quantum alloué**.

NB1 : Si deux processus arrivent en même temps, par défaut l'algorithme FIFO est appliqué pour l'ordonnancement, si cela ne résout pas le problème on précise d'autres critères d'ordonnancement (Algo SJF par exemple).

NB2 : Même si un processus est seul dans la file d'attente, si son temps de service dépasse le quantum alloué, il est interrompu et rechargé dans le processeur.

2- SRT (Shorted Reminded Time)

C'est un algorithme qui trie les processus par rapport à leur temps de service restant (quasiment la même chose que l'algorithme SJF). Cependant, la condition de réquisition est que si un processus de la file d'attente a un temps de service strictement inférieur au **temps de service restant du processus en cours d'exécution**, ce dernier est éjecté du processeur et celui avec le temps de service le plus court prend sa place.

Cet algorithme, étant contraignant par rapport au temps d'exécution, a été optimisé de façon que les processus soient aussi définis par rapport à un seuil d'exécution (Exemple : Si le processus n'a pas atteint 50% d'exécution, il est suspendu et rechargé dans la file d'attente, un autre processus de temps de service plus court venant prendre sa place ; au cas contraire, le processus continue son exécution jusqu'à la fin). Cela permet ainsi d'éviter les changements fréquents de contexte et ainsi optimiser les performances du processeur.

CHAPITRE 3 : MISE EN ŒUVRE DE PROCESSUS (UNIX/LINUX)

I- Généralités :

(Cf. Chapitre 1)

- Ressources nécessaires à un processus :
  - **Ressources matérielles :** processeur, périphériques, etc
  - **Ressources logicielles :** code, contexte d'exécution (compteur ordinal, fichiers ouverts), mémoire…
- Mode d'exécution : dépend de l'utilisateur ayant lancé le processus (utilisateur ou root ou système)

1- Identifiant d'un processus : PID

Chaque processus est identifié par un PID. Chaque PID est spécifique à un processus.

Le PID est un entier de type **pid_t** (équivalent à **int**), **toujours positif**, commençant à partir de 0 et incrémenté.

Lors du démarrage d'un appareil sous Linux, la fonction **start_kernel ()** est appelée par impulsion électrique. Cette fonction créera le processus de PID 0, **le seul créé par cette fonction**. Ce processus occupera la 1ere entrée de la table des processus. Il va initialiser les structures de données du noyau, autoriser les interruptions, créer un thread 1 (qui n'est pas encore exactement un processus, il s'agit à peu près d'un processus léger), puis dormir, lorsque ce processus est réveillé, tous les autres processus sont tués.

Après sa création, le thread 1 se réactive et crée d'autres threads pour gérer le swapping, le cache disque. Puis il va exécuter une des primitives de **exec ()**, ce qui fait que le thread 1 deviendra un **processus INIT**. Ce processus aura 1 comme PID. Par la suite, c'est de lui que naitront tous les autres processus. Finalement, on aura une structure de processus sous forme arborescente.

Pour visualiser les processus on tape la commande **ps** qui va afficher le minimum de processus (souvent les commandes exécutées) (avec les option -aux pour afficher tous les processus qui s'exécutent).

Pour visualiser l'arborescence de processus on tape la commande **pstree**.

Lorsqu'on tape la commande **ls /proc,** tous les nombres affichés sont des répertoires des processus à qui leurs PID ont été pris comme noms de répertoires.

2- **Attributs d'un processus**

Les attributs sont les infos des processus qui permettent de les distinguer des autres.

- IDENTIFICATION :

La fonction **getpid(void)**, qui ne prend aucun argument, renvoie la valeur du PID du processus en question. Cette fonction répond toujours dans le processus courant auquel il a été appelé.

Un processus peut aussi créer un autre processus, qui aura lui-même son propre PID. Si ce processus veut connaître l'identifiant du processus parent alors il va exécuter la fonction **getppid(void)**, qui ne prend non plus aucun argument. Le noyau renverra le PID du processus parent **qui doit obligatoirement ne pas être encore « mort » (bloqué ou terminé).**

- PROPRIETAIRE REEL :

Il s'agit de l'utilisateur (et du groupe auquel il appartient) ayant lancé le processus. Les fonctions sont **getuid(void)** et **getgid(void)** qui ne prennent rien en paramètre, qui renvoient respectivement le **user ID** et le **group ID**.

- PROPRIETAIRE EFFECTIF :

Cet attribut détermine les droits du processus par rapport à l'utilisateur l'ayant lancé. Les fonctions **geteuid(void) et getegid(void)** donnent respectivement les droits du processus découlant de l'utilisateur correspondant, et les droits du processus découlant du groupe correspondant. Ce propriétaire effectif peut modifier les droits du processus par les fonctions **setuid(uid_t uid)** et **setgid(gid_t gid)**.

NB : Le propriétaire effectif n'est pas le même que le propriétaire réel.

II- Création de processus sous UNIX

Les processus sont toujours créés dans le noyau. Ce dernier n'est accessible que par appels système.

L'appel système pour la création de processus est la fonction **fork(void)** qui ne prend rien en paramètre et qui renvoie en retour une valeur de type **pid_t**.

La fonction fork(void) est définie dans unistd.h et la valeur de retour est définie dans sys/types.h. La création de processus ne nécessite aucune ressource au noyau ; raison pour laquelle la fonction ne prend pas de paramètres.

Le noyau peut refuser la demande de création de processus :

- A cause du fait que le processus parent n'a pas assez d'espace mémoire
- Ou bien à cause du fait que le processus parent ne peut plus avoir de processus fils

Dans ce cas, qui est très rare, le noyau renvoie la valeur **-1.** Au cas contraire, le processus fils est créé.

Si le processus fils est créé, la valeur de retour de la fonction fork(void) au processus parent est le **PID du processus fils** et celle au processus fils est **0** (s'il veut connaitre son PID il exécute la fonction getpid(void) et getppid(void) pour connaître le PID du processus parent).

Lorsqu'on fait l'appel de la fonction fork(void), il peut y avoir 2 possibilités, soit le processus père s'exécute avant le fils, soit l'inverse. Ce qui permet de savoir quel processus s'est exécuté en premier et la valeur de retour de la fonction fork(void). La ligne d'instruction ayant pour valeur de pid 0 est celle du processus fils.

NB1 : Si on ne fait rien pour éviter cela, le processus fils et le processus père exécuteront les mêmes instructions. Il se posera le problème de sécurité.

NB2 : Il peut arriver que le ppid du fils est différent du pid du père, cela est dû au fait que le processus père est exécuté (avant le processus fils) et tué ; donc le processus fils devient orphelin et est hébergé par un autre processus père.

NB3 : La commande **ps -aux** permet d'afficher la liste des processus courants dans le système. En 1ere colonne, on a l'utilisateur ou l'entité système ayant lancé le processus. En 2e colonne nous avons le pid du processus. En dernière colonne, on a le répertoire correspondant au processus.

Utilisation typique de fork() :

Si on ne spécifie pas les codes des processus père et fils, il est quasi impossible de les différencier. Il faudra donc approfondir les recherches en ajoutant d'autres segments de code à chaque processus. On peut par exemple utiliser les **structures conditionnelles**.

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
    pid_t status;
    status = fork();
    switch (status) {
        case -1 :
            perror("Creation de processus");
            return -1;
        case 0 : /*Code du fils*/
            printf("[%d] Je viens de naitre\n", getpid());
            printf("[%d] Mon pere %d\n", getpid(), getppid());
            break;
        default : /* Code du pere*/
            printf("[%d] J\'ai engendre\n", getpid());
            printf("[%d] Mon fils est %d\n", getpid(), status);
    }
    printf("[%d] Je termine\n", getpid());
    exit(EXIT_SUCCESS);
}
```

Explication :

Les 2 processus (père et fils) exécuteront tous les deux les instructions dans le bloc switch. La différence réside dans le fait que le processus fils l'exécutera avec la valeur 0 et le processus père l'exécutera avec la valeur du pid du processus fils. Lorsque le processus utilisera le cas 0. Il exécute le bloc de code le correspondant.

Le processus père n'exécutera pas ce bloc de code parce que status pour lui n'aura pas la valeur 0.

Pour contraindre le fils à exécuter uniquement le bloc de code correspondant au case 0, on met l'instruction **break**.

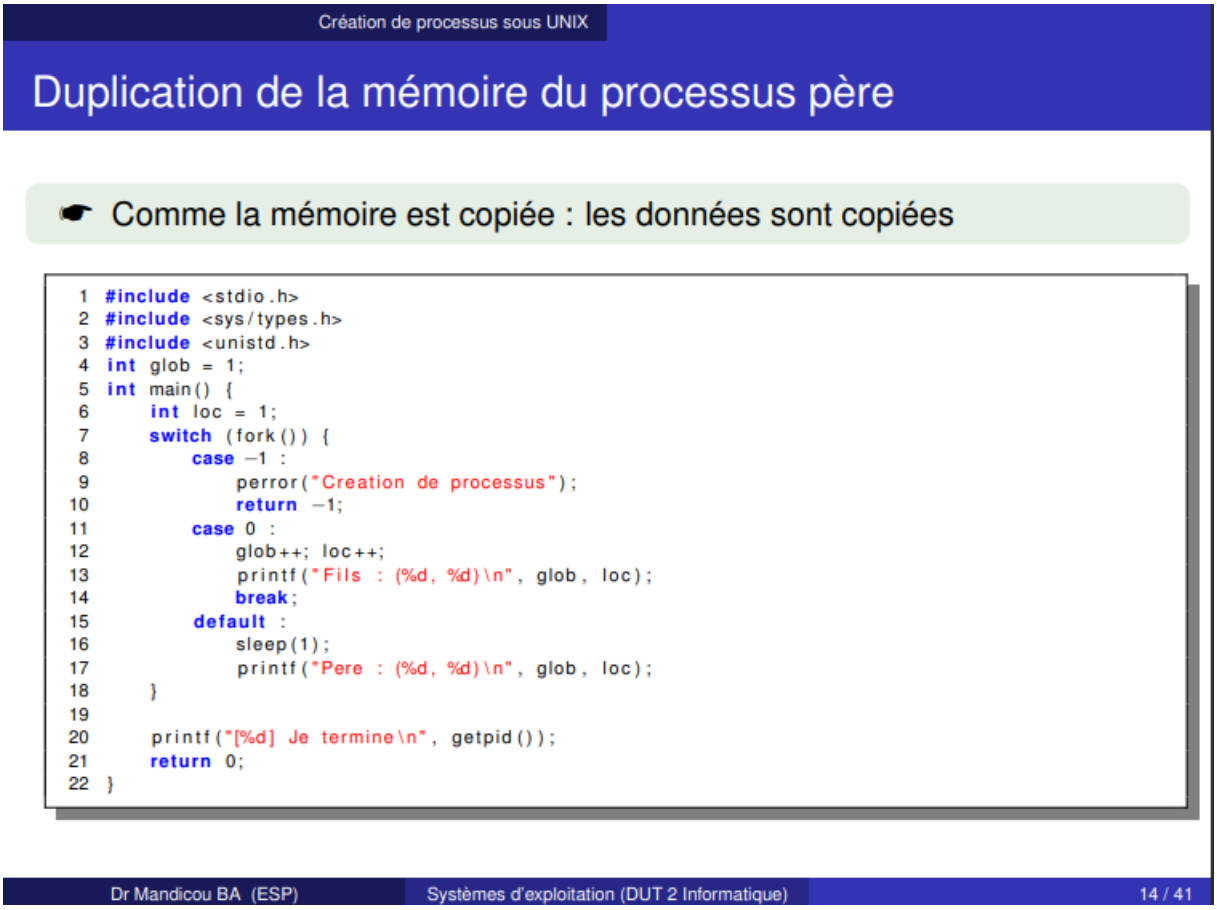
Quant au père il exécutera le bloc de code correspondant à l'instruction *default*.

Il n'est pas possible pour lui de connaître la valeur du pid du fils à l'avance, raison pour laquelle on ne met que le bloc *default* que le processus père exécutera.

Les 2 processus exécuteront ensuite les 2 dernières lignes du code montrant ainsi que les processus sont à l'état « terminé ».

Mémoires des processus père et fils :

Lorsque le processus fils s'exécute, l'espace mémoire du processus père sera copié et donné à ce premier.



Explication :

La variable glob lors de sa création à un espace mémoire identifié par une adresse qui lui sera attribué ; de même que la variable loc.

L'instruction switch prend en paramètre la fonction fork(void). A partir de là, la mémoire du processus père est copiée. Le fils recevra donc les informations de la mémoire du père (variables, données etc). La seule chose qui change sera l'adresse des variables du processus fils. Ce qui fait que les variables du processus ne seront pas les mêmes que celles du processus père.

La fonction fork() donne ses valeurs de retour.

Le fils rentre dans le case 0. Cependant les instructions glob++ et loc++ n'auront effet que sur les variables du **processus fils**.

Lorsque le père rentre dans le case *default* il sera suspendu pendant 1 seconde avant d'exécuter l'instruction qui suit.

NB : Lorsque les espaces mémoires sont dupliqués pour attribuer cela au processus fils, les buffers d'écriture (mémoire d'écriture temporaire) également

le sont. Il se posera donc le problème de la sécurité entre les processus. Ce qui explique le code ci-dessous :

```
1
2  #include <stdio.h>
3  #include <sys/types.h>
4  #include <unistd.h>
5  int main() {
6      printf("avant ");
7      fork();
8      printf("apres\n");
9      return 0;
10 }
```

Pour vider le buffer d'écriture avant la création du processus fils on ajoute '\n' au niveau l'instruction printf(« avant »), ou bien on utilise la fonction **fflush(stdout)** (*stdout* en paramètre).

NB : Lors de la copie de l'espace mémoire, le pid du processus père n'est pas copié, de même que le pid du processus père de ce processus père et ainsi de suite. Le temps d'exécution du processus père n'est pas copié, de même que sa priorité. Les verrous des fichiers associés au processus père ne sont pas non plus copiés.

III- Différents types de terminaison de processus :

En général, un processus termine son exécution après avoir terminé les instructions de la fonction main() et qu'en même temps les instructions se soient exécutées de la même manière. On parle dans ce cas de **terminaison normale**.

Une terminaison est un signal qui vient interrompre/tuer un processus. Il s'agit dans ce cas d'une **terminaison anormale**. Une terminaison anormale peut être initiée par un utilisateur (Ctrl + C par exemple), ou par un autre processus par l'appel système **kill()** ou par le noyau (en cas d'erreur d'exécution non prise en charge par le noyau).

Actions réalisées à la terminaison d'un processus :

Considérons un processus P1 avec comme fils P2 qui à son tour, a comme fils P21, P22 et P23.

Si P2 meurt, les ressources utilisées par P2 seront libérées et les fichiers ouverts par P2 seront fermés. En même temps, P21, P22 et P23 deviennent des processus orphelins et auront comme nouveau père le processus **init** et ils auront tous comme PPID = 1.

En outre, le processus P1 reçoit un signal du noyau lui informant de la mort du processus P2. C'est un signal **SIGCHLD.** Par défaut, ce signal est ignoré par P1 ; dans ce cas P2 deviendra un **processus zombie** : son PCB ne sera pas effacé et il ne sera pas effacé de la table des processus et il gardera toujours son PID d'origine.

NB : L'appel système permettant de faire tout ce qui a été décrit précédemment est l'appel système **exit(x)** (qui est une fonction qui appelle le véritable appel système **_exit(x)**). Cette fonction retourne une valeur comprise entre **0** et **255**. Et en fonction des valeurs retournées, le programmeur décide des actions à faire. En cas de terminaison anormale, la fonction renvoie **-1**.

Comment éviter d'avoir P2 comme processus zombie ? :

La fonction dont on aura besoin est la fonction **wait(int* status)** de type **pid_t**.

Cette fonction suspend le processus P1 jusqu'à ce qu'un de ses processus fils termine de s'exécuter (en supposant que le processus P1 a plusieurs fils). Il se présentera 3 cas :

- Le père n'a pas encore de fils mort au moment de l'appel de la fonction wait(). Dans ce cas, le père est bloqué jusqu'au décès d'un processus. Un signal SIGCHLD sera envoyé à ce moment-là au processus père, en même temps que *status* qui était mis en paramètre (cela permettra au père de connaître le PID du processus fils mort).
- Le père appelle la fonction wait() **après** que le processus fils est mort. Dans ce cas-ci, le signal SIGCHLD + status sont envoyés automatiquement au processus père mais est ignoré. Entre le temps de la mort du processus fils et l'appel de la

fonction wait() par le processus père, le processus fils reste un processus zombie. Après l'appel de la fonction wait(), le signal SIGCHLD n'est plus ignoré par le père et le processus fils est tué.

- Le processus père appelle la fonction wait() alors qu'**il n'a pas encore de processus fils**. Dans ce cas, automatiquement, la fonction renvoie la valeur -1.

NB : Dans le cas où le processus fils devient un processus orphelin et après un processus zombie, il sera adopté par init qui le tuera instantanément.

La fonction wait() prend en paramètre un pointeur de type **int.** Ce paramètre contient des informations sur la cause de la mort du processus fils. On a plusieurs cas :

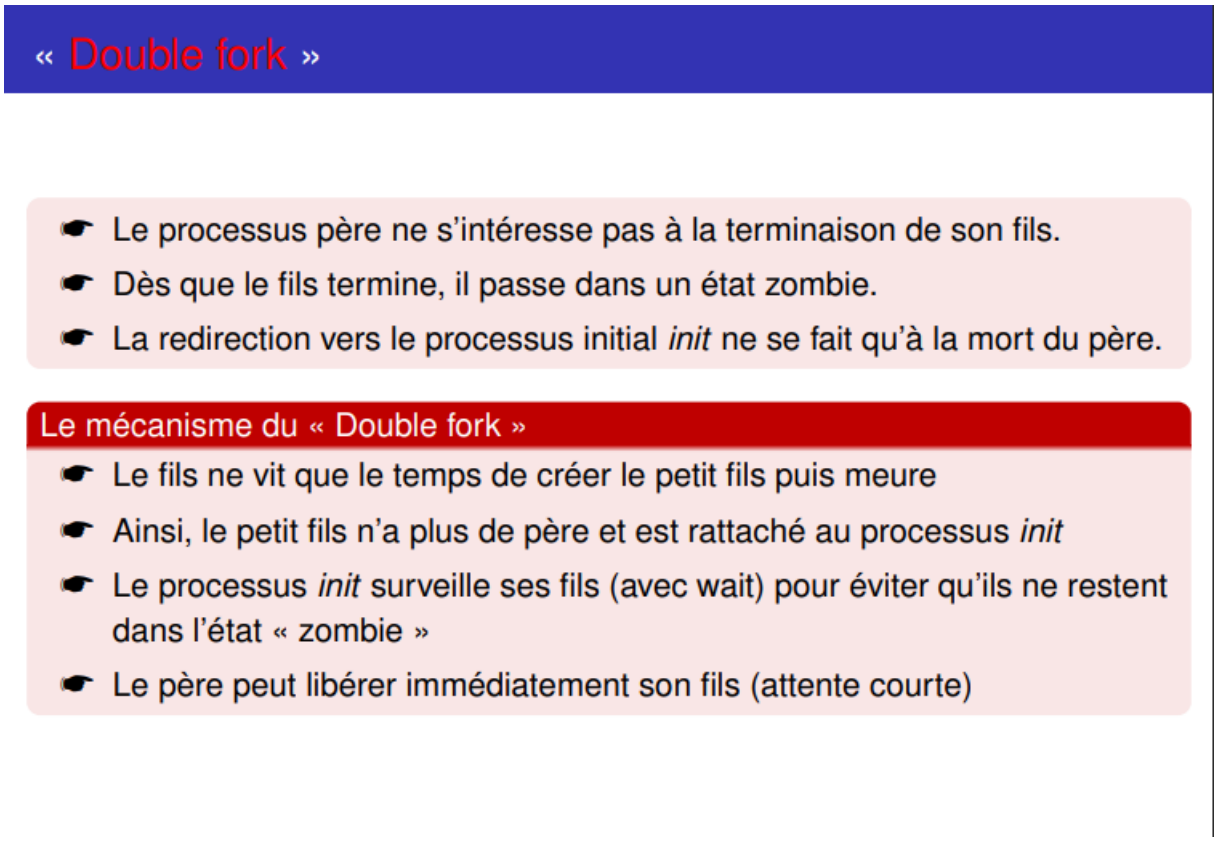
- Le programmeur ignore les causes de la mort du processus fils. Dans ce cas on aura comme syntaxe : *wait(NULL)*. Et donc, au moment de la mort du processus, le processus père ne connaîtra que le PID du processus fils mort.
- Le programmeur souhaite connaître les causes de la mort du processus fils. Par suite, il fera un passage par référence avec la syntaxe suivante : *wait(&status)* (status déclarée au préalable) et il aura accès à cette zone de mémoire à travers des **macros.** Il vérifiera si la terminaison est normale ou anormale.
    - S'il s'agit d'une terminaison normale, Il utilisera le bloc de code : *if (WIFEXITED(status) [passage par valeur])*. Le programmeur peut exécuter le code : *int x = WEXITSTATUS(status)* qui sera la valeur retournée par le processus fils mort.
    - S'il s'agit d'une terminaison anormale, il utilisera le bloc de code : *if (WIFSIGNALED(status))* ; et dans le bloc, le programmeur peut exécuter l'instruction suivante : *int s = WTERMSIG(status)* pour avoir les informations sur le signal d'extinction du processus.

IV- PROCESSUS ORPHELINS – PROCESSUS ZOMBIES

Un **processus orphelin** est un processus non encore mort dont le père est mort. Son nouveau parent sera donc le processus init et leur PPID aura pour valeur 1.

Un **processus zombie** est un processus ayant terminé son exécution mais qui n'a pas été tué. Il sera tué lorsque le processus père aura fait appel à la fonction wait.

Souvent lorsque le processus père fait appel à la fonction wait(), il est bloqué jusqu'à ce que le processus fils décède. Dans ce cas, il se posera le problème de la performance des appareils vu que plusieurs processus sont bloqués. Pour éviter que le processus père ne se bloque, le processus père peut faire appel au **double fork.** En d'autres termes, le processus père crée un processus fils et fait appel à la fonction wait ; ce processus fils n'aura qu'une chose à faire : créer son processus fils à lui qui exécutera les instructions nécessaires. Ce qui fait que le 1er processus fils ne vivra que pour créer le processus suivant et donnera la possibilité au processus père de reprendre son cours normal d'exécution.



V- PRIMITIVES EXEC()

Un recouvrement est un procédé en programmation système qui permet d'exécuter un programme existant. Dans ce cas, le programme externe existant qui est chargé va substituer le processus appelant, ce qui fait que les instructions après l'appel du programme seront perdues par défaut.

La primitive exec() a 2 parties principales :

- execl() pour lesquels le nombre d'arguments du programme chargé est connu.
- execv() pour lesquels le nombre d'arguments du programme chargé n'est pas connu. Dans ce cas, ces primitives prévoient un tableau pour garder les futurs arguments.

1- execl() est constitué de 3 parties :

- execl()

Le 1er paramètre : char *path qui est le chemin du fichier

Le 2e paramètre : char *argv[] qui est la liste des arguments

Le 3e paramètre : NULL qui marque la fin de la liste des arguments

- execle()

Les 3 premiers paramètres sont identiques à la différence qu'il y a un 4e paramètre spécifie l'environnement d'exécution du programme chargé

- execlp()

La spécificité est que le programme appelant et le programme appelé sont dans le même répertoire

2- execv()

Les primitives exec() : Second groupe

- Second groupe d'exec() : les arguments sont passées sous forme de tableau :

```
1  int execv(char *path,char *argv[])
2
3  /* argv : pointeur vers le tableau contenant les arguments */
4
5  int execve(char  *path,char  *argv[],char *envp[])
6
7  int execvp(char  *file,char  *argv[])
8
```

- Lors de l'appel d'une primitive exec(), il y a recouvrement du segment d'instructions du processus appelant
- Ce qui implique qu'il n'y a pas de retour d'un exec() réussi (l'adresse de retour a disparu)
- Le code du processus appelant est détruit

```
1  #include <stdio.h>
2  int main() {
3      execl("/bin/ls","ls",NULL) ;
4      printf ("je ne suis pas mort\n") ;
5      return 0;
6  }
```

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
int main() {

    if ( fork()==0 ) execl( "/bin/ls","ls",NULL) ;

    else {

        sleep(2) ; /* attend la fin de ls pour exécuter printf() */

        printf ("je suis le p're et je peux continuer") ; }
}
```

☛ Dans ce cas, le fils meurt après l'éxécution de ls, et le père continue à vivre et exécute printf().

CHAPITRE 4 : INTER-BLOCAGE, SYNCHRONISATION ET COMMUNICATION DE PROCESSUS

I- INTER-BLOCAGE DE PROCESSUS

On dit que 2 processus sont en interblocage lorsque chaque processus a besoin d'une ressource détenue par l'autre sans que les processus détenant ces ressources ne parviennent à l'abandonner.

- Un ensemble de processus est en inter-blocage si chaque processus attend un événement que seul un autre processus de l'ensemble peut engendrer
- 2 (ou +) processus sont en attente de la libération d'une ressource attribuée à l'autre
- Un ensemble de processus est bloqué !

1- Conditions menant à un inter blocage

1. Ressource limitée (exclusion mutuelle)
2. Attente circulaire : il doit y avoir au moins deux processus chacun attendant une ressource détenu par un autre
3. Occupation et attente : les processus qui détiennent des ressources peuvent en demander de nouvelles
4. Pas de réquisition(non préemption) : les ressources sont libérées par le processus

2- Stratégie de gestion

- Détecter et résoudre
- Prévenir la formation d'inter-blocages en empêchant l'apparition d'une des 4 conditions
- Éviter les inter-blocages en allouant de manière « intelligente » les ressources

a- Détecter et résoudre :

Une approche serait de retirer la ressource à un processus afin de la donner à un autre. Néanmoins cette action n'est pas toujours possible.

Une autre approche serait de faire une sauvegarde du système à chaque fois qu'une version stable est atteinte. Ce qui fait qu'au moment de l'inter blocage, on revient à la dernière version stable du système.

b- Prévision de l'inter blocage

Une approche est de donner la ressource à un processus avec un quantum alloué et dans le même temps, les autres processus ayant besoin de la ressource sont bloqués.

Une autre approche sera que les processus demandent la même ressource en même temps. Ce qui fait que chaque processus utilise la ressource à tour de rôle, ce qui pourrait éviter le conflit de ressources.

Il existe une 3e approche où tous les processus demandent les ressources dont ils ont besoin. Et donc les processus sont exécutés si et seulement si les ressources nécessaires sont disponibles, bien évidemment avec un quantum alloué.

c- Allouer « correctement »

Une première approche est d'allouer les ressources aux processus les plus productives possibles. De plus, il va éviter une mobilisation non productive des ressources.

La 2e approche est l'algorithme du banquier.

- Immobilisation non productive de ressources

☛ « Algorithme du banquier » :

- L'algorithme du banquier utilise la métaphore d'un banquier d'une petite ville qui dispose d'un certain montant d'argent à prêter à ses clients.
- Les clients sont des processus, l'argent correspond aux ressources et le banquier à l'OS.
- L'algorithme vérifie si une requête mène à un état non sûr et la refuse en telle cas.
- Sinon, tant que les requêtes laissent un état sûr derrière, elles sont allouées
  - Annonce des besoins
  - Déterminer à quel moment la ressource peut être alloué en sécurité

☛ On peut s'attaquer aux quatre conditions de l'inter-blocage en essayant de les éliminer, empêchant ainsi tout inter-blocage.

1. La condition d'exclusion mutuelle : le fait de ne pas réserver exclusivement les ressources permet d'éliminer cette condition.
   - démon d'impression est un exemple de solution de ce type
2. La condition de détention et d'attente : on peut exiger que tous les processus commandent toutes leurs ressources avant de commencer à les utiliser.
   - solution, en plus de mal utiliser les ressources exigent de connaître par avance les ressources nécessaires.
   - En tel cas, un algorithme du banquier ferait l'affaire.
3. La condition de non-préemption : cette solution est au mieux très délicate et au pire impossible. Penser à une table traçante que l'on retirerait.

☛ On peut s'attaquer aux quatre conditions de l'inter-blocage en essayant de les éliminer, empêchant ainsi tout inter-blocage.

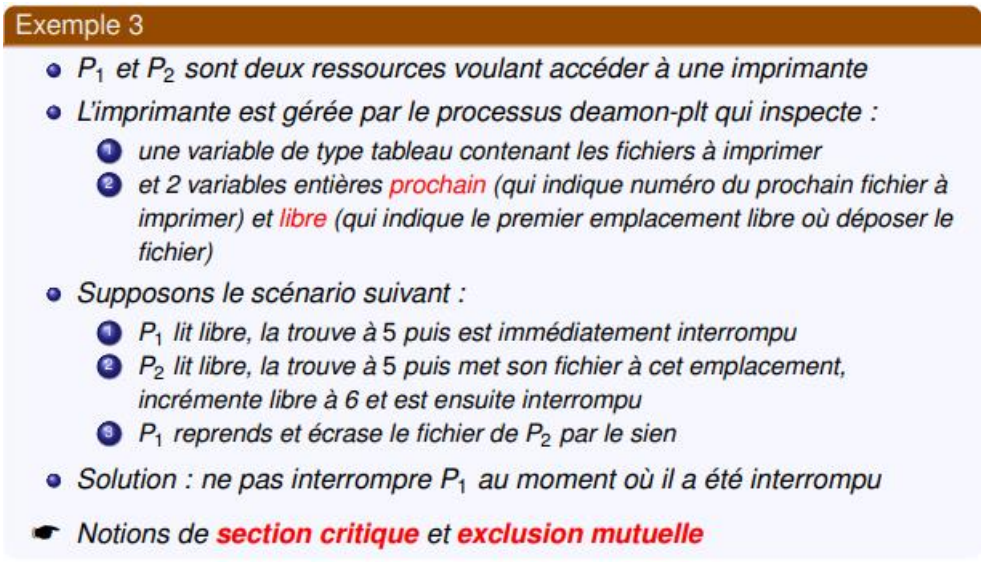
4 la condition d'attente circulaire :

- peut interdire l'accès à plusieurs ressources, ce qui est un peut contraignant
- Une autre solution consiste à ordonner les ressources et exiger que celles-ci soient réservées dans un ordre précis.
- Un processus demandant une ressource d'ordre inférieure à l'une qu'elle détient se la verra alors refusée.
- Il peut arriver qu'il ne soit pas possible en pratique d'ordonner les ressources
- De plus, cela est contraignant pour les processus.

II- SEMAPHORES

1- Accès concurrent

Il se produit un problème d'accès concurrent lorsque 2 processus veulent la même ressource (partagent la même ressource) alors que celle-ci ne peut pas être partagée. L'accès en lecture n'est pas un souci.



2- Section critique

Une section critique est une instance où un processus se trouve et que l'on ne doit pas interrompre. Elle est une portion du programme qui contient une ou des ressource(s) critique(s) Dans ce cas, les autres processus doivent être bloqués au risque de créer une incohérence. Une section critique contient des variables ou des ressources partagées.

3- Ressource critique

Une ressource est critique lorsque de l'accès concurrent il peut en suivre une incohérence. Les processus voulant accéder à cette ressource critique sont dits **en compétition.**

4- Exclusion mutuelle

Une exclusion mutuelle se produit si une ressource n'est accessible que par 1 processus. Si ce processus est interrompu, il faudra attendre la reprise de ce processus, qu'il termine son exécution et libère la ressource avant que cette ressource ne soit remise en jeu entre les processus en file d'attente.

Pour résoudre ces problèmes, l'une des solutions est d'utiliser les **sémaphores.**

5- Sémaphores

Un sémaphore est un compteur entier indiquant combien de fois ou combien de processus peuvent utiliser la ressource en même temps.

Est associé au sémaphore une file d'attente indiquant les processus en attente de la ressource. Le nombre de processus en attente est connu par la valeur de l'entier du sémaphore (valeur membre). Par exemple, si la valeur est -3 alors il y a 3 processus en attente de la ressource. Ce qui permet d'enlever le défaut des structures conditionnelles car, si la ressource est non disponible, les processus en attente sont donc obligatoirement bloqués jusqu'à la libération par un processus.

Pour cela, on associe à ce sémaphore 2 primitives P() et V().

On distingue 2 types de sémaphores : **binaire** et **n-aire**.

Un sémaphore binaire ne peut prendre que 2 valeurs : 0 ou 1. Il est utilisé dans le cas des exclusions mutuelles où soit la ressource est disponible soit elle ne l'est pas.

Un sémaphore n-aire peut prendre n valeurs. Il indique le nombre de processus pouvant utiliser la ressource en question.

NB : Un sémaphore doit être **obligatoirement** initialisé et initialisé correctement. L'initialisation dépend d'emblée de la disponibilité de la ressource. Autrement dit, à l'état initial, si la ressource est disponible, le sémaphore est initialisé à sa **valeur maximale** ; sinon le sémaphore est initialisé à sa **valeur minimale**.

Pour demander l'utilisation de la ressource protégée par le sémaphore, on utilise la primitive **P()** et pour terminer l'utilisation de la ressource, on utilise la primitive **V()**.

- Lorsqu'il y a une demande d'utilisation, la primitive P() décrémente la valeur du sémaphore de 1. Si cette valeur est supérieure ou égale à 0, la demande d'utilisation est approuvée au processus. Sinon le processus demandeur passe dans la file d'attente associé au sémaphore.
- Lorsque le processus finit d'utiliser la ressource, celui-ci exécute la primitive V(). Il incrémente la valeur du sémaphore de 1. Si cette valeur est inférieure ou égale 0, cela signifie qu'il y avait au moins 1 processus qui était en file d'attente du sémaphore. Dans ce cas, il prend 1 processus de la file d'attente (selon l'algorithme d'ordonnancement).

Lorsqu'il y a le cas d'exclusion mutuelle et que la ressource est disponible, alors 1 processus prend la ressource et le reste est dans la file d'attente.

NB : Dans le cas où il y a 2 processus synchronisés, le processus qui doit produire le résultat dont le processus suivant a besoin passe en premier dans la ressource, le second en attendant exécute la primitive P() et prendra la ressource à la fin de l'exécution du premier.

III- COMMUNICATION INTERPROCESSUS

Nous avons 2 modes de communication :

- **Communication à mémoire partagée**
- **Communication par échange de messages**

1- Communication par échange de messages

Comme son nom l'indique, cette communication fonctionne sur le fait que 2 processus s'échangent des messages. Les canaux de communication sont de 3 types :

- **Signaux**
- **Tubes de communication**
- **Sockets**

Les **signaux** et les **tubes de communication** ne sont valables que si les processus communiquant se trouvent dans le même système.

Les **sockets** sont faits pour faire communiquer 2 processus de 2 systèmes distincts.

2- Communication à mémoire partagée

Les processus octroient une portion de leur espace mémoire en accès en **lecture seule** aux autres processus. Dans ce type de messages nous pouvons remarquer qu'il y a :

- Les files de messages
- Les tableaux de sémaphores
- Les segments de mémoire partagée

La **communication par échange de messages** et la **communication à mémoire partagée** constituent la **NORME POSIX DU SYSTEM V DE LINUX.**

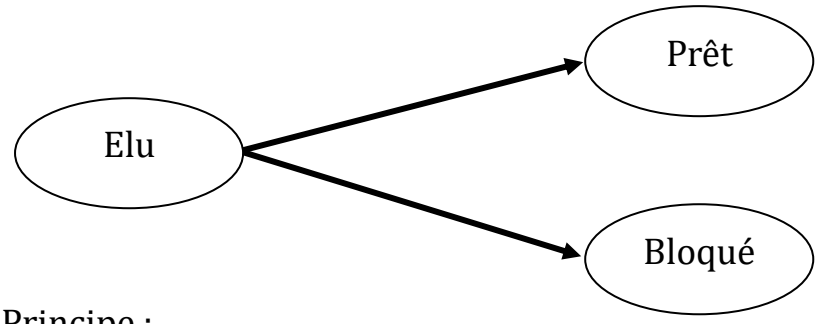
NB : La **communication à mémoire partagée** nécessite que les processus communiquant soient dans le même système.

A- COMMUNICATION A BASE DE SIGNAUX

Les signaux sont des interruptions logicielles. Concrètement ils sont sous la forme d'impulsions électriques.

- Les **interruptions logicielles** ou **signaux** sont utilisées par le système d'exploitation pour aviser les processus utilisateurs de l'occurrence d'un événement important.
- De nombreuses erreurs détectées par le matériel, comme l'exécution d'une instruction non autorisée (une division par 0, par exemple) ou l'emploi d'une adresse non valide, sont converties en signaux qui sont envoyés au processus fautif.
- Ce mécanisme permet à un processus de réagir à cet événement sans être obligé d'en tester en permanence l'arrivée.
- Les processus peuvent indiquer au système ce qui doit se passer à la réception d'un signal

Les signaux n'interviennent que dans 2 transitions d'état de processus comme l'indique ce schéma :



Principe :

Cette communication se base sur un ensemble de cardinalité finie de signaux (64 avec la norme POSIX).

Les signaux POSIX sont répartis en **signaux classiques (1 – 32)** et les **signaux en temps réel (33-64)**. Il convient de noter qu'un signal classique ne signifie pas forcément que la réponse ne se fera pas en temps réel comme c'est le cas pour les signaux en temps réel.

A chaque signal est associé un traitement par défaut. Ce dernier peut être pour :

- **Arrêter le processus** qui peut se faire de 2 manières :
  - **Arrêt brutal**
  - **Arrêt par génération de fichier core**
- **Ignorer le signal**
- **Redéfinir le traitement par défaut**

Il existe des signaux dont le traitement par défaut n'est pas modifiable.

- On peut ainsi ignorer le signal, ou bien le prendre en compte, ou encore laisser le SE appliquer le comportement par défaut :
  - en général tuer le processus
- Certains signaux ne peuvent être ni ignorés, ni capturés
- Si un processus choisit de prendre en compte les signaux qu'il recroit, il doit alors spécifier la procédure de gestion de signal.
- Quand un signal arrive, la procédure associée est exécutée.
- A la fin de l'exécution de la procédure, le processus s'exécute à partir de instruction qui suit celle durant laquelle l'interruption a eu lieu

Synthèse

- Les signaux sont utilisés pour établir une communication minimale entre processus, une communication avec le monde extérieure et faire la gestion des erreurs.

NB1 : Avec les signaux il ne peut pas y avoir de réception multiple. 1 bit est réservé à 1 signal. En d'autres termes, lorsqu'un processus reçoit un signal I, même si ce dernier est envoyé plusieurs fois, les autres occurrences sont ignorées. De plus, si un processus reçoit plusieurs signaux différents, le traitement suivant sera fait :

- Classement des signaux par ordre de priorité (généralement plus le numéro de signal est petit, plus sa priorité est grande)
- Pour chaque signal, on vérifie si l'utilisateur l'ayant lancé a les droits d'utiliser ce signal sur ce processus. Si oui, le signal est exécuté, sinon il est ignoré.

NB2 : Le noyau prévoit toujours 2 octets par processus pour la réception des signaux.

NB3 : Au niveau du PCB du processus, il y aura :

- Le champ **gsig qui** est un pointeur qui pointe vers le traitement par défaut du signal à traiter
- Le champ **sigset** qui contient les signaux reçus non encore traités (2 octets).

Décision prise à l'arrivée d'un signal

- Tous les signaux ont une routine de service, ou une action par défaut. Cette action peut être du type :
  - terminaison du processus
  - ignorer le signal
  - Créer un fichier *core*
  - Stopper le processus
  - La procédure de service ne peut pas être modifiée
  - Le signal ne peut pas être ignoré

NB4 : Des noms sont également associés aux numéros des signaux. Son nom a pour format : **SIGXXXX**

Exemple: SIGKILL => 9

SIGTERM => 15

SIGCHLD => 17

SIGALARM => 14

Ceci a des avantages. Non seulement c'est facile à retenir, mais il permet également d'éviter les changements d'implémentation de numéros de signal en passant d'un environnement à un autre.

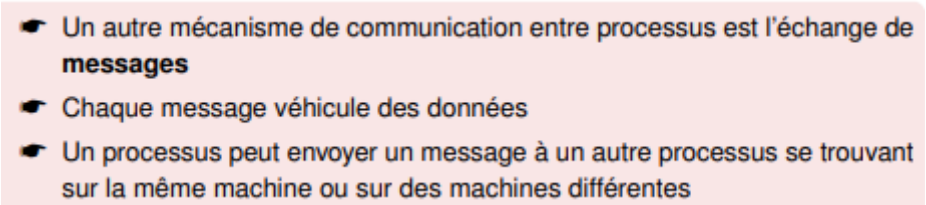
NB5 : La communication par signaux effectue un **échange de messages** et non un échange de données. Les signaux uniquement sont transmis et non leurs traitements.

Inconvénients :

- **Possibilité de perte ou d'ignorance des signaux** (réception de plusieurs signaux à la fois ou le fait d'ignorer les traitements des signaux)
- **Communication minimale** : seuls les messages sont transmis et non les données

B- TUBES DE COMMUNICATION

Un tube de communication est placé entre 2 processus pour permettre l'échange de messages et de données. Il s'agit d'une alternative à la communication par signaux.



Au niveau des extrémités du tube, on y trouve des descripteurs de fichiers.

Le descripteur **0** est pour la **lecture**. Le descripteur **1** est pour l'**écriture**.

Lors de l'envoi, le **descripteur 0 de l'expéditeur** et le **descripteur 1 du destinataire** sont fermés. Nous pouvons donc en déduire que la communication par tube est **unidirectionnelle**.

De ce fait, pour avoir une communication bidirectionnelle, il faudra avoir 2 tubes de communication allant en sens inverses.

Nous avons 2 types de tubes : les **tubes anonymes (unnamed pipe)** et les **tubes nommés (named pipe)**.

<table>
<tr><th>Tubes anonymes</th><th>Tubes nommés</th></tr>
<tr><td>Pas de noms
Pas dans le système de fichiers
Existent seulement dans la mémoire du processus
Connus que par filiation</td><td>Ont des noms
Présents dans le système de fichiers
Sont accessibles par n'importe quel processus ayant les droits</td></tr>
<tr><td>Faible taille : 4 ko</td><td>Grande taille : 40 ko</td></tr>
<tr><td colspan="2">1- Les tubes sont de type FIFO
2- La quantité de données à écrire simultanément est majorée par la taille. Un tube plein n'est plus accessible en écriture. Un tube vide n'est plus accessible en lecture. Au moment de l'écriture, un autre processus ne peut pas lire les données
3- Le tube est protégé par un sémaphore d'exclusion mutuelle.
4- La lecture est destructive. Dès la lecture des données, celles-ci sont détruites
5- Impossible d'écrire si le tube est plein
6- Les tubes sont associés à un signal SIGPIPE qui est un signal qui effectue un arrêt de processus avec génération de fichier core</td></tr>
<tr><td>Détruite par l'appel système close()</td><td>Existe dans le système de fichiers jusqu'à sa suppression</td></tr>
<tr><td>Détruite après la terminaison d'un processus utilisant le tube</td><td>Nécessité de faire une opération de suppression pour détruire le tube.</td></tr>
<tr><td>Suppression du tube dès sa fermeture</td><td></td></tr>
</table>

C- SOCKETS

Un socket est un couple (@IP, N°Port). Elle permet d'effectuer la communication réseau. Elle fonctionne selon le principe suivant :

Le noyau envoie les paquets au switch/routeur de son réseau. Ce dernier vérifie sa table d'adressage afin de voir qui est le routeur destinataire si celui-ci ne se trouve pas dans son réseau. Arrivé au routeur destinataire, celui-ci regarde l'adresse hôte du destinataire et l'envoie à la carte réseau de ce dernier. Après cela, le paquet transite entre les 7 couches ISO jusqu'à atteindre la couche d'application où le message sera reçu. (Détails : cf. Cours Réseaux DUT1).

NB : **Appels systèmes**

Les appels systèmes sont les fonctions associées aux actions exécutées sur le noyau par un utilisateur.

Exemple : fork (), la commande mkdir donnant lieu à l'appel système mkdir()

- Un appel système est un moyen de communiquer directement avec le noyau de la machine
- Le noyau regroupe toutes les opérations vitales de la machine
- Ainsi il est impossible d'écrire directement sur le disque dur
- L'utilisateur doit passer par des appels systèmes qui contrôlent les actions qu'il fait
- Ceci permet de garantir :
  1. la sécurité des données car le noyau interdira à un utilisateur d'ouvrir les fichiers auxquels il n'a pas accès
  2. l'intégrité des données sur le disque. Un utilisateur ne peut pas par mégarde effacer un secteur du disque ou modifier son contenu
- Ainsi, les appels système sont les fonctions permettant la communication avec le noyau
  - open, read, write, etc.

- Travaillent en relation directe avec le noyau
- Retournent un entier positif ou nul en cas de succès et -1 en cas d'échec
- Par défaut le noyau peut bloquer les appels systèmes et ainsi bloquer l'application si la fonctionnalité demandée ne peut pas être servie immédiatement
- Ne resservent pas de la mémoire dans le noyau.
- Les résultats sont obligatoirement stockés dans l'espace du processus (dans l'espace utilisateur)
- il faut prévoir cet espace par allocation de variable (statique, pile) ou de mémoire (malloc(). . .)

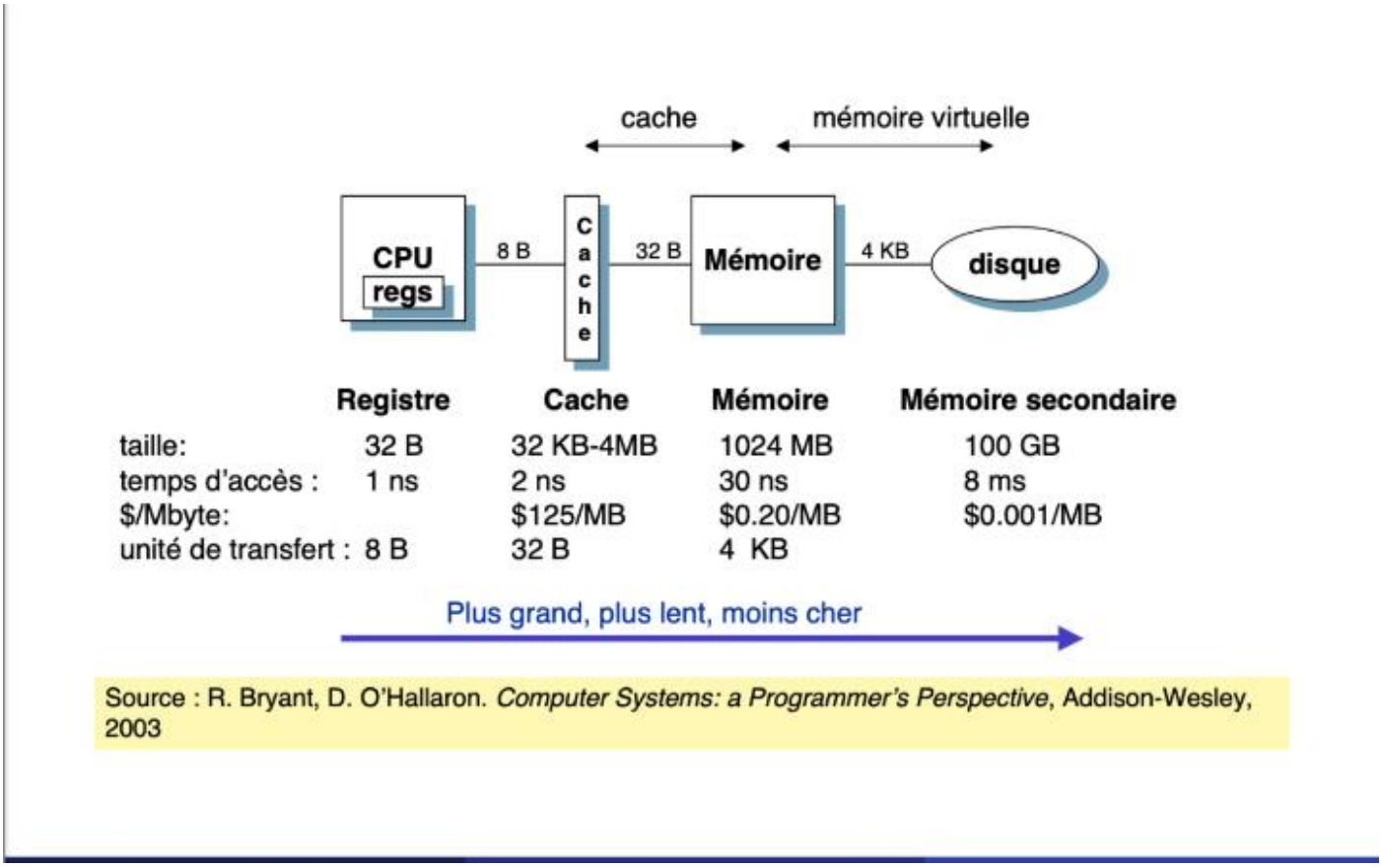
CHAPITRE 5 : GESTION DE LA MÉMOIRE

Nous nous sommes intéressés jusque-là aux processus du point de vue fonctionnel, c'est-à-dire l'exécution avec les instances, les communications, les ressources etc. Pour ce chapitre nous allons aborder la gestion de la mémoire pour les processus.

I- INTRODUCTION A LA GESTION DE LA MÉMOIRE

La mémoire principale est l'espace où se trouve les processus avec tout ce qu'il faut pour leur exécution. Il s'agit d'un espace à gérer avec attention pour que les processus s'exécutent normalement. Avec le temps, la taille de la mémoire principale des machines augmente conséquemment. De même, les besoins en mémoire des programmes augmentent.

De plus, il n'est pas possible de connaître à l'avance la taille mémoire que peut occuper un processus, ensuite la mémoire principale est plus petite, plus rapide et plus volatile que la mémoire secondaire. Même chose que la mémoire cache par rapport à la mémoire principale.



Source : R. Bryant, D. O'Hallaron. *Computer Systems: a Programmer's Perspective*, Addison-Wesley, 2003

La mémoire en générale se divise en 2 catégories

- La **mémoire vive** comme la RAM
- La **mémoire de masse** comme les disques durs, les clés USB etc

Du fait de la différence de rapidité entre les deux mémoires, il faudra donc avoir une entité capable de gérer les 2 mémoires de manière efficace. De plus, la mémoire est un espace protégé, il ne peut pas y avoir de violation d'espace.

IDEE INTUITIVE DE LA GESTION DE MÉMOIRE

La mémoire principale (encore dite mémoire physique) est perçue comme un tableau avec plusieurs zones mémoire. Le processeur extrait les informations de cette mémoire à travers le **MMU (Memory Management Unit)** qui a pour rôle de suivre l'évolution de l'espace mémoire, d'effectuer l'allocation ou la libération des zones mémoires, de contrôler le **swapping** (déplacer des instructions de la mémoire principale vers la mémoire secondaire).

Nous avons 2 types d'adresses :

- **Adresses logiques** ou **adresses virtuelles** qui sont gérées par le CPU
- **Adresses physiques** qui sont présentes au niveau de la mémoire physique

Entre les 2 types d'adresses, se trouvent les MMU se chargeant de faire la correspondance entre les adresses physiques et les adresses virtuelles.

II- GESTION BASIQUE DE LA MÉMOIRE

1- MONOPROGRAMMATION SANS VA-ET-VIENT NI PAGINATION

Une des options était d'allouer une portion de l'espace mémoire où l'on stocke le SE avec sa RAM, le reste est pour le processus.

Une deuxième option était d'allouer une portion de l'espace mémoire où l'on stocke le SE avec sa ROM, le reste étant pour le processus.

Une troisième option était d'allouer une portion de l'espace mémoire où l'on stocke le SE avec sa RAM et d'allouer une portion de l'espace mémoire où l'on stocke le SE avec sa ROM, le reste étant pour le processus.

Ces techniques ont été abandonnées du fait des problèmes de la monoprogrammation.

2- MULTIPROGRAMMATION AVEC PARTITION FIXE

Comme on est dans un système multiprogrammé, il faudra donc partitionner la mémoire et donner à chaque processus une partition.

Le partitionnement se fait au **démarrage du SE**. Comme les programmes sont de tailles différentes, les partitions n'auront pas la même taille. Comment ça se fait ?

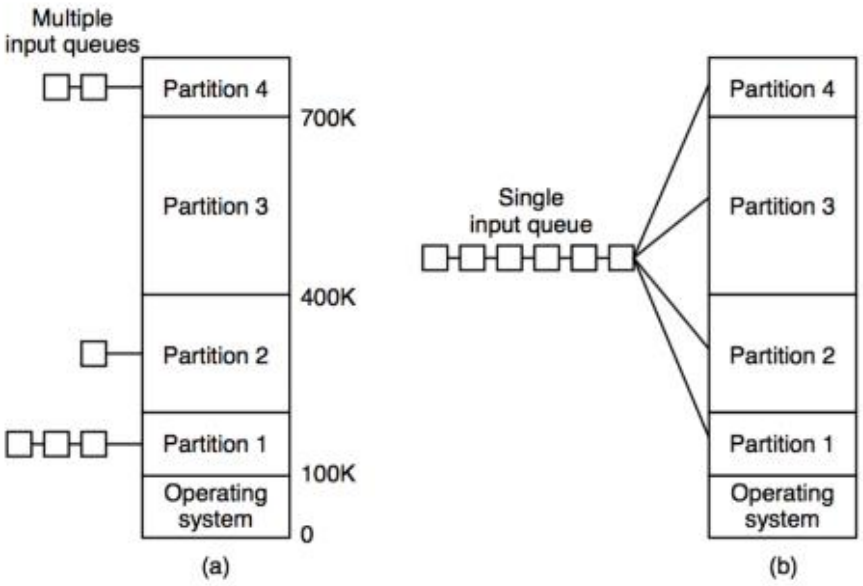


Figure – Partitions fixes de la mémoire

A chaque partition effectuée, on affecte une file d'attente. Ce qui fait que les files d'attente ne contiendront que les processus demandant une taille correspondant à la

partition. L'inconvénient est que si une file d'attente est pleine et qu'un autre processus de même taille arrive, celui-ci ne pourra pas être rangé dans la file.

Pour éviter cela, on fait en sorte d'avoir une seule file d'attente commune à toutes les partitions. Cette file est gérée par FIFO. Si une des partitions est libérée, on y met le processus qui peut tenir dans cette taille.

Ces techniques ont également été abandonnées car le partitionnement fixe est un problème et quelque soit la partition effectuée, il y aura au moins un processus non satisfait. Pour résoudre cela, nous sommes partis vers le **partitionnement dynamique** qui est plus simple du point de vue espace mémoire, mais plus complexe du point de vue implémentation.

- Le fait de ne plus fixer le partitionnement de la mémoire rend plus complexe la gestion de l'espace libre
- Cependant il faut pouvoir trouver et choisir rapidement de la mémoire libre pour l'attribuer à un processus
- Il est donc nécessaire de mettre en œuvre des structures de donnée efficaces pour la gestion de l'espace libre

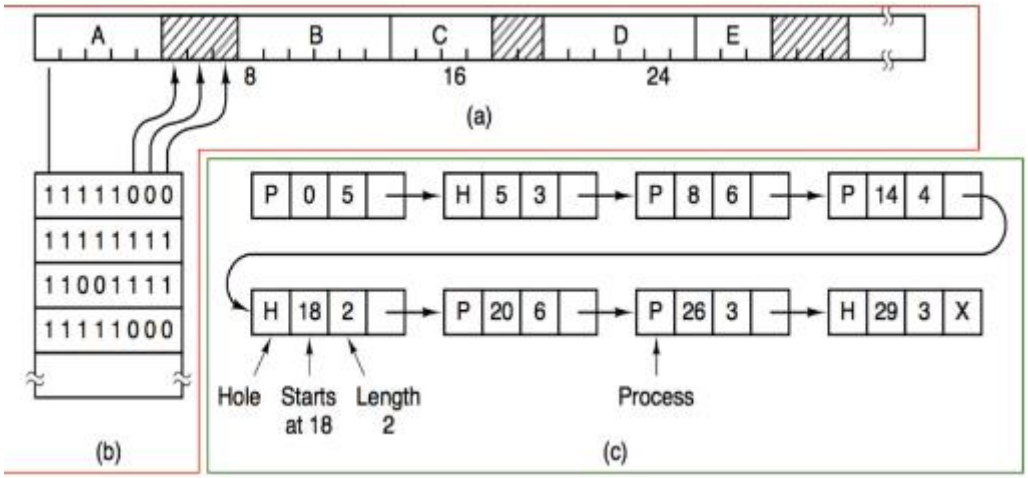
III- SWAPPING

- Swapping : un système qui peut déplacer les processus sur le disque quand il manque d'espace
- Si on n'exploite pas au mieux l'espace libre en mémoire principale :
  - Le mouvement des processus entre la mémoire et le disque (va-et-vient) risque de devenir très fréquent,
  - Sachant que les accès au disque son très lents, les performances du système risquent alors de se détériorer rapidement.
- Pour exploiter au mieux l'espace libre en mémoire principale : partitionnement dynamique de la mémoire
- Le fait de ne plus fixer le partitionnement de la mémoire rend plus complexe la gestion de l'espace libre

Comme nous effectuons un partitionnement dynamique, un processus n'aura donc que l'espace nécessaire à son exécution. Le fait maintenant de pouvoir charger et décharger les processus entre les 2 mémoires est ce qu'on appelle le **swapping**. Pour cela il faudra avoir une cartographie complète de l'espace mémoire. De cette façon, si un espace est libre, les nouveaux processus seront chargés à cet endroit. Sinon on effectue le swapping en déchargeant des processus et en rechargeant d'autres.

1- GESTION DES ZONES LIBRES DE LA MEMOIRE

a- GESTION PAR TABLEAU DE BITS



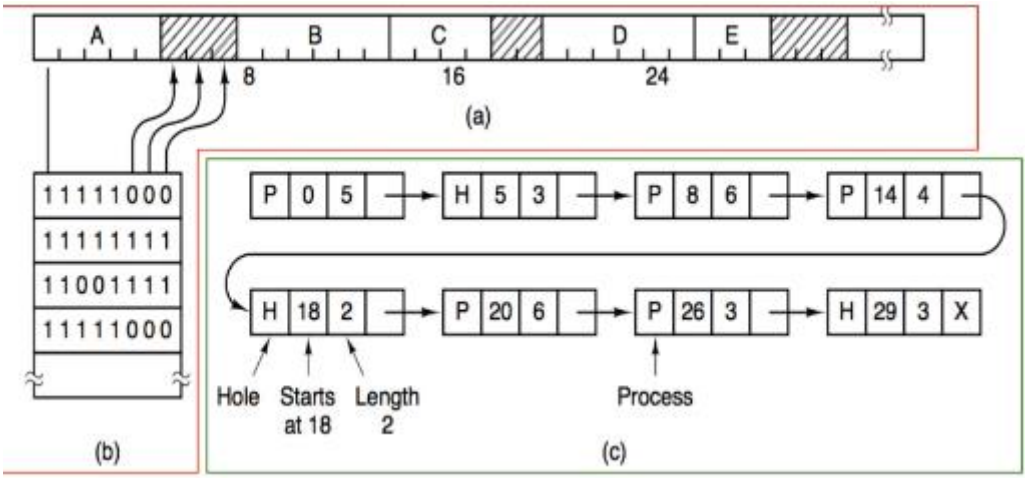
L'idée est d'associer à la mémoire un tableau binaire. Les zones occupées dans la mémoire principale auront 1 dans leur zone du tableau binaire et 0 sinon. Cela se fait à travers les correspondances par mots. Le parcours du tableau se fait à partir du début.

L'inconvénient est que si la mémoire est très grande, le tableau binaire sera très grand, donc peu efficace. De ce fait, il faudra trouver une nouvelle méthode.

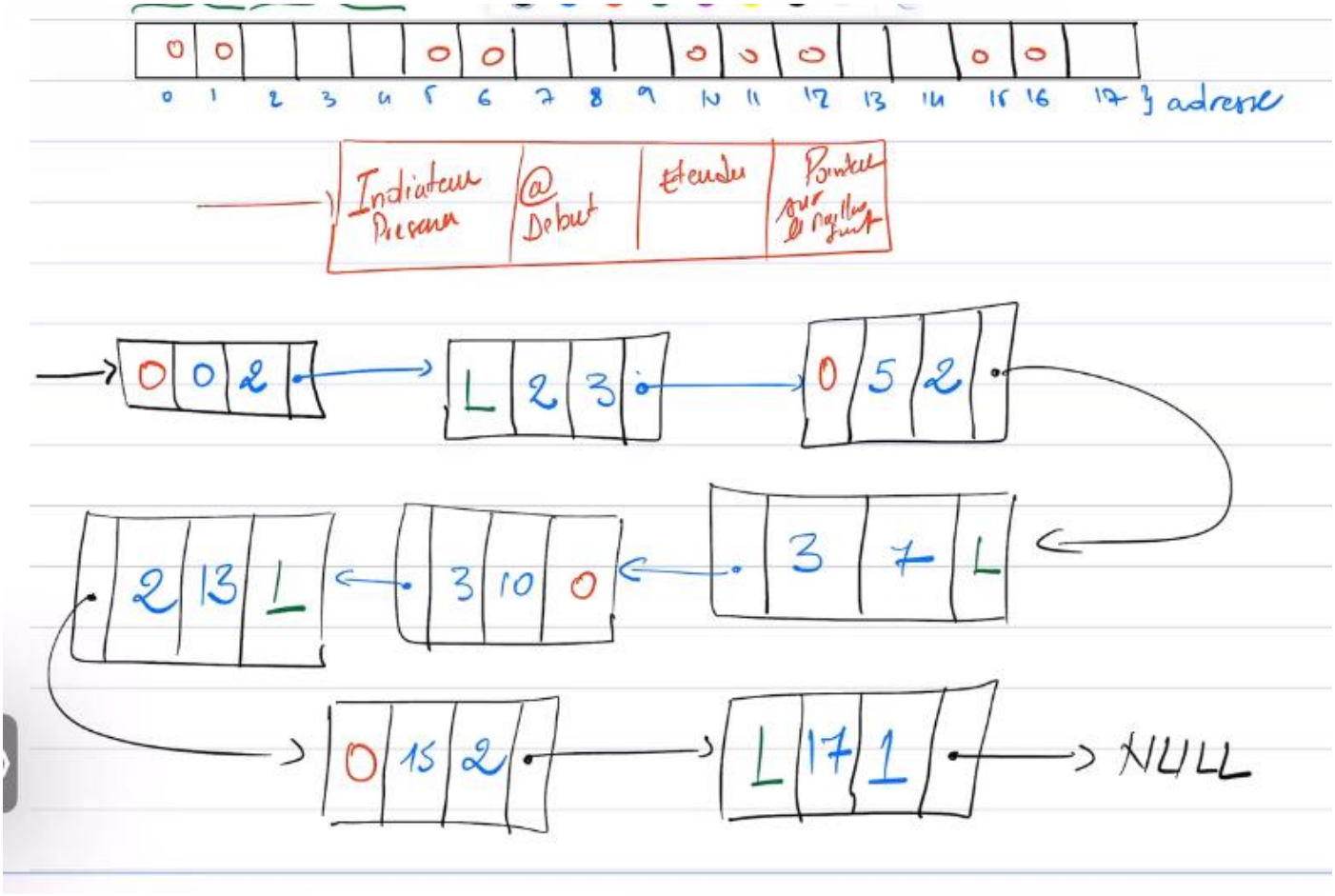
b- GESTION PAR LISTE CHAINEES

- Une autre solution consiste à chaîner les segments libres et occupés
- La figure (c) montre un tel chaînage, les segments occupés par un processus sont marqués (P) les libres sont marqués (H)
- La liste est triée sur les adresses, ce qui facilite la mise à jour.

Une chaine peut être simplement ou doublement chainé. Chaque maillon de la chaîne est constitué de 4 éléments. Le 1er élément est un **indicateur de présence** qui indique si la zone correspondante est occupée (1) ou non (0). Le 2e élément est l'**adresse de début**. Le 3e élément est l'**étendue**. Le 4e élément est le **pointeur** vers le maillon suivant.



Exemple :



Si deux maillons consécutifs ont le même état libre, il faudra les fusionner. Pour cela, il se présente 3 cas :

- 1er cas : On a un espace libre entre 2 espaces occupés.
- 2e cas : On a 2 espaces consécutifs libres qui seront fusionnés
- 3e cas : Le processus avait un espace libre avant et après, dès qu'il termine, ces espaces vont fusionner

Comment faire maintenant pour allouer l'espace libre lorsqu'un processus arrive ?

Nous avons 4 techniques d'allocation de mémoire :

- **ALGORITHME DE PREMIERE ZONE LIBRE**

Le principe est simple : la mémoire est parcourue à partir du début. Dès qu'il atteint le premier espace libre **suffisamment important** pour accueillir le processus, il effectue l'allocation.

Si l'espace libre est égale à la taille requise par le processus, il n'y a pas de fragmentation ; si l'espace libre est supérieure à la taille requise par le processus, la taille du nouvel espace libre sera donc : **Taille libre initiale – Taille du processus**.

- **ALGORITHME DE LA ZONE LIBRE SUIVANTE**

Il s'agit du même principe que l'algorithme de première zone libre ; néanmoins que la recherche commence à partir de la **dernière zone à être allouée**.

- **ALGORITHME DU MEILLEUR AJUSTEMENT**

Il permet d'éviter les problèmes des 2 algorithmes précédents. Le principe est simple : on considère les espaces libres pouvant accueillir le processus, celui ayant la plus petite taille est alloué est au processus.

Un inconvénient est que cela prend trop de temps car il faut faire un parcours intégral de la mémoire pour avoir tous les espaces mémoires candidats. Un autre en est qu'il peut se produire une fragmentation très mauvaise de la mémoire vu qu'il peut y avoir un espace libre qui ne sera jamais alloué.

- **ALGORITHME DU PLUS GRAND RESIDU**

Le principe est le même que celui de l'algorithme du meilleur ajustement à la différence que l'espace ayant la plus grande taille est celui qui sera alloué au processus. Cette méthode ne donne pas non plus de meilleurs résultats que l'algorithme précédent.

IV- MÉMOIRE VIRTUELLE

La mémoire principale est toujours de petite taille par rapport à la mémoire secondaire. De ce fait, un processus n'aura qu'une petite portion de la mémoire principale comme espace d'exécution. Or il peut y avoir des applications qui sont très gourmandes en termes de mémoire.

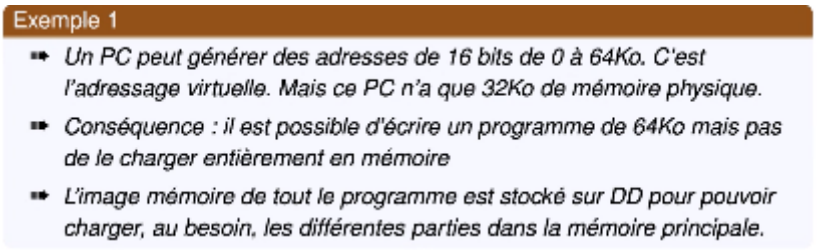
L'idée qui en a découlé est de doter chaque processus d'une mémoire virtuelle très grande par rapport à la portion de la mémoire qui lui a été allouée. Il s'en suivra naturellement un **espace d'adressage virtuel** où les instructions du programmeur y seront placés.

Grâce au swapping il y aura donc conversion entre l'espace d'adressage physique (qui manipule les adresses physiques) et l'espace d'adressage virtuel (qui manipule les adresses virtuelles.

La mémoire virtuelle est allouée aux processus pour ne pas contraindre les développeurs à se restreindre à un seul espace de stockage minimal. Cette mémoire virtuelle est donc située dans la mémoire secondaire.

1- Approche par la mémoire virtuelle

Chaque processus possède un espace d'adressage virtuel. Ce dernier est découpé en **pages**. Ces pages se situent soit en mémoire physique (RAM) soit dans la mémoire secondaire. Prenons l'exemple suivant :



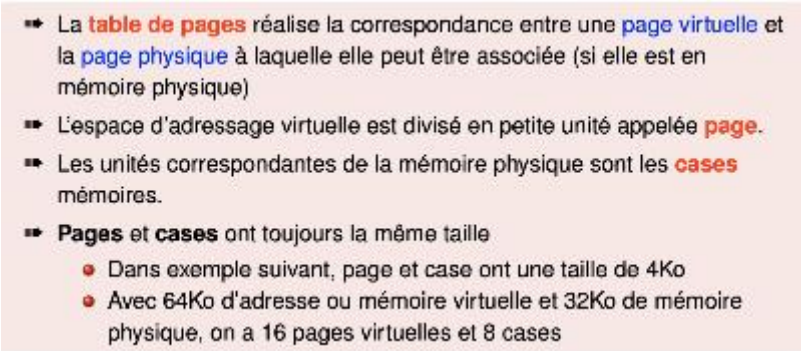
Ici, on n'a pas assez d'espace mémoire physique par rapport à l'espace d'adressage virtuel. Or le premier est découpé en **cases** et le second est découpé en **pages**. Ce découpage est pour que la taille des pages (Tp) soit égale à la taille des cases (Tc).

Ici on aura Tp = Tc = 4 ko.

Nombre de pages : Np = Teav /Tp = 64/4 = 16 pages

Nombre de cases : Nc = Teap/Tc = 32/4 = 8 cases

Tout cela fait partie de la conception de la **table de pages**. Cette table a pour rôle de faire la correspondance entre les cases et les pages. Elle permet également de calculer l'adresse d'une page à partir de l'adresse d'une case et inversement.



Les 2 espaces d'adressage contiennent une colonne qui contient les numéros de pages ou de cases partant de 0 à n – 1.

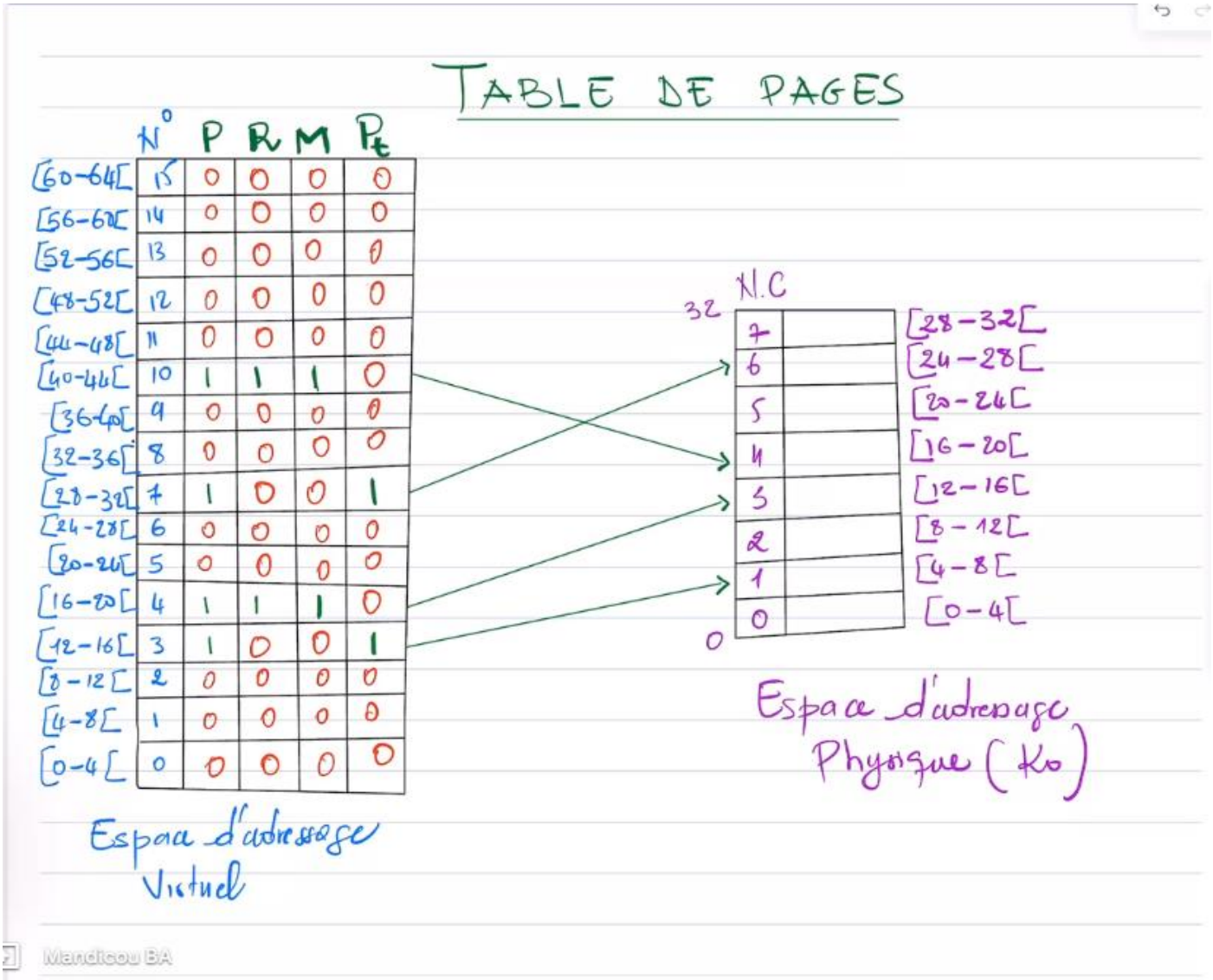
Au niveau de l'EAV, la 2e colonne donne les **bits de présence** de chaque page (1 si la page est présente en mémoire, 0 sinon).

La 3e colonne montre les **bits de référence** qui montrent quel(s) processus concerne une page **présente en mémoire**. Une page non présente en mémoire ne peut pas être référencée. De plus une page présente en mémoire **peut ne pas être référencée**.

La 4e colonne contient les **bits de modification** qui indique si une page **présente en mémoire** a été modifiée. Naturellement, les pages non présentes en mémoire ne peuvent pas être modifiées. En outre, les pages **non référencées** ne peuvent pas être modifiées. Donc le bit de modification est à 1 si la page **est présente en mémoire** et **est référencée**.

La 5e colonne contient les **bits de protection** qui montre si une page est modifiable ou non. Il faut nécessairement que la page soit présente en mémoire. Si le bit de modification est à 1 alors celui de protection est à 0. Ainsi est obtenue la table de pages.

Exemple :



La table de pages permet de calculer l'adresse virtuelle AV (ou adresse logique AL) à partir de l'adresse physique AP (ou adresse réelle AR) et inversement. Elle est donc une table de correspondance (TC).

Nous avons la table des pages, les adresses (logique/virtuelle et physique/réelle), la taille des pages et cases (qui sont toujours égales).

Soit AL, comment calculer AP ?

Notons NP le numéro de page et NC le numéro de case.

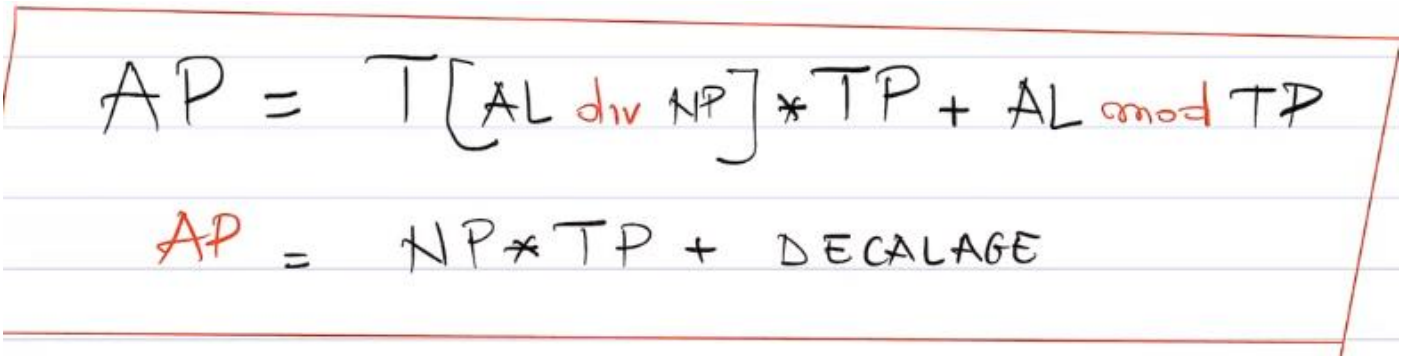
Premièrement, nous récupérons le numéro de page. Pour calculer le numéro de page, nous effectuons la **division entière** entre l'adresse logique et la taille de page. A partir de là nous connaitrons le numéro de case correspondant si la page est présente en mémoire.

Deuxièmement, nous calculons le décalage. Pour cela, nous trouvons le **reste de la division** entre l'adresse logique et la taille de page.

Pour une page nous pouvons connaitre le nombre d'adresses qu'elle contient. Le décalage qu'on aura trouvé au niveau de la page **est la même** que celui au niveau de la case.

L'adresse physique sera déterminée comme suit :

AP = T[AL **div** NP] * TP + AL **mod** TP = NP * TP + DECALAGE



Exemple :

Soit l'adresse A1 = 30560 (Adresse virtuelle). Calculer l'adresse physique correspondante.

Réponse :

On sait que TP = 4Ko (pour l'exemple du haut).

NP = A1 div Tp = 30560 div 4096 = 7

Décalage : D = A1 mod Tp = 30560 mod 4096 = 1888

A1 € [28ko – 32ko[, Donc D = 28k + 1888.

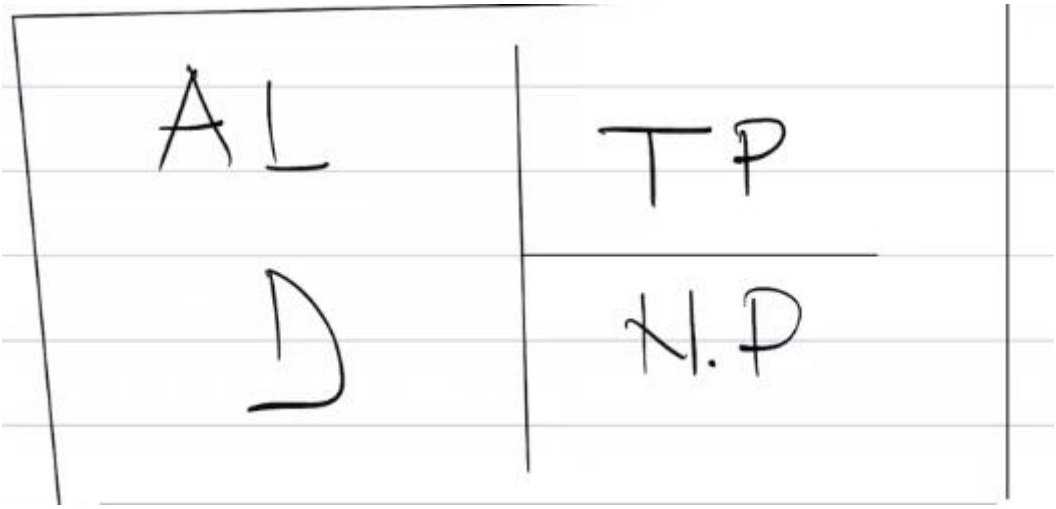
A1 = 4096 x 7 + 1888

TC[7] ➔ 6

AP(A1) = 6 x 4096 + 1888

AP(A1) = 26464

A1 = 26464 (Adresse physique) CQFD



AL = TP x NP + D (D : Décalage)

NP ➔ NC =➔ AP = NC x TP + D

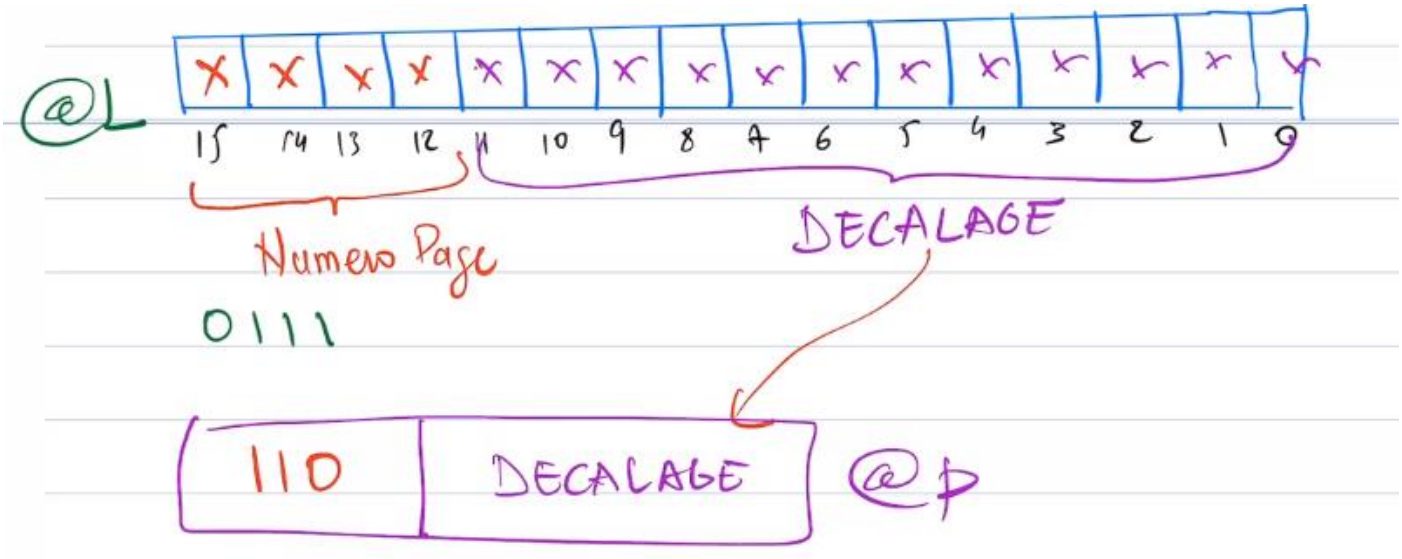
Le calcul peut aussi se faire en binaire.

TP = 4ko = 4096 = 2^x => X = 12

Décalage est codé sur 12 bits
@ codées sur 16 bits

Nombre de page = E.A.V / TP = 64ko/4ko = 16

$16 = 2^y$ => y = 4 => Pages sont codées sur 4 bits
12 + 4 = 16 bits



NB : Lorsque le processeur veut exécuter une instruction présente dans une page et que celle-ci n'est pas présente en mémoire, il se produira un **défaut de page**. Cela entrainera le fait qu'il y aura un remplacement de page. Il est recommandé de remplacer uniquement les pages non référencées sinon il sera nécessaire d'avoir une sauvegarde sous disque qui va prendre en espace mémoire, ce qui sera problématique du point de vue de l'efficacité. Les remplacements de page se font à travers les **algorithmes de remplacement de page**.

Un défaut de page n'est valable que si le processeur souhaite exécuter l'instruction. Un défaut de page est causé par un déroutement. Pour pouvoir exécuter ça, il faudra effectuer le swapping. Si les cases ne sont pas toutes occupées, cette page sera affectée à une case, sinon il faudra appliquer les algorithmes de remplacement de page.

V- ALGORITHME DE REMPLACEMENT DE PAGES

1- Remplacement optimal de pages

Le principe est de charger la page qui mettra le plus de temps à être référencée, ce qui permettra d'éviter de gaspiller des ressources et de l'énergie pour charger et décharger une page mettant peu de temps à être référencée. Cela signifie donc qu'il faudra connaître les dates de référencement à l'avance : ce qui est pratiquement **impossible** ou **irréalisable**.

2- Remplacement de la page non récemment utilisée

Cet algorithme présente deux approches :

- On décharge une page **non référencée** car étant celle qui n'a pas été utilisée récemment.
- Si toutes les pages sont référencées, on va décharger dans ce cas une page **non modifiée** car ne nécessitant pas d'accéder au disque pour effectuer une sauvegarde

Si aucun des deux cas n'est possible, on retire une page quelconque. Pour améliorer cette solution, périodiquement ; le bit de référencement de chaque page est remis à 0.

3- Algorithme FIFO (Premier entré, premier sorti)

Comme son nom l'indique, l'idée est de décharger la première page à être référencée. Cela présente néanmoins un danger car souvent ces pages contiennent les données nécessaires au déroulement du programme.

4- Remplacement avec seconde chance

Il est identique à FIFO ; sauf que si la première page est toujours référencée, on lui laisse un cycle de présence et est placée en queue de file d'attente, le temps que les autres soient vérifiées. Ce qui signifie que la première page n'est pas nécessairement celle à être déchargée.

5- Remplacement avec horloge

Il est identique au précédent, sauf que la queue est circulaire.

6- Remplacement de la page la moins récemment utilisée

Il est identique au 2^{e} algorithme sauf que si toutes les pages sont référencées, on décharge qui a mis le plus de temps sans être utilisée.

VI- SEGMENTATION

Avec le système de pagination, toutes les pages ont la même taille, ce qui n'est pas toujours utile.

Avec la segmentation, on crée des segments de tailles différentes dont la taille peut varier au cours de l'exécution. Ce qui inclut aussi des systèmes d'adressage différents. De ce fait, pour un processus, le segment de code peut être partagé. Le même processus peut avoir plusieurs segments de code. La segmentation est donc plus difficile à implémenter.

Dans la segmentation, l'espace d'adressage est géré par le noyau et non par le développeur. De plus on a plusieurs espaces d'adressage, les segments sont également protégées et chaque segment a sa propre table de pages (avec des tailles de pages possiblement différentes).