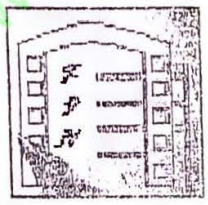


Université Mohamed Premier
Faculté Pluridisciplinaire
Département de Mathématiques & Informatique
Nador

Support de Cours

Systèmes d'exploitation II

Professeur : Fatima EL HAOUSSI

Filière: SMI-S4
Année Universitaire

Sommaire

Chapitre 1 Les processus

1. Aspects généraux des processus

Un processus est un programme qui s'exécute et possède des compteurs de son exécution, des registres, des variables et une pile d'exécution. Son exécution est, en général, une alternance de calculs effectués par le processeur et de requêtes d'Entrée/Sortie effectuées par les périphériques.

1.1. Etats d'un processus

Lorsqu'un processus s'exécute, il change d'état, comme on peut le voir sur la figure 1.1. Il peut se trouver alors dans l'un des trois états principaux:

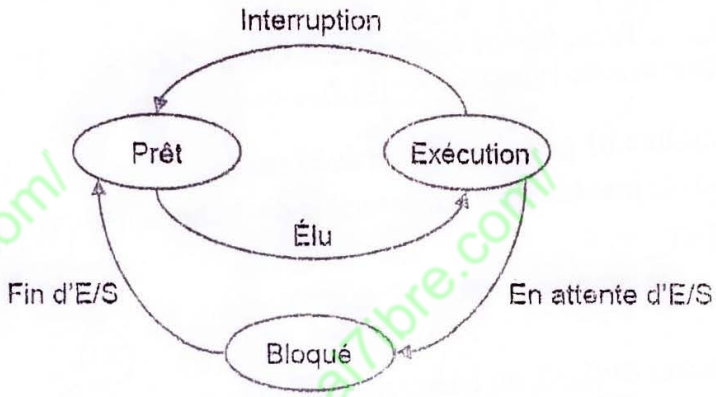


FIG. 1.1 – Les états principaux d'un processus.

Elu : s'il est en cours d'exécution sur le processeur.
Bloqué : s'il est en attente d'un événement externe (bloc disque, frappe clavier,...).
Prêt : suspendu provisoirement pour permettre l'exécution d'un autre processus.

Initialement, un processus est à l'état **prêt**. Il passe à l'état **exécution**, lorsque le processeur entame son exécution. Un processus passe de l'état exécution à l'état prêt, lorsqu'il est suspendu provisoirement pour permettre l'exécution d'un autre processus. Il passe de l'état exécution à l'état bloqué, si le processus ne peut plus poursuivre son exécution (demande d'une E/S). Il se met alors en attente d'un événement (fin de l'E/S). Lorsque l'événement survient, il redevient prêt.

1.2. Simultanéité

On appelle **simultanéité** l'activation de plusieurs processus au même temps.
Si le nombre de processeurs est au moins égal au nombre de processus, on parle de **simultanéité totale**, sinon de **pseudo-simultanéité**.

La simultanéité est obtenue par commutation temporelle d'un processus à l'autre sur le processeur. Si les basculements sont suffisamment fréquents, l'utilisateur a l'illusion d'une simultanéité totale.

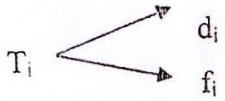
2. Modèles de représentation des processus

2.1. Processus séquentiels

On appelle tâche une unité élémentaire de traitement ayant une cohérence logique. Si l'exécution du processus P est constituée de l'exécution **séquentielle** des tâches T_1, T_2,..., T_n, on écrit :

$$P = T_1 T_2 ... T_n$$

On découpe encore :



d : début, f : fin

Suite de tâches T_1, T_2, ..., T_n est associé à une suite $d_1f_1d_2f_2....d_nf_n$
$d_1f_1d_2f_2....d_nf_n$ est appelé mot de l'alphabet $A=\{d_1,f_1,...,d_n,f_n\}$

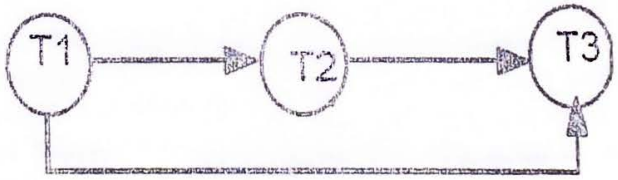
2.2. Système de tâches et graphes de précédences

On définit une relation de précédence sur E ensemble des tâches notée < comme :

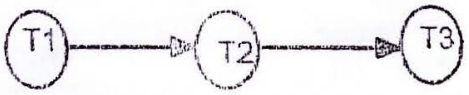
i) $\forall T \in E$ on n'a pas $T<T$
ii) $\forall T, T' \in E \times E$ on n'a pas simultanément $T<T'$ et $T'<T$
iii) < est transitive.

Soit un système de tâches S=(E,<), on interprète la relation < comme : T<T' <=> terminaison de T nécessairement avant début de T'. Ainsi, si on a ni T<T' ni T'<T on dira que les tâches sont exécutables en parallèle.

Une relation de précédence peut être représentée par un graphe orienté. Par exemple, la chaîne de tâches S = ((T1, T2, T3), (Ti<Tj pour i inférieur à j)) a pour graphe :



Qu'on peut simplifier en représentant le graphe de la plus petite relation qui a même fermeture transitive que la relation de précédence : c'est le graphe de précédence :

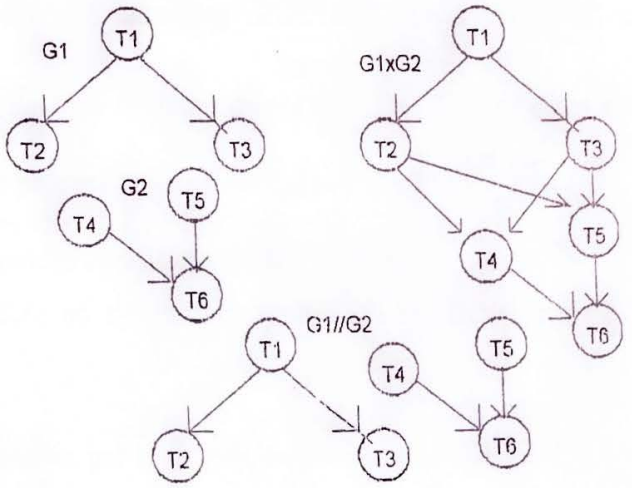


On munit l'ensemble des graphes de précédence de deux opérations :

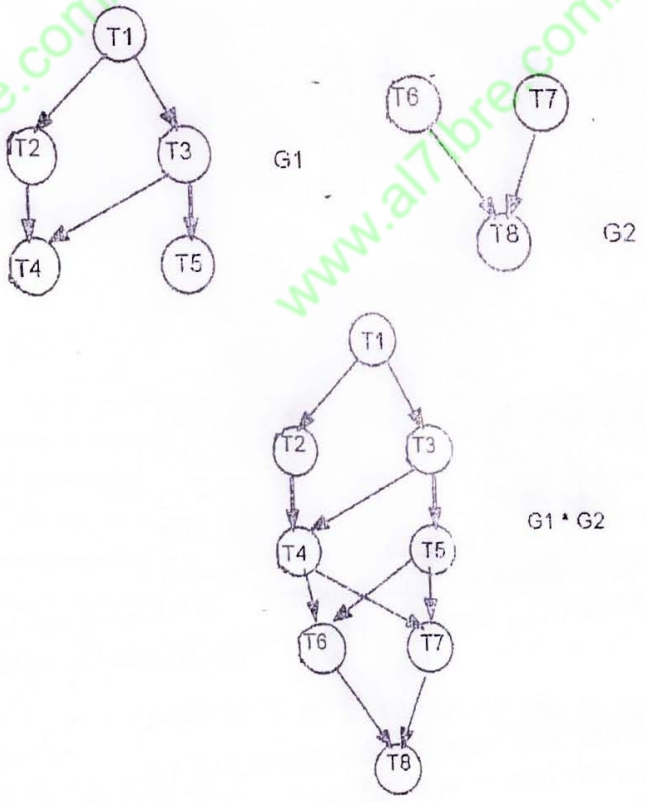
- **la composition parallèle:** G1 et G2 étant deux graphes de précédence correspondant à des ensembles de tâches disjoints, G1 // G2 est l'union de G1 et de G2.
- **le produit:** G1 * G2 reprend les contraintes de G1 et de G2 avec en plus la contrainte qu'aucune tâche de G2 ne peut être initialisée avant que toutes les tâches de G1 ne soient achevées.

Exemple 1:

Soient deux systèmes de tâche G1 et G2. On définit le produit de G1 et de G2 en ajoutant au graphe des arêtes joignant chaque sommet terminal de G1 à chaque sommet initial de G2. Exemple ci-dessous ainsi que la mise en parallèle.



Exemple 2:



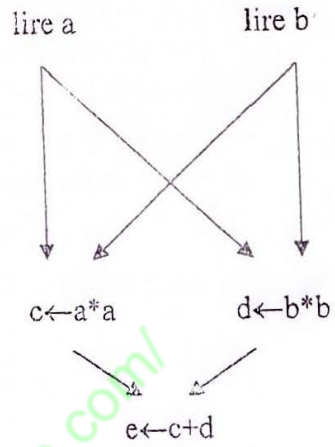
3. Interactions de processus

3.1. Parallélisme = Indéterminisme

La mise en parallèle s'effectue par la structure algorithmique :

parbegin
.....
parend

Exemple 1:



soit début

```
parbegin
lire a
lire b
parend
parbegin
c← a * a
d← b * b
parend
e←c + d
fin
```

Notion d'état:

La mémoire centrale peut être vue comme une suite de cellules Ci: M=(C1,C2,...,Cm).
L'état k du système Sk est Sk = [C1(k), ..., Cm(k)] après l'événement ak du comportement w=a1a2...al.
k=0 étant l'état initial

Chaque tâche utilise certaines cellules soit pour les consulter soit pour les modifier. On caractérise les tâches par leur domaine de lecture et d'écriture :

Tâche	Domaine de lecture	Domaine d'écriture
Ti	Li={C'1, ..., C'p}	Ei={C"1, ...,C"q}

A une tâche T, on associe une fonction F_T de L dans E.

Définition 1 : un système de tâches S est déterminé si pour tous comportements w et w' et pour toute cellule C de M, on a :

$$V(C, w) = V(C, w')$$

Tout système séquentiel est déterminé.
Il existe une relation entre le caractère déterminé d'un système et une propriété de non-interférence de tâches.

Définition 2 : Soit S=(E,<) un système de tâches. T et T' sont dites **non interférentes** vis à vis de S si :

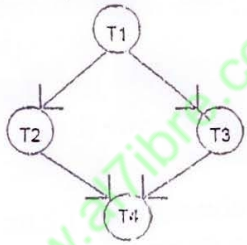
- ou bien T est un prédécesseur ou successeur de T'
- ou bien $L_T \cap E_{T'} = L_{T'} \cap E_T = E_T \cap E_{T'} = \varnothing$

Ces conditions sont appelées conditions de **Bernstein**.

Théorème : Soit S=(E,<) un système de tâches. Si les tâches sont 2 à 2 non **interférentes** alors il est **déterminé**.

Exemple 2 :

Soient 4 tâches caractérisées par le graphe de précédence ci-contre.



T1 : N←0
T2 : N←N+1
T3 : N←N+1
T4 : afficher N

Tâche	Domaine de lecture	Domaine d'écriture
T1	L1=∅	E1={N}
T2	L2={N}	E2={N}
T3	L3={N}	E3={N}
T4	L4={N}	E4=∅

3.2. Parallélisme maximal

Définition 1 : deux systèmes S et S' construits sur le même ensemble de tâches sont équivalents si :

- S et S' sont déterminés
- Et pour tout comportement w de S, pour tout comportement w' de S', pour toute cellule C de M, on a : V (C, w) = V (C, w')

Autrement dit, la suite des valeurs écrites dans toute cellule par tout comportement de l'un ou l'autre système est unique.

Déf... 2 : un système S est de parallélisme maximal si :

- ...st déterminé
- ...on graphe de précédence G vérifie : la suppression de tout arc(T, T') entraîne ...terférence des tâches T et T'.

Théo... : S = (E, <) étant un système déterminé, il existe un unique système S' de para... maximal équivalent à S. S'est tel que : S' = (E, <') avec <' fermeture transitive de la re...

R = { ... | T < T' et ($L_T \cap E_{T'} \neq \emptyset$ ou $L_{T'} \cap E_T \neq \emptyset$ ou $E_T \cap E_{T'} \neq \emptyset$) et $E_T \neq \emptyset$ et $E_{T'} \neq \emptyset$}

Algo... : il découle du théorème :

- ...struire le graphe de R
- ...miner tous les arcs (T, T') redondants, c'est à dire tels qu'il existe un chemin de T à T' ...ontenant plus d'un arc

4. L... ...tion des processus

Dans ... partie, nous allons présenter quelques notions théoriques concernant les processus, puis l... ...q modes d'exécution d'une commande sous Linux, ainsi que les commandes *ps* et *kill* qui permettent de gérer les processus. Enfin nous présentons le job control.

4.1 Notions théoriques sur les processus

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

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

Implémentation des processus : Pour implémenter les processus, le système d'exploitation utilise un tableau de structure, appelé **table des processus**. Cette dernière comprend une entrée par processus, allouée dynamiquement, correspondant au processus associé à ce programme : c'est le **bloc de contrôle du processus** (*Process Control Block*, souvent abrégé PCB). Ce bloc contient, entres autres, les informations suivantes :

- le PID, le PPID, l'UID et le GID du processus ;
- l'état du processus ;
- les fichiers ouverts par le processus ;
- le répertoire courant du processus ;
- le terminal attaché au processus ;
- les signaux reçus par le processus ;
- le contexte processeur et mémoire du processus (c'est-à-dire l'état des registres et des données mémoires du processus).

Grâce à ces informations stockées dans la table des processus, un processus bloqué pourra redémarrer ultérieurement avec les mêmes caractéristiques.

Classification des processus

Il est possible de distinguer deux types de processus : les processus système et les processus utilisateurs.

a) Les processus système (daemons)

Ces processus ne sont sous le contrôle d'aucun terminal et ont comme propriétaire l'administrateur du système ou un uid d'administration. Ils assurent des tâches d'ordre général, parfois disponibles à tous les utilisateurs du système. Ils ne sont d'habitude stoppés qu'à l'arrêt du système d'exploitation. Les plus courants sont :

- *init* : initialise un processus par terminal connecté sur la machine, permettant la connexion des utilisateurs. Ce processus a le numéro 1. C'est le processus parent de tous les interpréteurs de commandes créés par la connexion d'un utilisateur.
- *crond* : permet l'exécution d'un programme en mode cyclique.
- *xinetd* : super démon internet chargé de créer les processus serveurs réseau sur requêtes des clients.

b) Les processus utilisateurs

Ils correspondent à chaque exécution d'un programme par l'utilisateur, le premier d'entre eux étant l'interpréteur de commandes à la connexion. Ces processus appartiennent à l'utilisateur et sont généralement attachés à un terminal.

4.2. Exécution d'une commande

Il existe cinq modes d'exécution d'une commande sous Linux :

- Le mode interactif (foreground),
- Le mode en arrière-plan (background), appelé aussi mode asynchrone,
- Le mode différé,
- Le mode batch,
- Le mode cyclique.

4.2.1. Le mode interactif

En mode interactif, mode le plus fréquemment utilisé, la commande est lancée à partir d'un interpréteur de commandes. Pendant l'exécution de la commande, l'utilisateur ne peut pas utiliser le terminal pour lancer une autre commande.
Pour interrompre l'exécution d'une commande en utilisant la combinaison de touches *<ctrl c>*, ou de la suspendre à l'aide de *<ctrl z>*.

4.2.2. Le mode en arrière-plan

Le lancement de commandes en arrière-plan permet de rendre immédiatement le contrôle à l'utilisateur. Cette fonctionnalité est intéressante pour des tâches ne nécessitant pas d'interaction entre l'utilisateur et la tâche, comme par exemple la compilation d'un programme. La commande est lancée suivie du caractère ***&***. Son exécution peut être surveillée par les commandes ***ps*** ou ***jobs***.

Exemple

Le fichier *essai* est un exécutable. Son exécution n'interagit pas avec l'utilisateur.
etudiant>**./ essai &**
L'utilisateur a la main. Il peut continuer à travailler tout en surveillant régulièrement l'exécution de la commande ***./essai***.

4.2.3. Le mode différé

L'exécution différée d'une commande est réalisée à l'aide de la commande *at* qui permet de déclencher l'exécution d'une commande à une date fixée.

Exemple

etudiant> **at 20:05 20/02/15 <commande**

Demande la lancement du contenu de *commande* le 20/02/15 à 20 h 05.

4.2.4. Le mode batch

Le batch permet de placer une commande dans une file d'attente. Le système exécutera toujours la commande placée en tête dans la file d'attente. Ainsi toutes les commandes lancées par *batch* seront exécutées séquentiellement quel que soit l'utilisateur qui a mis la commande dans la file d'attente. La gestion des sorties standard est similaire à celle de la commande *at*.

4.2.5. Le mode cyclique

L'exécution cyclique d'une tâche est réalisée à l'aide de la commande *crontab*. Le processus *cron* (daemon) consulte un fichier dans lequel sont définies les commandes à exécuter à date fixée. L'autorisation d'utilisation de *crontab* pour un utilisateur est identique à la méthode définie pour l'autorisation d'utilisation de la commande *at*. Les fichiers vérifiés sont */etc/cron.allow* puis */etc/cron.deny*.

4.3. La commande ps

La commande **ps** options permet d'obtenir des renseignements sur l'état des processus en cours. Parmi les options les plus intéressantes sont :

-e affiche des renseignements sur tous les processus en cours,

-C affiche des renseignements sur tous les processus d'un certain nom : *ps C bash*

-f génère, pour chaque processus, le nom de l'utilisateur (UID), le numéro du processus (PID), le numéro du processus père (PPID), l'heure de lancement du processus (STIME), le nom du terminal (TTY) et le temps d'exécution du processus (TIME).

Pour afficher tous les processus en cours d'exécution, on peut utiliser l'option (a : processus de tous les utilisateurs ; u : affichage détaillé ; x : démons), dans le résultat qui s'affiche, vous pouvez voir la liste de tous vos processus en cours d'exécution.

La première colonne **USER** correspond à l'utilisateur qui a lancé le processus.

La deuxième colonne **PID** indique le numéro de PID du processus.

La huitième colonne **STAT** correspond à l'état du processus.

La neuvième colonne **START** correspond à l'heure du lancement du processus.

Enfin, la dernière colonne **COMMAND** correspond au chemin complet de la commande lancée par l'utilisateur (car même si vous lancez un exécutable en mode graphique, vous avez une commande qui s'exécute).

4.4. La commande nice

Les processus tournent avec un certain degré de priorité, un processus plus prioritaire aura tendance à s'accaparer plus souvent les ressources du système pour arriver le plus vite possible au terme de son exécution. C'est le rôle du système d'exploitation de gérer ces priorités.
Nous disposons de la commande **nice** pour modifier la priorité d'un processus. La syntaxe est la suivante :
nice -valeur commande

Plus le nombre est grand, plus la priorité est faible. Par exemple une valeur de 0 donne la priorité la plus haute, et une valeur de 20 donne la priorité la plus faible.

Par exemple : etudiant>nice -5 ps -ef

4.5. La commande kill

Cette commande **kill** signal PID sert essentiellement à arrêter des processus en arrière-plan (background). En effet, pour arrêter le ou les processus liés à la commande en cours d'exécution, il est beaucoup plus simple de taper <ctrl-c>. Si nous voulons arrêter un processus, nous devons connaître son **PID** (commande ps).
L'option ***signal*** correspond au signal envoyé au processus. Un certain nombre d'événements (erreur, *<ctrl c>*,...) peut être signalé à un processus à l'aide d'un signal. Ce mécanisme permet la communication interprocessus, notamment entre le système d'exploitation et le processus. Les principaux signaux sont :

SIGHUP (SIGnal Hang UP : fin du shell)
SIGINT (SIGnal INTerrupt : interruption du programme)
SIGKILL (SIGnal KILL : tuer le processus)
SIGTERM (SIGnal TERMinate : terminaison douce)
SIGQUIT (SIGnal QUIT : terminaison brutale)
SIGSTOP (SIGnal STOP : stopper le processus)

4.6. Le job control

Le **job** est une ligne de commandes shell. Il est composé d'un ou de plusieurs processus dans le cas d'utilisation du tube. Chaque job est numéroté de 1 à N par le shell. Ce numéro est interne au shell. Un job peut se trouver dans trois états :

- **avant-plan** ("foreground"). Le job s'exécute et vous n'avez pas la main en shell.
- **arrière-plan** ("background"). Le job s'exécute et vous avez la main en shell.
- **suspendu** ("suspended"). Le job est en attente, il ne s'exécute pas.

Le passage d'un état du job à un autre est présenté sur la figure 4.1.

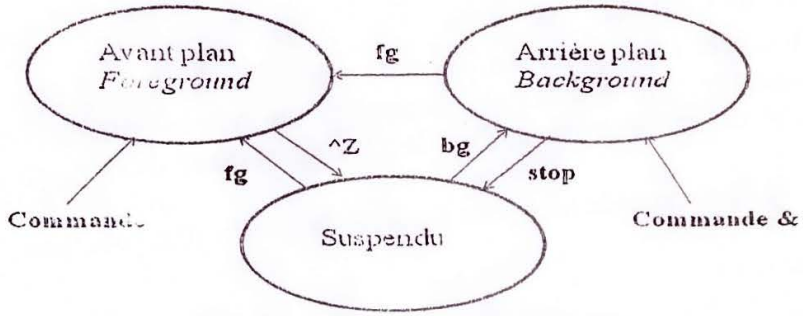


FIG. 4.1 – Différents états d'un job.

4.6.1. Job en arrière-plan

La création d'un job en arrière-plan est réalisée par l'utilisation du lancement d'une commande en "background". La commande *jobs -l* permet de lister les jobs en cours.

Exemple:
```
etudiant> sleep 60&
[1] 2736
etudiant> sleep 50&
[2] 2957
etudiant> sleep 40&
[3] 3100
etudiant> jobs -l
[1]   2736  Running sleep  60&
[2] - 2957  Running sleep  50&
[3] + 3100  Running sleep  40&
```

Dans l'exemple précédent, le caractère + indique le job le plus récent, le caractère – indique le deuxième plus récent.

4.6.2. Job suspendu

Une commande peut être suspendue lorsqu'elle est en avant-plan par la touche *<ctrl Z>*. Lorsqu'elle est en arrière-plan la commande *kill –SIGSTOP %numero job* ou *kill -STOP %numero job* permet de suspendre la commande associée au numéro de job *numero job*.
La commande *bg %numero job* permet de basculer un job suspendu en job en arrière-plan.

Exemple : Nous utilisons ici la commande stty qui permet de visualiser et de modifier les caractéristiques de la liaison associé à l'entrée standard.
Syntaxe

stty [options] [arguments]

Option
-a affiche toutes les caractéristiques de la liaison
susp définit le caractère de suspension de processus, par défaut <ctrl-z>.

```
etudiant> stty susp '^Z'
etudiant>sleep 200&
etudiant> sleep 100
```

<ctrl Z>
[1] 2655 Stopped
etudiant> **jobs -l**
[1] 2655 Stopped sleep 100
[2] -2957 Running sleep 200&
etudiant> **kill –SIGSTOP %2**
[2] Stopped sleep 200
etudaint> **bg %2**

4.6.3. Job en avant-plan

Toute commande lancée interactivement en shell est en avant-plan. Un job en arrière plan ou suspendu peut être mis en avant-plan par la commande *fg %numero job*.

Exemple:
etudiant> **jobs -l**
[1] 2655 Stopped sleep 100
[2] -2957 Running sleep 200&
etudiant> **fg %1**
§ *Attente de quelques secondes,*
§ *la commande sleep 100 est en avant plan*

4.6.4. La commande kill et le job control

L'un des grands avantages du "job control" est de faciliter grandement les possibilités pour tuer un processus. En effet tuer un job est plus pratique que tuer les processus qui le constituent : il est inutile de manipuler des numéros de processus, surtout dans le cas où le job est constitué de plusieurs processus (tube, commandes groupées).

Exemple:
etudiant> **(sleep 55; sleep 66)&**
[1] 11877
etudiant> **jobs -l**
[1]+ 11877 Running (sleep 55; sleep 66) &
etudiant> **kill %1**
etudiant> **jobs -l**
[1]+ 11877 Terminated (sleep 55; sleep 66)

Chapitre 2 Ordonnancement

1. Définitions

L'ordonnanceur définit l'ordre dans lequel les processus prêts utilisent l'UC (en acquièrent la ressource) et la durée d'utilisation, en utilisant un algorithme d'ordonnancement. Nous pouvons distinguer 5 critères pour effectuer un bon ordonnancement:

Efficacité : le processeur doit travailler à 100 % du temps.
Rendement : l'ordinateur doit exécuter le maximum de programmes en un temps donné.
Temps de réponse: le temps moyen pour répondre aux entrées de les utilisateurs (systèmes interactifs).
Temps d'exécution : chaque programme doit s'exécuter le plus vite possible.
Equité : chaque processus reçoit sa part du temps processeur.

Le temps de séjour pour chaque processus est obtenu soustrayant le temps d'entrée du processus du temps de terminaison.
Le temps d'attente est calculé soustrayant le temps d'exécution du temps de séjour.

$$Temps\ moyen\ d'attente = \frac{\sum Temps\ attente}{nbre\ de\ processus}$$

$$Temps\ moyen\ de\ séjour = \frac{\sum Temps\ de\ séjour}{nbre\ de\ processus}$$

Il existe deux familles d'algorithmes :

Sans réquisition ou non préemptif : le choix d'un nouveau processus ne se fait que sur blocage ou terminaison du processus courant.

Avec réquisition ou préemptif: à intervalle régulier, l'ordonnanceur reprend la main et élit un nouveau processus actif.

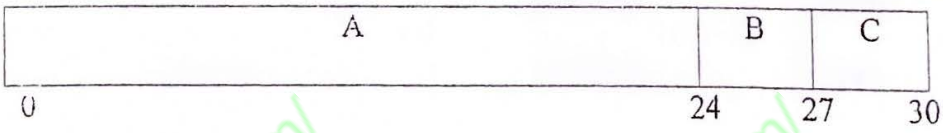
2. Premier arrivé premier servi PAPS (FIFO : First in first out)

L'ordonnanceur (non préemptif), utilise une pile dans laquelle sont stockés dans l'ordre d'arrivée les processus en attente du processeur. En général, le **Premier Arrivé est le Premier Servi**. Cet algorithme est facile à implanter, mais il est loin d'optimiser le temps de traitement moyen.

Exemple :

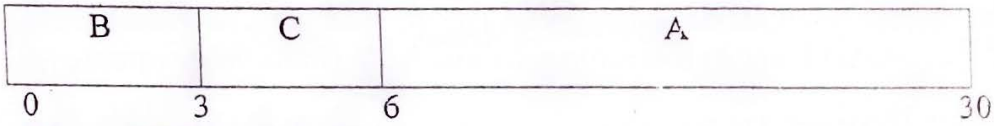
Processus	Temps d'exécution
A	24
B	3
C	3

Ordre d'arrivée : A, B, C
Schéma d'exécution:



Temps d'attente de A est 0
Temps d'attente de B est 24
Temps d'attente de C est 27
Temps moyen d'attente : (0+24+27)/3=17

Ordre d'arrivée : B, C, A
Schéma d'exécution :



Temps d'attente de A est 6
Temps d'attente de B est 0
Temps d'attente de C est 3
Temps moyen d'attente : (0+6+3)/3=3
L'ordre d'arrivée B, C, A est meilleur que dans le cas précédent

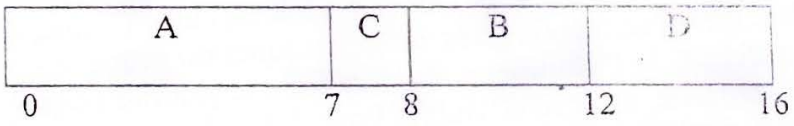
3. Plus court temps d'exécution PCTE (SJF : Shortest First job)

L'ordonnancement basé sur l'algorithme du travail le plus court d'abord (plus petit temps d'exécution) est également non préemptif. A partir d'une estimation du temps nécessaire à l'exécution des processus, le prochain élu est donc le plus court.

Exemple :

Processus	Temps d'arrivage	Temps d'exécution
A	0	7
B	2	4
C	4	1
D	5	4

Schéma d'exécution :



Processus	Temps de séjour
A	7-0=7
C	8-4=4
B	12-2=10
D	16-5=11

Processus	Temps d'attente
A	7-7=0
C	4-1=3
B	10-4=6
D	11-4=7

Le temps moyen d'attente est : (0+3+6+7)/4=4

4. Plus petit temps de séjour (SRT)

L'ordonnancement du plus petit temps de séjour ou **Shortest Remaining Time** est la version préemptive de l'algorithme SJF. 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.

5. Tourniquet (RR : Round Robin)

L'ordonnancement par tourniquet, circulaire ou (Round Robin) est préemptif. Chaque processus dispose d'un quantum de temps pendant lequel il est autorisé à s'exécuter.

Si le processus actif se bloque ou s'achève avant la fin de son quantum, le processeur est immédiatement alloué à un autre processus. Si le quantum s'achève avant la fin du processus, le processeur est alloué au processus suivant dans la liste et le processus précédent se trouve ainsi en queue de liste.

Exemple :

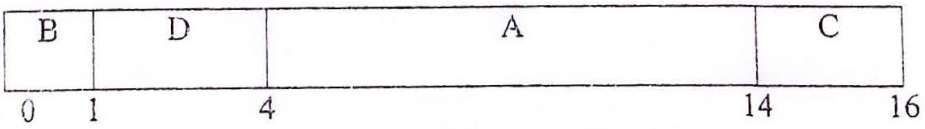
Soit trois processus A, B, C faisant uniquement du calcul. Les temps d'exécution des trois processus sont respectivement 15, 4 et 7 millisecondes. Le quantum est fixé à 3ms.

A	A	A	B	B	B	C	C	C	A	A	A	B	C	C	C	A	A	A	C	A	A	A	A	A
1				5					10					15						20					25

6. Avec priorité

Pour l'ordonnancement préemptif à priorité, on affecte une valeur à chaque processus. Le processus élu est celui qui à la priorité la plus élevée (valeur la plus faible), et en cas d'égalité, on utilise FIFO. Les priorités peuvent être fixées en fonction des caractéristiques du processus (beaucoup d'E/S, utilisation de la mémoire,...), de l'utilisateur, ou même d'une priorité fixée par l'administrateur ou l'utilisateur (commande **nice** sous Unix). Pour éviter le risque de famine des processus de trop faible priorité, on peut augmenter la priorité en fonction du temps d'attente écoulé.

Exemple :
Soit A, B, C et D des processus de temps d'exécution respectifs 10, 1, 2, 3, et de priorité respectives 3, 1, 3, 2.



7. Files multiples

On associe une file d'attente et un algorithme d'ordonnancement à chaque niveau de priorité. Si les priorités évoluent dynamiquement, le SE doit organiser la remontée des tâches.

8. A deux niveaux

La taille de la mémoire centrale de l'ordinateur peut être insuffisante pour contenir tous les processus prêts à être exécutés. Certains sont contraints de résider sur disque.

Un ordonnanceur de bas niveau applique l'un des algorithmes précédents aux processus résidant en mémoire centrale.

Un ordonnanceur de haut niveau retire de la mémoire les processus qui y sont resté assez longtemps et transfère en mémoire des processus résidant sur disque.

10. Cas d'étude « Ordonnancement Unix »

L'ordonnanceur d'UNIX est un ordonnanceur à deux niveaux, à priorité qui ordonnance les processus de même priorité selon l'algorithme du tourniquet.

L'ordonnanceur de bas niveau se charge de sélectionner un processus parmi ceux qui sont prêts et résidents en mémoire. Cette restriction permet d'éviter lors de commutation de contextes qu'il y ait un chargement à partir du disque d'un processus en mémoire (réduction du temps de commutation).

L'ordonnanceur de haut niveau se charge de ramener des processus prêts en mémoire en transférant éventuellement des processus sur disque.

Chapitre 3 : Introduction à la programmation système

1. Qu'est-ce que la programmation système ?

Lorsque l'on dispose d'un système d'exploitation, ce dernier permet de différencier deux types de programmes :
Les **programmes d'application** des utilisateurs. Ces programmes sont réalisés lors de la programmation dite « classique », celle que nous avons faite par exemple sur le langage C.
Les **programmes systèmes** qui permettent le fonctionnement de l'ordinateur. C'est ce type de programme que nous allons créer dans ce cours.

Exemples : L'accès aux fichiers, la gestion des processus, la programmation réseau, les entrées/sorties, la gestion de la mémoire...

Il faut savoir que le langage C, à partir duquel nous programmerons, a été créé spécialement pour la programmation système, plus précisément pour le développement du système d'exploitation... UNIX. Il est donc particulièrement adapté à ce type de programmation.

2. Les bases de la programmation système

Dans cette partie, nous allons aborder quelques termes de vocabulaire indispensables pour la suite du cours.
Les systèmes Unix sont des systèmes d'exploitation qui sont constitués de plusieurs programmes, et chacun d'eux fournit un service au système. Tous les programmes qui fournissent des services similaires sont regroupés dans une **couche logicielle**.
Une couche logicielle qui a accès au matériel informatique s'appelle une couche d'**abstraction matérielle**.

Le **noyau** est une sorte de logiciel d'arrière-plan qui assure les communications entre ces programmes. C'est donc par lui qu'il va nous falloir passer pour avoir accès aux informations du système.
Pour accéder à ces informations, nous allons utiliser des fonctions qui permettent de communiquer avec le noyau. Ces fonctions s'appellent des **appels-systèmes**.

Les **appels-systèmes** constituent l'interface du système d'exploitation et sont les points d'entrées permettant l'exécution d'une fonction du système. Les appels système sont directement appelables depuis un programme. Les commandes permettent elles d'appeler les fonctions du système par l'interpréteur de commande.

3. Outils de développement

Le développement en C sous Unix comme sous la plupart des autres systèmes d'exploitation met en œuvre principalement cinq types d'utilitaires :

- *L'éditeur de texte*, qui est à l'origine de tout le processus de développement applicatif. Il nous permet de créer et de modifier les fichiers source.
- *Le compilateur*, qui permet de passer d'un fichier source à un fichier objet. Cette transformation se fait en réalité en plusieurs étapes grâce à différents composants (préprocesseur C, compilateur, assembleur).

- *L'éditeur de liens*, qui assure le regroupement des fichiers objet provenant des différents modules et les associe avec les bibliothèques utilisées pour l'application. Nous obtenons ici un fichier exécutable.
- *Le débogueur*, qui peut alors permettre l'exécution pas à pas du code, l'examen des variables internes, etc. Pour cela, il a besoin du fichier exécutable et du code source.
- *Utilitaires annexes* travaillant à partir du code source, comme le vérificateur Lint, les outils de documentation automatique, etc.

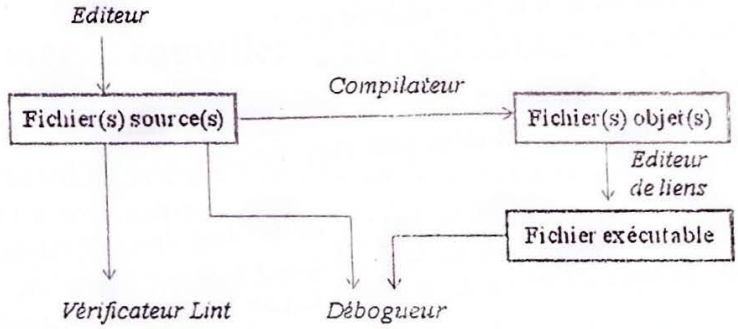


FIG. 3.1- Processus de développement en C

4. Éditeurs de texte

L'éditeur de texte est probablement la fenêtre de l'écran que le développeur regarde le plus. Il passe la majeure partie de son temps à saisir, relire, modifier son code, et il est essentiel de maîtriser parfaitement les commandes de base pour le déplacement, les fonctions de copier-coller et le basculement rapide entre plusieurs fichiers source. Il existe deux champions de l'édition de texte sous Unix, Vi d'une part et Emacs de l'autre.

4.1 Emacs

Emacs est théoriquement un éditeur de texte, mais des possibilités d'extension par l'intermédiaire de scripts Lisp en ont fait une énorme machine capable d'offrir l'essentiel des commandes dont un développeur peut rêver.

4.2 Vi

Vi est beaucoup plus léger, il offre nettement moins de fonctionnalités et de possibilités d'extensions qu'Emacs. Les avantages de **Vi** sont sa disponibilité sur toutes les plates formes Unix et la possibilité de l'utiliser même sur un système très réduit pour réparer des fichiers de configuration.

5. La compilation sous Unix

Le langage C est un langage compilé. Il y a:

- Le compilateur C utilisé sous Linux est gcc (*Gnu C Compiler*). On peut également l'invoquer sous le nom cc, comme c'est l'usage sous Unix. Ou g++ si on compile du code C++.
- Le debugger : l'outil utilisé sous Linux est nommé gdb (*Gnu Debugger*).
- Les outils de trace: trace en SunOS, truss en Solaris.

- Des outils associés: lint (qui recherche les éventuels problèmes), time (qui permet de connaître la rapidité d'exécution d'un programme), ldd (qui permet de connaître les librairies dynamiques utilisées par un programme), size, nm, ar.

Compiler, c'est:
Depuis une source a.c:

- passer le préprocesseur a.i
- générer un fichier assembleur a.s
- faire l'assemblage de ce fichier pour obtenir a.o
- faire l'édition de liens avec les librairies utiles

Plusieurs options simples peuvent être utilisées, dont les plus courantes sont :

- Le chemin de recherche des bibliothèques supplémentaires (en plus de /lib et /usr/lib), précédé de l'option –L.
- Le nom d'une bibliothèque supplémentaire à utiliser lors de l'édition des liens, précédé du préfixe -l. Il s'agit bien du nom de la bibliothèque, et pas du fichier. Par exemple la commande –lm permet d'inclure le fichier libm.so indispensable pour les fonctions mathématiques.
- On peut préciser le nom du fichier exécutable, précédé de l'option –o.
- Le chemin de recherche des fichiers d'en-tête (en plus de /usr/include), précédé de l'option –I.
- -O<number>: optimisation du code
- -c : s'arrêter au fichier .o
- -g : générer les informations pour le debugger
- L'option la plus couramment utilisée est -Wall, pour activer tous les avertissements.
- -static : force l'utilisation des librairies statiques (par défaut, les librairies utilisées sont dynamiques).

Exemple:

```
gcc -o mon_fichier mon_fichier.c
gcc -c mon_fichier.c
gcc -o mon_fichier mon_fichier.o
gcc -o mon_fichier mon_fichier.c –lm              # avec la librairie math en dynamique
gcc -o mon_fichier mon_fichier.c -static –lm      # avec la librairie math en statique
```

6. Gestion des erreurs

Avec la programmation système, nous allons étudier et manipuler tout ce qui touche à notre système d'exploitation.
Ainsi, nous devrons faire face assez souvent à des codes d'erreurs. La gestion des erreurs est donc un élément fondamental dans la programmation système.

La variable globale errno

Pour signaler une erreur, les fonctions renvoient une valeur spéciale qui est généralement -1 (sauf pour quelques exceptions). La valeur d'erreur alerte l'appelant de la survenance d'une erreur, mais elle ne fournit pas la description de ce qui s'est produit. La variable globale **errno** est alors utilisée pour en trouver la cause.
Cette variable est définie dans <errno.h> comme suit :

#include<errno.h>
extern int errno ;

La fonction perror

La bibliothèque C met également à notre disposition une fonction perror permettant une description de l'erreur qui s'est produite. Cette-fonction, la voici :

#include<stdio.h>
void perror(const char *s) ;

7. Programmer et compiler C sous Ubuntu

7.1. Installer gcc

Ouvrir un terminal. Il s'agit d'une fenêtre dans laquelle on va taper des commandes systèmes.
Passer en mode super-utilisateur.
smi@smi-desktop:~$ sudo su
Password: <- tapez votre password utilisateur
root@smi-desktop:~#
Remarquez que l'invite est # à la place de $, tapez :
root@smi-desktop:~# apt-get install gcc-4.7
Tapez ensuite ctrl-d ou exit pour retrouver le terminal utilisateur (invite $)
smi@smi-desktop:~$
gcc est maintenant complètement installé.

7.2. Un premier programme : « Hello World!»

On va pouvoir essayer un premier programme.
Créer un répertoire par la commande **mkdir tpc** et placer vous dans ce répertoire :
etudiant@etudiant-desktop:~$ cd tpc
Lancez un éditeur vi par la commande vi hello.c
Tapez le programme ci-dessous dans l'éditeur vi

```
#include <stdio.h>
int main (void) {
printf("Hello World!\n");
return 0 ;
}
```

7.3. Compiler par gcc

On peut maintenant le compiler par gcc
etudiant@etudiant -desktop:/~$ gcc -Wall hello.c -o hello
etudiant@etudiant -desktop:/~/tpc$
S'il y a des erreurs corrigez-les puis recommencez la compilation.
Lorsqu'il n'y a plus d'erreur on obtient le fichier exécutable hello que l'on peut lancer par :
etudiant@etudiant-desktop:/~/tpc$ ./hello
Hello World!
etudiant@etudiant-desktop:/~/tpc$

Chapitre 4 Système de gestion de fichiers

1. Notions générales

La mémoire centrale de l'ordinateur est une mémoire volatile, c'est-à-dire que son contenu s'efface lorsque l'alimentation électrique de l'ordinateur est interrompue.
Il faut stocker les données devant être conservées au delà de l'arrêt de la machine sur un support de masse permanent.
L'unité de conservation sur le support de masse est le **fichier**.

- Le système d'exploitation gère les fichiers via le **Système de gestion de fichiers (SGF)**.
- Deux vues : une vue logique (utilisateur), une vue physique (système).

1.1. Le fichier logique

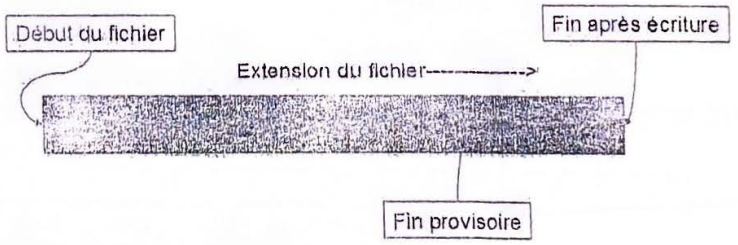
Le fichier logique est la vue de l'utilisateur de l'ensemble des données mémorisées sur le support de masse

- Un type de donnée standard défini dans les langages de programmation sur lequel peuvent être appliquées des opérations spécifiques comme la création, ouverture, fermeture et destruction de fichier.
- Un ensemble de données groupées sous forme d'enregistrements, désigné par un nom, Accessible selon différentes méthodes d'accès.

Les modes d'accès les plus courants sont l'accès séquentiel, l'accès indexé et l'accès direct.

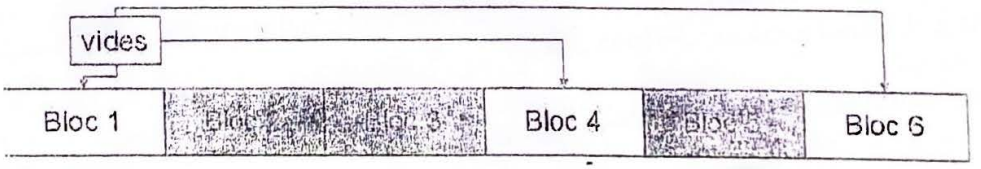
Accès séquentiel :

- Le plus simple
- Tout ajout d'information s'effectue à la fin du fichier et met à jour le pointeur de fin de fichier



Accès direct **(aussi dit accès aléatoire)** :

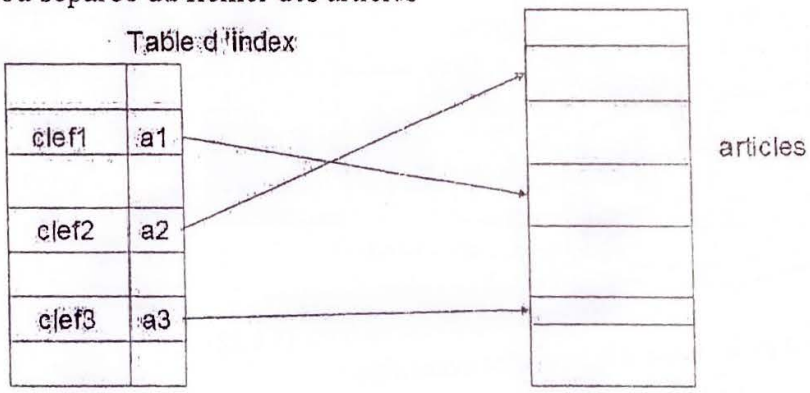
- Constitué d'enregistrements logiques de longueur fixe
- Permet l'accès immédiat à un enregistrement
- La taille du fichier est connue et peut être réservée à sa création (taille d'un bloc * nombre de blocs)



On peut accéder directement au bloc 5 sans que les précédents aient été remplis au préalable

Accès indexé :

- Nécessite d'avoir un ensemble de clés ordonnées
- Relation entre clés et articles établie au moyen d'une table d'index qui peut être incluse ou séparée du fichier des articles



1.2. Le fichier physique

Le fichier physique correspond à l'implémentation sur le support de masse de l'unité de conservation fichier.

a) Structure du disque dur

Un disque est constitué d'un ensemble de **plateaux**, empilés verticalement sur un même axe, formant ainsi ce que l'on appelle une pile de disques. Chaque plateau est composé d'une ou deux **faces**.

Chaque face est divisée en :

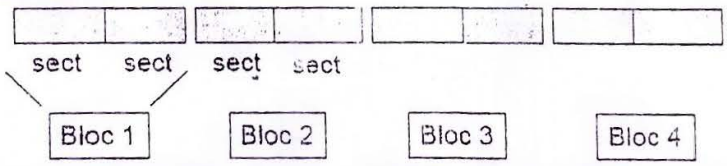
- **Secteurs**
- **Pistes**

L'intersection secteur et piste est un **Bloc**.

L'empilement des pistes forme un **Cylindre**.

Les opérations de lecture et d'écriture du SGF se font bloc par bloc.

1 bloc = 2 secteurs de 512 octets soit 1KO.



La plus petite unité accessible physiquement sur le disque est donc le secteur. Pour optimiser les opérations de lecture et écriture, les secteurs sont regroupés en bloc.

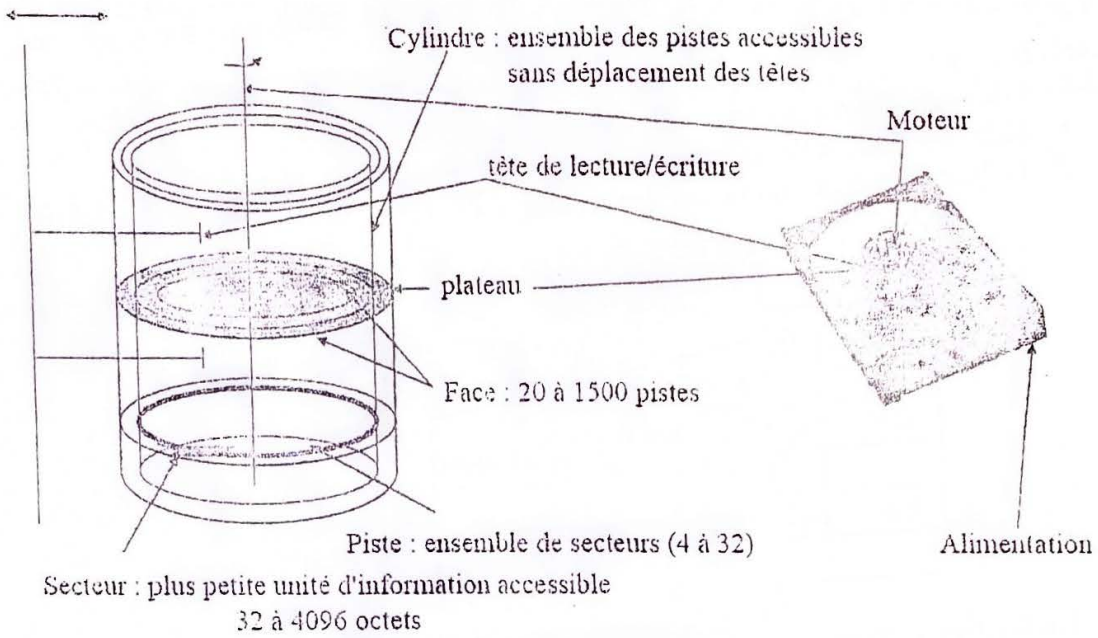


FIG. 1.1 – Structure du disque dur

b) Méthodes d'allocation de la mémoire secondaire

Un fichier physique est constitué d'un ensemble de blocs physique. Il existe plusieurs méthodes d'allocation des blocs physiques :

- allocation contiguë (séquentielle simple)
- allocation par blocs chainés
- allocation indexée

Allocation contiguë

Un fichier occupe un ensemble de blocs contigus sur le disque.

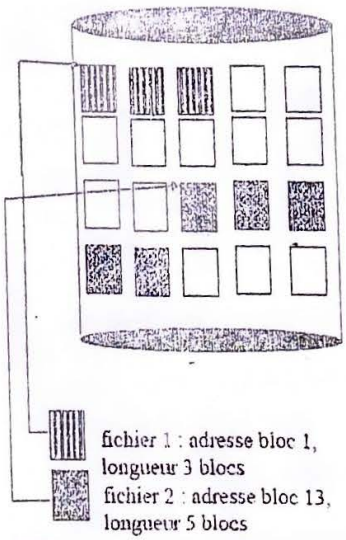


FIG. 1.2 – Allocation contiguë

Il découle de cette méthode d'allocation des problèmes :

- **Fragmentation** de l'espace disque pour trouver des espaces suffisants à une nouvelle allocation.
- **Compactage** pour regrouper les espaces libres dispersés et insuffisants en un seul espace libre exploitable

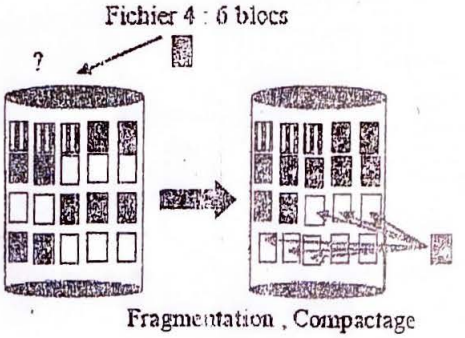


FIG. 1.3 – Création d'un nouveau fichier

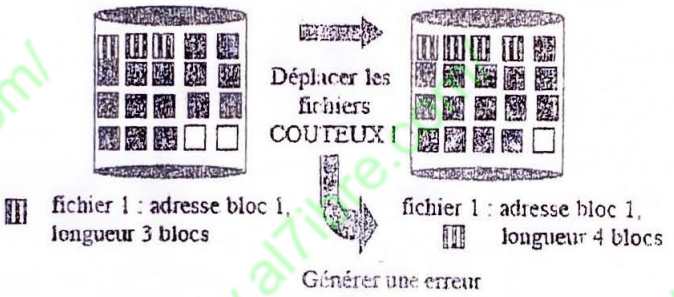


FIG. 1.4 – Extension du fichier

Allocation par bloc chainée

Un fichier est constitué comme une liste chainée de blocs physiques, qui peuvent être dispersés n'importe où sur le support de masse.

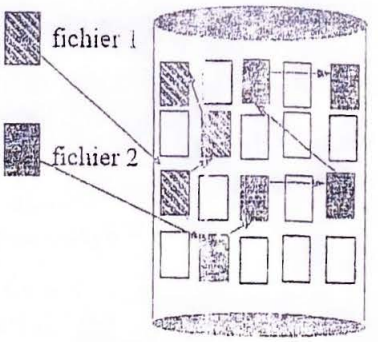


FIG. 1.5 – Allocation par blocs chainés

- Extension simple du fichier : il suffit d'allouer un nouveau bloc physique et le chaîner au dernier bloc physique du fichier.
- Pas de fragmentation

Difficultés :

- Le seul mode d'accès utilisable est le mode d'accès séquentiel

- Le chainage du bloc suivant occupe de la place dans un bloc

Allocation indexée

L'allocation indexée vise à supprimer les deux inconvénients de la méthode d'allocation chaînée. Dans cette méthodes, toutes les adresses des blocs physiques constituant un fichier sont rangées dans une table appelée index, elle-même contenue dans un ou plusieurs blocs disque.

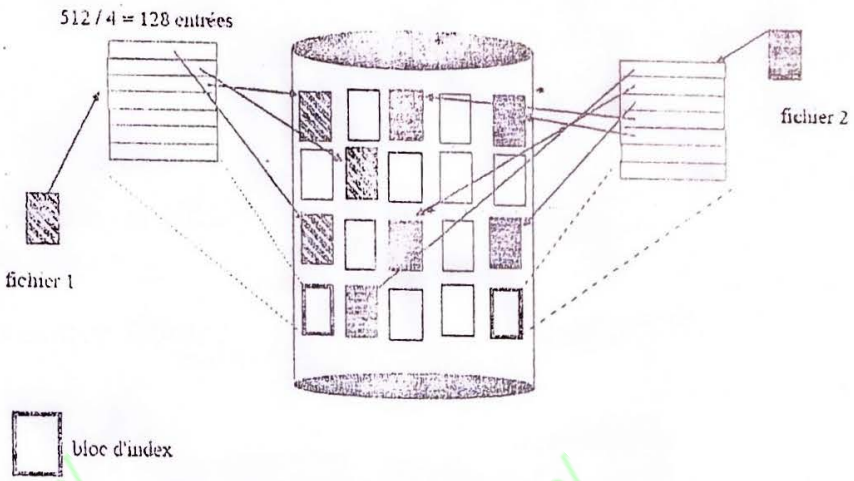


FIG. 1.5 – méthode d'allocation indexée

- Supporte bien l'accès direct
- « Perte de place » dans le bloc d'index

Gestion de l'espace libre

Le système maintient une liste d'espace libre, qui mémorise tous les blocs disque libres (non alloués).

Lors de la création d'un fichier ou de l'extension d'un fichier, le système recherche dans la liste d'espace libre de la quantité requise d'espace et allocation au fichier. L'espace alloué est supprimé de la liste. Lors de la destruction d'un fichier, l'espace libéré est intégré à la liste d'espace libre.

Il existe différentes représentations possibles de l'espace libre

- vecteur de bits
- liste chainée des blocs libres

Gestion de l'espace libre par un vecteur de bits

La liste d'espace libre est représentée par un vecteur binaire, dans lequel chaque bloc est figuré par un bit. La longueur de la chaîne binaire est égale au nombre de blocs existants sur le disque.

- Bloc libre : bit à 0
- Bloc alloué : bit à 1

Gestion de l'espace libre par liste chainée

La liste d'espace libre est représentée par une liste chainée des blocs libres.

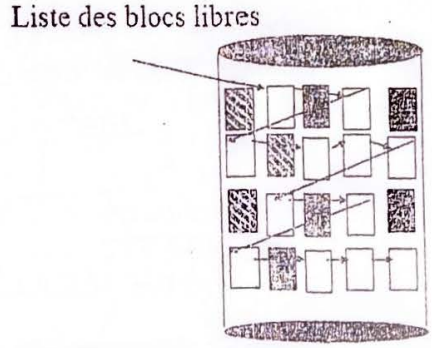


FIG. 1.6 – Gestion de l'espace libre par liste chaînée

- Nécessite le parcours d'une grande partie de la liste chaînée
- Difficile de trouver un groupe de blocs libres

1.3. Correspondance fichier logique-fichier physique

1.3.1. Notion de répertoire

Le système de gestion de fichiers effectue la correspondance entre fichiers logiques et fichiers physique par le biais d'une table appelée **répertoire** permettant de référencer tous les fichiers existant sur le disque avec leur nom et leurs caractéristiques principales.
Le répertoire stocke pour chaque fichier l'adresse des zones de données allouées au fichier.

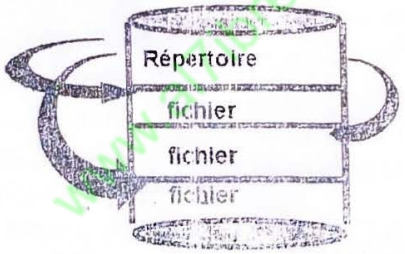


FIG. 1.7 – Répertoire

Un répertoire est une zone disque réservée par le SGF
Le répertoire comprend un certain nombre d'entrées. Une entrée de répertoire concernant un fichier donné, contient les informations suivantes :

- Nom du fichier physique
- Type du fichier
- Taille du fichier
- Le nom propriétaire du fichier
- Protections appliquées au fichier
- Date de création
- Adresse des zones de données

1.3.2. Organisation des répertoires

Répertoire à structure arborescente :

- chaque utilisateur dispose d'un sous-répertoire propre (*Répertoire de travail)*
- l'utilisateur peut créer des sous-répertoires à l'intérieur de son répertoire de travail

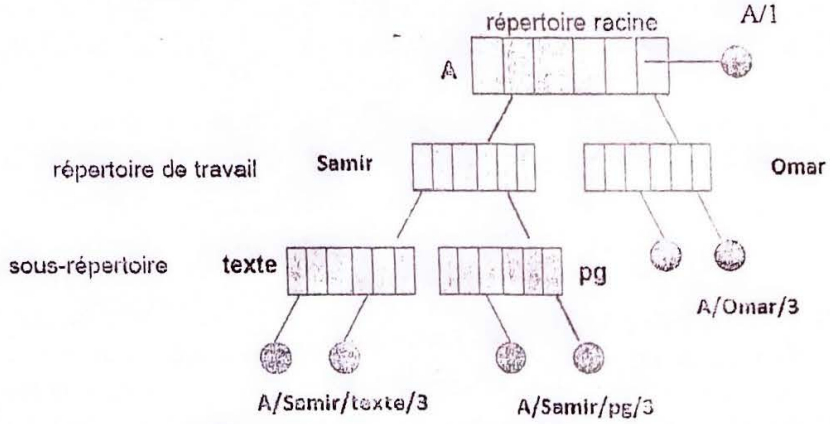


FIG. 1.8 – Structure de répertoire

1.3.3. Notion de partition

Le SGF d'un ordinateur peut comporter des milliers de fichiers répartis sur plusieurs giga-octets de disque ce qui complique la gestion de ces fichiers.

Solution : Diviser l'ensemble de SGF en morceaux indépendants appelés **partitions**.

Chaque partition constitue un disque virtuel, auquel est associé un répertoire qui référence l'ensemble des fichiers présents sur la partition. Le disque virtuel crée peut physiquement correspondre à une seule partie de disque physique.

Chaque partition est repérée par un nom appelé **label**. Pour pouvoir être accessible, la partition doit être connectée à l'arborescence de fichiers de la machine qui correspond à un répertoire, c'est l'**opération de montage de la partition**.

2. Les différents types de SGF

- MS-DOS utilise la **FAT** (File Allocation Table) pour y conserver les chaînages entre les blocs.
- Windows NT utilise la **MFT** (Master File Table) associé au système **NTFS** (New Technology File System).
- UNIX : extfs, ext2, ext3, ext4...

2.1. FAT

On parle généralement de système de fichiers **FAT16** et **FAT32.**

- Le FAT16 est utilisé par MS-DOS. En FAT16, les numéros de blocs sont écrits sur 16 bits. Si on suppose que la taille d'un bloc est 32Ko, la taille maximale adressable est alors 2Go (2^{16} x 32 Ko= 2097152 Ko = 2Go).
- Le FAT32 est pris en charge par Windows 95 et les versions qui ont suivis. Les numéros de blocs sont écrits sur 32 bits (en réalité, sur 28bits, 4 bits étant réservés). Si on suppose que la taille d'un bloc est de 32 ko, la taille maximale adressable théoriquement est de 8 To (2^{28} x 32 Ko = 8 To).

2.2. NTFS

Le système de fichiers NTFS (New Technology File System) est utilisé par Windows2000, WindowsNT, Windows XP et Windows Vista. Il utilise un système basé sur une structure appelée MFT (Master File Table), permettant de contenir des informations détaillées sur les fichiers.
La limite théorique de la taille d'une partition est de 16 hexa octets (17 milliards de To), mais la limite physique d'un disque est de 2To (va encoder en 64 bits $= 2^{64} = 18\ 446\ 744\ 073\ 709\ 551\ 616 = 16$ EiB (1 exbibyte = 1EiB= 260 bytes)).

2.3. Le système de gestion de fichiers de Linux

Un fichier physique Linux est dentifié par un nom et sans structure logique (suite d'octets).
La méthode d'allocation mise en œuvre est de type d'allocation indexée.
Un fichier Linux est composé d'un descripteur appelé **inode** et de blocs physiques, qui sont des blocs d'index ou des blocs de données.
Un bloc est identifié par un numéro codé sur 4 octets. La taille d'un bloc est un multiple de la taille d'un secteur (512 octets).

2.3.1. L'inode

L'inode du fichier est une structure stockée sur le disque, allouée à la création du fichier et repérée par un numéro. Contient les attributs du fichier :

- Nom
- Type : fichiers normaux, répertoires, périphériques, tubes nommés, sockets
- Droits d'accès
- Heures diverses
- Taille du fichier en octets
- Table des adresses des blocs de données

L'inode du fichier contient un tableau de EXT2_N_BLOCKS entrées qui égale par défaut à 15. L'organisation de cette table suit l'allocation indexé.

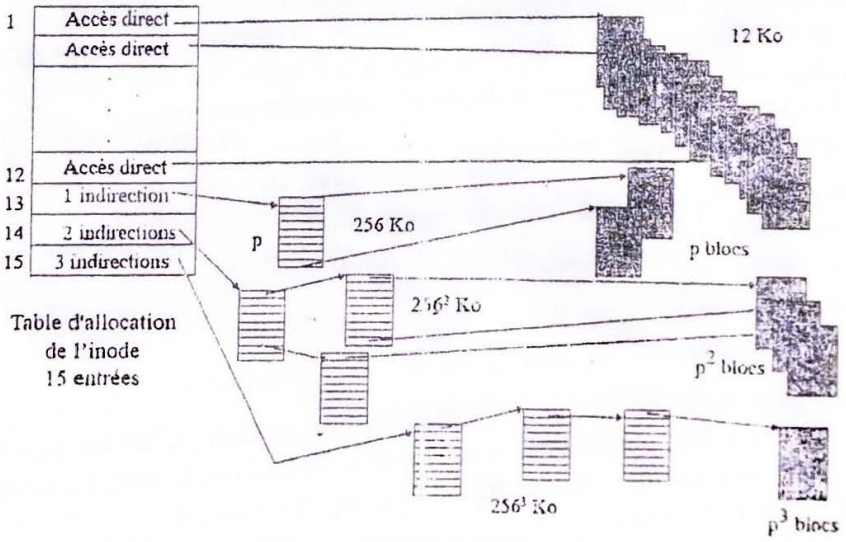
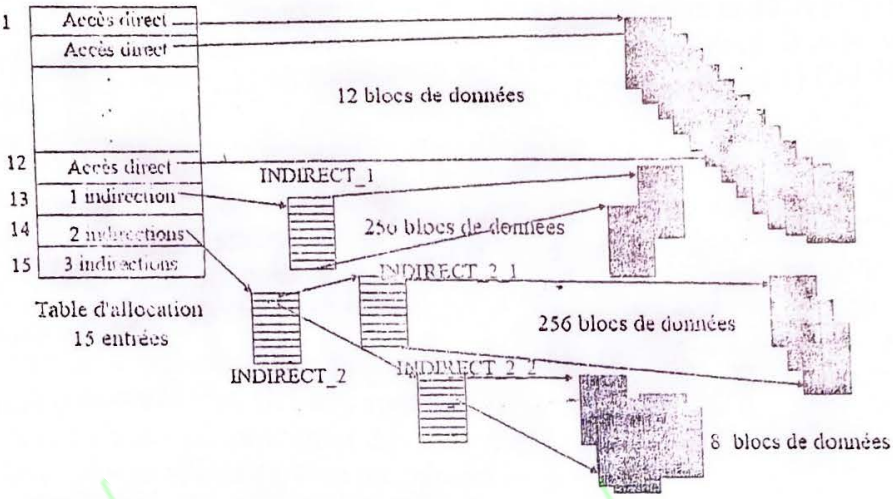


FIG. 2.1 – Adressage des blocs constituant un fichier

Si on considère des blocs de 1024 octets et qu'une adresse de bloc occupe 4 octets.
p nombre d'entrées d'un bloc d'index = taille bloc / taille numéro de bloc =256 entrées dans le bloc d'index.

Exemple :



Un fichier de 532 Ko, c'est-à-dire contient 532 blocs

Allocation : 12 blocs en accès direct
: 256 blocs de données pointés par le bloc index INDIRECT 1
: restent 532–12–256=264 blocs. Tous ces blocs sont pointés à partir du bloc d'index INDIRECT_2.
2 blocs d'index INDIRECT_2_1 et INDIRECT_2_2 sont nécessaires à ce niveau

2.3.2. Types de fichiers Linux

On distingue différents types de fichiers :

Les **fichiers répertoires** contiennent des informations sur les fichiers et les sous-répertoires. Du point de vue système, un répertoire est un fichier contenant des noms de fichiers. Un booléen indique pour chaque fichier du répertoire s'il s'agit d'un vrai fichier ou d'un sous-répertoire. d en début des droits d'accès indique un répertoire (directory).

```
etudiant> ls –ld Textes
drwx r-x ---    2 etudiant etudiant  1024 Jan 25 14 :07 Textes/
```

Les **fichiers ordinaires** contiennent les informations des utilisateurs. Ils sont en général des fichiers ASCII ou binaires. Un tiret (-) en début des droits d'accès indique un fichier ordinaire.

```
etudiant> ls –l Textes/lettre.txt
-rwx r-- r--    1 etudiant etudiant  81 Nov 30 14 :19 Textes/lettre.txt
```

Les **liens symboliques** sont une catégorie particulière de fichiers. C'est un raccourci vers un autre fichier. Le fichier « lien symbolique » contient le chemin et le nom du fichier à prendre en compte. L en début des droits d'accès indique un lien symbolique.

```
etudiant> ls –l Imprimer/lettre.txt
lrwx rwx rwx    1 etudiant etudiant  31 Jan 19 14 :16 Imprimer/lettre.txt
```

-> /home/etudiant/Textes/lettre.txt

Les **tubes nommés** sont des fichiers sur disque gérés comme un tube (pipe) entre un processus producteur et un processus consommateur. Un processus écrit des caractères dans un tube, un autre lit ces caractères à partir de ce tube. Les deux processus doivent se synchroniser. Il peut être crée avec **mknod** suivi du nom (tube ci-dessous) et de **p**, comme suit :

```
etudiant> mknod tube p     #ou mkfifo tube
etudiant>ls –l tube
prw- r--  r-- 1  etudiant etudiant   0 Jan 25 16 : 09 tube|
```

Les **sockets** permettent à un programme client d'échanger des données avec un serveur. L'usage des sockets est plus complexe que celui des tuyaux nommés, mais permet une communication bidirectionnelle avec chaque programme client. Aucune commande n'est disponible pour créer de tels sockets : ils sont créés par chaque serveur utilisant ces sockets.

Les périphériques sont vus comme des **fichiers spéciaux** qui appartiennent au sous-répertoire **/dev**. Nous avons deux types : les **fichiers spéciaux caractère** sont liés aux Entrées/Sorties et permettent de modéliser les périphériques d'E/S série tels que les terminaux, les imprimantes et les réseaux. Les **fichiers spéciaux bloc** modélisent les disques.
Le b précédant les droits d'accès de /dev/fd0 (floppy disk : lecteur de disquette) ou de /dev/hdc1 indique que le périphérique échange des blocs de données. Le c devant les droits d'accès de /dev/tty (le terminal de l'utilisateur) indique que les échanges se font caractère. La mémoire physique peut être accédée, si on en a les droits, en utilisant le fichier périphérique /dev/mem.

```
etudiant>ls –li /dev/fd0
  2312  brw- --- ---  1 etudiant floppy  2.  0  May  5 2015  /dev/fd0
etudiant>ls –li /dev/hdc1
  2380  brw- rw- ---  1 root     disk   22.  0  May  5 2015  /dev/hdc1
etudiant>ls –li /dev/lp0
  2590  crw- rw- ---  1  root daemon   6.    0 May  5 2015  /dev/lp0
etudiant>ls –li /dev/tty
  3200  crw- rw- rw-  1 root    root    5.     0 May  5 2015  /dev/tty
etudiant>ls –li /dev/mem
  2594  crw- r-- ---  1    root    kmem    1.  0 May  5 2015  /dev/mem
```

2.3.3. Structure d'un répertoire

Le système de gestion de fichiers **ext2** est organisé selon une forme arborescence. La racine de l'arborescence est représenté par le répertoire racine symbolisé par le caractère / et chacun des nœuds de l'arbre est lui-même un répertoire.

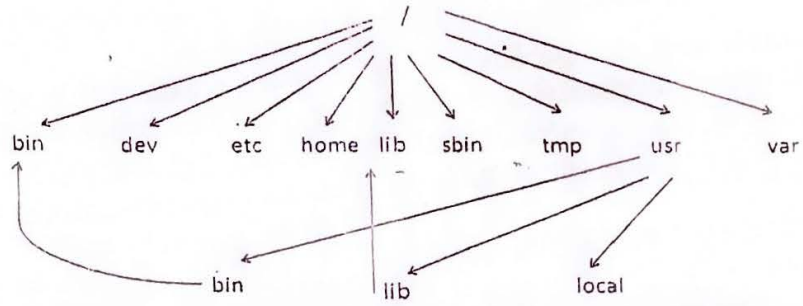


FIG. 2.1 – L'arbre des répertoires et fichiers du système

Sur la figure 2.1, il existe également un lien symbolique entre les répertoires /usr/lib et /lib.

Un répertoire est implémenté comme un type spécial de fichier, pour lequel les données contiennent les informations suivantes pour chaque fichier :

- Le numéro d'inode
- La longueur de l'entrée du répertoire
- La longueur du nom de fichier
- Le type du fichier
- Le nom du fichier

Fonctions des répertoires de la Figure 2.1

Contenu des principaux répertoires :

/bin : commandes de base d'Unix (ls, cat, cp, mv, rm, vi, etc.)

/dev : les fichiers spéciaux représentant les périphériques,
/dev/fd0 : lecteur de disquette
/dev/hdc1 : partition d'un disque dur
/dev/cdrom : périphérique lecteur de CD-ROM
/dev/lp0 : imprimante

/etc : fichier d'initialisation, de configuration, de mots de passe
/etc/passwd contient les mots de passe
/etc/fstab contient le système de fichiers à monter lors du lancement du système (fstab : file system table)

/home : répertoires personnels des utilisateurs

/lib : bibliothèques de programmes

/sbin : commandes d'administration : fsck, mkfs

/tmp : fichiers temporaires du système ou des utilisateurs

/usr : programmes et utilitaires mis à la disposition des utilisateurs.
/usr/bin : exécutables des utilitaires : cc, man
/usr/include : les fichiers d'en-tête pour développer en C comme par exemple stdio.h
/usr/games : répertoires de jeux
/usr/local contient les commandes locales
/usr/src contient les sources des programmes du système

/var : les données qui varient (fichiers en attente d'impression, courrier électronique, fichiers cache, etc.)
/var/spool en attente d'impression ou d'envoi
/var/mail courrier

2.3.4. Structure d'une partition

Une partition Ext2 est composée tout d'abord d'un bloc boot utilisé lors du démarrage du système puis d'un ensemble de groupe de blocs, chaque groupe contenant des blocs de données et des inodes enregistrés dans des pistes adjacentes. Chaque groupe de blocs contient à son tour les informations suivantes :

- Une copie du **super-bloc** du SGF
- Un vecteur de bits pour gérer les blocs libres du groupe
- Un groupe d'inodes souvent appelé table des inodes.
- Un vecteur de bits pour gérer les inodes libres
- Des blocs disques

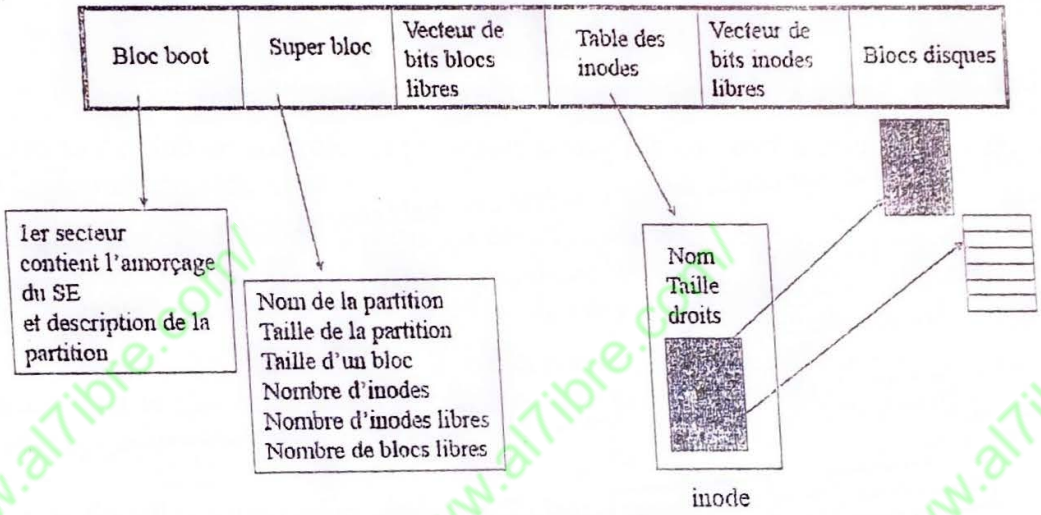


FIG. 2.2 – Partition Ext2

3. Le système de gestion de fichiers virtuel : VFS

3.2. Présentation

Le système Linux supporte d'autres systèmes de gestion de fichiers tels que Minix, MS-DOS, NTFS, les systèmes de gestion de fichiers des variantes d'Unix (Système V et BSD)…
Afin que l'accès à ces différents SGF soit transparent et uniforme pour l'utilisateur, une couche logicielle appelée VFS (Virtuel File System) est insérée dans le noyau Linux.

Cette couche logicielle offre à l'utilisateur un modèle de fichier commun à tous ces systèmes de gestion de fichiers, ainsi qu'un ensemble de primitives permettant de manipuler ce modèle commun.
Le VFS gère deux systèmes de cache :

- Un cache de noms qui conserve les conversions les plus récentes des noms de fichiers en numéro de périphérique et numéro d'inode.
- Un cache de tampons disque appelé buffer cache, qui contient un ensemble de blocs de données lus depuis le disque.

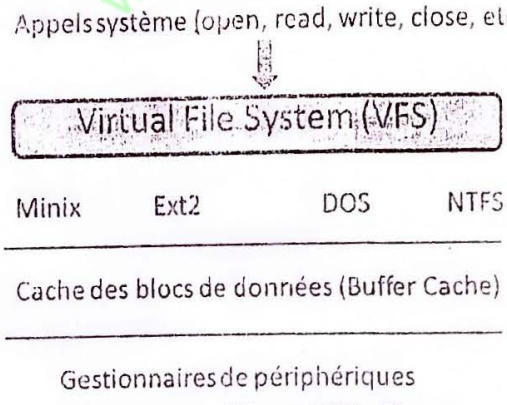


FIG. 3.1 – Virtual File System

3.2. Structure et fonctionnement du VFS

3.2.1. Les structures du VFS

La structure du VFS est organisée autour d'objets auxquels sont associés des traitements. Les objets sont :
L'objet système de gestion de fichiers (super-bloc), l'objet inode et l'objet fichier (file) qui sont définit dans <linux/fs.h>.
L'objet nom de fichier (dentry).

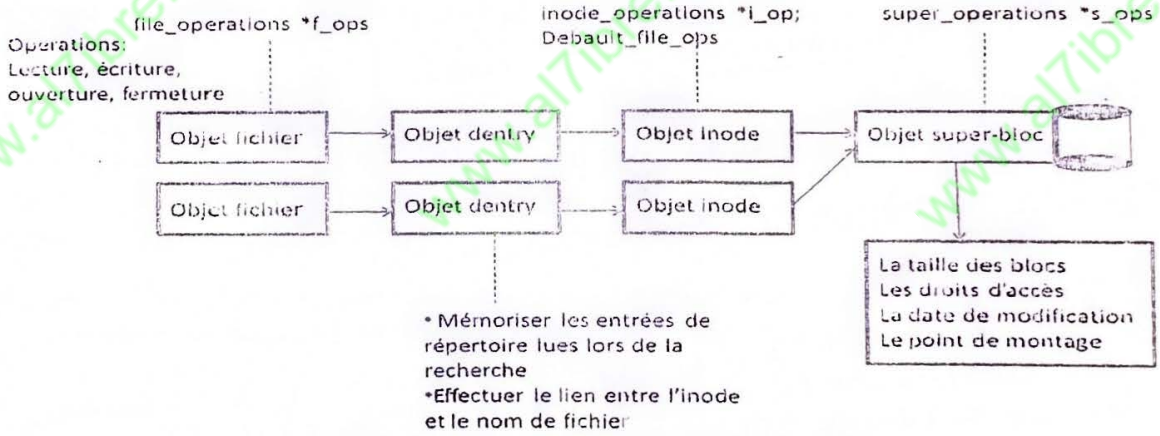


FIG. 3.2 – Objets du Virtual File System

3.2.2. Liaison du processus avec les objets du VFS

Les fichiers couramment ouverts par un processus sont mémorisés dans une table, appelée des **fichiers ouverts**, pointée depuis le bloc de contrôle du processus (champ files). Chacune des entrées de cette table (table fd), nommée **descripteur de fichier**, pointe vers un objet fichier (figure 3.3). Chaque objet fichier pointe à son tour vers un objet dentry, qui effectue la liaison entre le nom du fichier et son inode. Chaque objet dentry pointe vers l'objet inode du fichier, cet objet inode pointant à son tour vers l'objet super-bloc décrivant le système de gestion de fichiers auquel appartient le fichier.

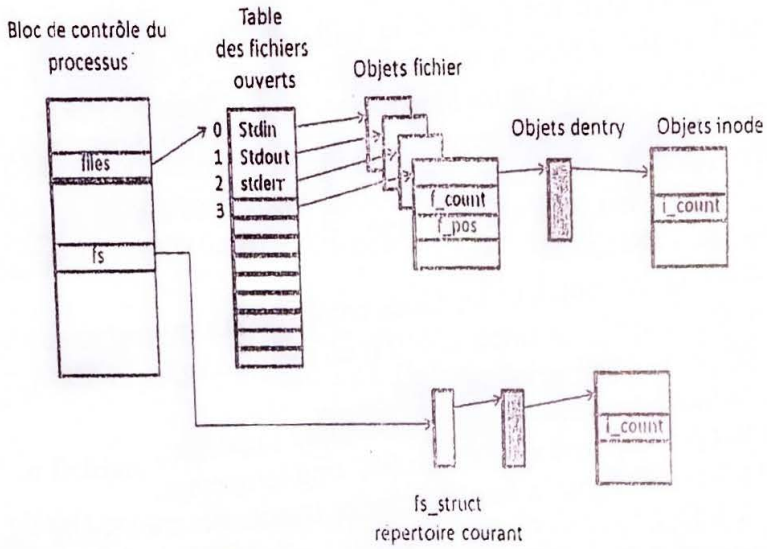


FIG. 3.3 – Tables des fichiers ouverts et relations entre les objets du VFS

Le champ fs du bloc de contrôle du processus pointe sur un objet fichier qui correspond au répertoire courant du processus.

f_count : nombre de références à partir des descripteurs de processus
f_pos : position courante
i_count : nombre de références à partir de la table des fichiers ouverts

Dans la table fd, les entrées 0, 1, 2 indiquent respectivement l'entrée standard (stdin) représentée par le clavier, la sortie standard (stdout) et la sortie d'erreur (stderr) représentées par l'écran.

3.2.3. Principe d'une opération générique : l'ouverture d'un fichier

Les actions effectuées par le VFS lors de l'ouverture d'un fichier par la primitive open(nom_fichier, mode_ouverture) :

- La routine d'enveloppe correspondant à la fonction open() lève une trappe, bascule le processus appelant en mode superviseur, puis appelle la routine système sys_open() après avoir passé les paramètres d'exécution (nom de fichier et mode d'ouverture).
- Alloue une entrée dans la table fd et création d'un nouvel objet fichier pointé par cette entrée.
- Obtention de l'inode associée au fichier objet.
- Les opérations concernant le fichier sont initialisées depuis le champ default_file_ops de la structure i_op de l'inode correspondant au fichier.
- L'appel de la fonction open() de l'objet fichier pour réaliser l'ouverture physique du fichier en fonction du type du SGF.
- L'index de l'entrée de la table fd pointant sur l'objet fichier est retourné à l'utilisateur.

3.2.4. Buffer cache

Le système maintient une liste de tampons mémoire qui joue le rôle de cache pour les blocs du disque et permet de réduire les entrées/sorties

- La taille d'un tampon est égale à la taille d'un bloc disque. Il est identifié par un numéro de bloc physique et numéro de périphérique.
- Lorsque le système doit lire un bloc depuis le disque : il cherche d'abord si le bloc est déjà présent dans la liste des tampons mémoire.
  - ✓ Si non, il prend un tampon libre et copie le bloc disque dans le tampon.
  - ✓ Si tous les tampons sont occupés, il libère un tampon en choisissant le moins récemment accédé.
- Les écritures s'effectuent dans les blocs copiés depuis le disque.
- Un tampon modifié n'est recopié que s'il est remplacé : possibilité de perte de données
- Les tampons sont sauvegardés en outre :
  - ✓ Le nombre de tampons modifiés est trop important;
  - ✓ Le tampon est resté modifié en MC trop longtemps;
  - ✓ Un processus force la sauvegarde des tampons le concernant : appels systèmes sync(), fsync()…

4. Primitives du VSF

4.1. Opérations sur les fichiers

a) Ouverture d'un fichier

L'ouverture d'un fichier s'effectue par un appel à la primitive open() dont le prototype est donné :

```
#include<sys/types.h>
#include <unistd.h>
#include<sys/stat.h>
#include <fcntl.h>
int open(const char *ref, int mode_ouv, mode_t mode);
```

Le premier paramètre *ref spécifié le chemin du fichier à ouvrir dans l'arborescence du SGF.
Le paramètre mode_ouv est une combinaison de constantes permettant de spécifier les options sur le mode d'ouverture. Ces constantes peuvent être combinées entre elles à l'aide de l'opération de OU binaire « | », ce sont :

O_RDONLY	ouverture en lecture
O_WRONLY	ouverture en écriture
O_RDWR	ouverture en lecture-écriture
O_CREAT	création du fichier s'il n'existe
O_TRUNC	suppression du fichier s'il existe
O_APPEND	écriture en fin de fichier (ajout en fin de fichier)
O_SYNC	écriture immédiate sur disque en vidant les tampons
O_NONBLOCK	ouverture non bloquante : s'il n'y a rien à lire, on n'attend pas

Le paramètre mode : si O_CREAT est vrai, le paramètre mode précise les droits d'accès du fichier à créer à l'aide de constantes symboliques définies dans <sys/stat.h> :

S_IRUSER, S_IRGRP, S_IROTH (Read for User, Group or Others),
S_IWUSR, S_IWGRP, S_IWOTH (Write for User, Group or Others),
S_IXUSR, S_IXGRP, S_IXOTH (eXecution for User, Group or Others),

S_IRWXU (R,W, X pour User),
S_IRWXG (R, W, X pour Group),
S_IRWXO (R, W, X pour Others),

Comme nous avons vu précédemment, l'ouverture de fichier entraîne la création d'une nouvelle entrée dans la table des fichiers ouverts du processus, le chargement de l'inode du fichier depuis le disque ou depuis le cache des inodes, enfin l'allocation d'un nouvel objet fichier.

L'index de l'entrée dans la table des fichiers ouverts, constituant le descripteur de fichier, est retourné comme résultat de la primitive open(). En cas d'échec la primitive retourne la valeur -1.

b) Fermeture d'un fichier

L'appel système **close()** permet de libérer un descripteur de fichier desc. Le fichier pointé par le descripteur libéré est fermé s'il n'a plus aucun descripteur associé au fichier.

```
#include<unistd>
int close (int desc);
```

c) Lecture et écriture dans un fichier

Lecture d'un fichier

```
#include <unistd.h>
int read(int desc, void *ptr_buf, size_t nb_octets);
```

L'appel système **read()** lit, à partir de la position courante du pointeur, nb_octets octets au maximum et les copie dans le tampon ptr_buf. En cas de succès, il retourne le nombre d'octets réellement lus. L'appel système read() retourne 0 si la fin de fichier est atteinte. Il retourne -1, en cas d'erreur.

Ecriture dans un fichier

```
#include <unistd.h>
int write(int desc, void *ptr_buf, int nb_octets);
```

L'appel système **write()** copie nb_octets octets du tampon ptr_buf vers le fichier desc, à partir de la position pointée par le pointeur de fichier. En cas de succès, il retourne le nombre d'octets réellement écrits. Sinon, il renvoie -1.

Exemple 1 : Le programme « copie_std.c » crée un fichier appelé "fichier" dans lequel il copie les données lues à partir du clavier. Les Entrées/Sorties sont effectuées avec des appels système (de bas niveau) sans utiliser la bibliothèque standard de C "stdio.h".

```
#include <unistd.h>
#include <fcntl.h>
#define taille 80
int main()
{
int fd,nbcar;
char buf[taille];
// créer un fichier
fd=open("fichier",O_CREAT|O_WRONLY);
```

```
if(fd==1)
{
write(2,"Erreur d'ouverture\n",25) ;
return 1;
}
write(1,"Ouverture avec succès\n",30) ;
// copier les données introduites à partir
// du clavier dans le fichier
while((nbcar=read(0,buf,taille))>0)
if(write(f,buf,nbcar) == 1)
return 1;
return 0;
}
```

d) Se positionner dans un fichier

Les lectures et les écritures sur un fichier s'effectuent de façon séquentielle. Il est cependant possible de modifier la position courante du pointeur de fichier à l'aide de la primitive **lseek()** dont le prototype est :

```
#include<unisd.h>
off_t lseek(int desc,off_t dep, int option) ;
```

Le pointeur du fichier désigné par desc est modifié d'une valeur égale à dep octets selon une base spécifiée dans le champ option. Ce champ option peut prendre trois valeurs :

SEEK_SET : le positionnement est effectué par rapport au début du fichier,
SEEK_CUR : le positionnement est effectué par rapport à la position courante,
SEEK_END : le positionnement est effectué par rapport à la fin du fichier.
Le pointeur retourne la position courante en octets par rapport au début du fichier à la suite du positionnement, et la valeur -1 en cas d'échec.

Exemple 2 : Accès directs aux éléments d'un fichier avec le programme seek.c

```
programme seek.c
#include<unistd.h>
#include<fcntl.h>
#include<stdio.h>
#include<stdlib.h>
int main(int argc,char* argv[])
{
int PLine[200];
int fd,ncar,i;
int nline=0,pfich=0;
char buffer[4096];
// ouverture du fichier en lecture seulement
fd=open(argv[1],O_RDONLY);
if(fd==-1) exit(1); // erreur d' ouverture
PLine[0]=0; // position de la ligne 0
while(1)
```

```
{ //lecture du fichier
ncar=read(fd,buffer,4096);
if(ncar==0) break; // fin de fichier
if(ncar==-1) exit(1); // erreur de lecture
for(i=0;i<ncar;i++)
{
pfich++;
// fin de ligne rencontrée
if(buffer[i]=='\n')
PLine[++nline]=pfich;
}
}
PLine[nline+1]=pfich;
for(i=0;i<=nline;i++)
printf("PLine[%d]=%d \n",i,PLine[i]);
// accès à la première ligne en utilisant lseek
lseek(fd,PLine[0],SEEK_SET);
ncar=read(fd,buffer,PLine[1]-PLine[0]);
// afficher à l'écran le contenu du buffer
write(1,buffer,ncar);
return 0;
}
```

Exécution de seek.c:

```
etudiant> cat >> exemple
nom
prenom
code
etudiant> cat exemple
nom
prenom
code
etudiant>./ seek exemple
PLine[0] = 0
PLine[1] = 4
PLine[2] = 11
PLine[3] = 16
Nom
```

Exemple 3 : Observer le petit programme lseek.c:

```
#include<unistd.h>
#include<fcntl.h>
#include<stdlib.h>
int main(void)
{
int fd;
if((fd=open("file",O_RDWR))<0)
```

```
exit(1);
if(lseek(fd,10,SEEK_SET)<0)
exit(1);
if(write(fd,"123",3)<0)
exit(1);
return 0;
}
```

L'exécution sur un fichier file:

```
etudiant> cat >> file
ABCDEFGHIJKLMNOPQRSTUVWXYZ
etudiant> ./lseek
etudiant> cat file
ABCDEFGHIJ123NOPQRSTUVWXYZ
```

Exemple 4 : L'exemple suivant crée un fichier nommé /home/etudiant/fichnotes avec des droits en lecture et écriture pour le propriétaire du fichier. Dans ce fichier, 2 couples nom d'élèves et note sont enregistrés. Puis le pointeur du fichier est positionné au début du fichier et l'ensemble des éléments précédemment écrits est relu.

```
#include <stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
int main(){
struct eleve {
char nom[10];
int note;
};
int fd,i,ret;
struct eleve un_eleve;
fd=open("/home/etudiant/fichnotes",O_RDWR|O_CREAT,S_IRUSR|S_IWUSR);
if(fd==-1)
perror("prob open");
i=0;
while(i<4){
printf("Donnez le nom de l'élève \n");
scanf("%s",un_eleve.nom);
printf("donnez la note de l'élève \n");
scanf("%d",&un_eleve.note);
write(fd,&un_eleve,sizeof(un_eleve));
i=i+1;
}
ret=lseek(fd,0,SEEK_SET);
if(ret==-1)
perror("prob lseek");
printf("la nouvelle position est %d \n",ret);
```

```
i=0;
while(i<4){
read(fd,&un_eleve,sizeof(un_eleve));
printf("le nom et la note de l'élève sont %s, %d \n",un_eleve.nom,un_eleve.note);
i=i+1;}
close(fd);
return 0 ;
}
```

e) Manipuler les attributs des fichiers

Accès aux attributs des fichiers

Les caractéristiques des fichiers sont enregistrées dans un inode sur disque. L'appel système **stat** permet d'obtenir certaines des informations de l'inode. Les résultats sont communiqués dans une structure de type **struct stat** décrite dans le fichier d'en-tête **<sys/stat.h>**. La syntaxe est la suivante :

```
int stat(const char *ref, struct stat *infos);
int fstat(cons tint desc, struct stat *infos);
```

stat permet d'obtenir les informations à partir du nom du fichier. Pour **fstat**, il faut utiliser le descripteur obtenu avec open.

```
struct stat {
mode_t st_mode;          /* type du fichier et droits accès utilisateur */
ino_t st_ino;            /* inode du fichier sur disque */
dev_t st_dev;            /* dispositif */
nlink_t st_nlink;        /* nombre de liens physiques */
uid_t st_uid;            /* propriétaire du fichier */
gid_t st_gid;            /* groupe propriétaire du fichier */
off_t st_size;           /* taille du fichier en octets */
time_t st_atime;         /* dernier accès */
time_t st_mtime;         /* la date de la dernière modification */
time_t st_ctime;         /* dernière modification de données */
uint_t st_blksize ;      /* taille d'un bloc dans le fichier */
int st_blocks ;          /*blocs alloués pour le fichier */
};
```

Il y a des macros prédéfinis qui prennent buf.st_mode comme argument et retournent 1 ou 0 :

- S_ISDIR(infos.st_mode) : vrai si le fichier est un répertoire.
- S_ISCHAR(infos.st_mode) : vrai si c'est un fichier spécial caractère.
- S_ISBLK(infos.st_mode) : vrai si c'est un fichier spécial bloc.
- S_ISREG(infos.st_mode) : vrai si c'est un fichier régulier.
- S_ISFIFO(infos.st_mode) : vrai si c'est un pipe ou FIFO.

Exemple 5 : Le programme « stat.c » permet de récupérer des informations à partir d'un fichier et de donner les caractéristiques d'une liste de fichiers.

```
#include<sys/stat.h>
#include<stdio.h>
```

```
#include<fcntl.h>
#include<stdlib.h>
#include<stdio.h>
int main(int argc,char* argv[])
{
struct stat buf;
mode_t mode;
char c;
int result;
result=stat(argv[1],&buf);
if(result==-1)
printf("Infos sur %s non disponibles\n",argv[1]);
else
{
mode=buf.st_mode;
if(S_ISDIR(mode)){
c='d';
printf("%s est un répertoire",argv[1]);}
else
if(S_ISREG(mode))
{c='-';
printf("%s est un fichier ordinaire\n",argv[1]);}
else
if(S_ISFIFO(mode))
{c='p';
printf("%s est un pipe",argv[1]);}
else
printf("%s est un fichier special",argv[1]);
printf("Droits d'accès :");
printf("%c",c);
printf("%c",mode & S_IRUSR ? 'r':'-');
printf("%c",mode & S_IRUSR ? 'w':'-');
printf("%c",mode & S_IXUSR ? 'x':'-');
printf("%c",mode & S_IRGRP ? 'r':'-');
printf("%c",mode & S_IWGRP ? 'w':'-');
printf("%c",mode & S_IXGRP ? 'x':'-');
printf("%c",mode & S_IROTH ? 'r':'-');
printf("%c",mode & S_IWOTH ? 'w':'-');
printf("%c",mode & S_IXOTH ? 'x':'-');
printf("\n");
printf("Numéro de l'inode : %ld \n",buf.st_ino);
printf("Mode (en hexa) :0x%x\n",buf.st_mode);
printf("Link : %d\n",buf.st_nlink);
printf("Numéro du propriétaire : %d \n",buf.st_uid);
printf("Numéro du groupe : %d \n",buf.st_gid);
printf("Taille du fichier : %ld \n",buf.st_size);
```

```
}
return 0;
}
```

Exécution de stat.c:

```
etudiant>./ stat
Infos sur (null) non disponibles
etudiant> ./stat stat.c
stat.c est un fichier ordinaire
Droits d'accès          : -rw- rw- r—
Numéro de l'inode       : 4849695
Mode  (en hexa)         : 0x81b4
Numéro du propriétaire  : 1000
Numéro du groupe        : 1000
Taille du fichier       : 1314
```

Création et destruction de liens

La création d'un nouveau lien physique ref2 pour le fichier ref1 s'effectue par appel à la primitive **link()** :

```
#include<unistd.h>
int link(const char *ref1, const char *ref2) ;
```

La suppression d'un lien physique ref s'effectue par appel à la primitive **unlink()** :

```
#include<unistd.h>
int unlink(const char *ref) ;
```

Exemple 6 : Le programme « unlink_cs.c » permet d'effacer (et encore re-utiliser) un fichier.

```
#include<unistd.h>
#include<fcntl.h>
#include<stdlib.h>
#define N 16
int main(void)
{
char chaine[N];
int fp;
write(1,"Création fichier",17);
fp=open("test",O_RDWR|O_CREAT);
if(fp<0)
exit(0);
write(fp,"\n0123456789abcdef",N);
write (1,"\nEffacement fichier",19);
if(unlink("test")==1)
{
perror("unlink");
exit(1);}
write (1,"\nRelecture du contenu du fichier",32);
lseek(fp,0,SEEK_SET);
```

```
read(fp,chaine,N);
write (1,chaine,N);
write (1,"\nFermeture fichier",18);
close(fp);
return 0;
}
```

Exécution de unlink_cs.c:
etudiant> gcc -o unlink unlink_cs.c
etudiant>./ unlink
Création fichier
Effacement fichier
Relecture du contenu du fichier
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Fermeture fichier

Modification du nom du fichier

La modification d'un nom de fichier s'effectue par un appel à la primitive **rename()** :

```
#include<unistd.h>
int rename(const char *ref_anc, const char *ref_nv) ;
```

Le fichier ref_anc est renommé ref_nv. La commande mv ref_anc ref_nv exécute la même action depuis l'interpréteur de commande.

Modification du propriétaire d'un fichier

Le propriétaire et le groupe propriétaire d'un fichier sont fixés lors de la création de ce fichier, en fonction de l'identité du processus demandant la création. Ces deux informations peuvent être modifiées par la suite en faisant appel aux primitives **chown()** et **fchown()**:

```
#include<unistd.h>
int chown(const char *ref, uid_t id_util, gid_t id_grp) ;
int fchown(int desc, uid_t id_util, gid_t id_grp) ;
```

La primitive **chown** prend comme premier paramètre le nom du fichier ref tandis que la primitive fchown prend comme premier paramètre le descripteur du fichier desc déjà ouvert. Le second paramètre id_util permet dans les deux cas de spécifier l'identité du nouveau propriétaire tandis que le troisième id_grp permet de spécifier l'identité du groupe du nouveau propriétaire. Chacun de ces deux derniers paramètres peut être omis et remplacé par la valeur -1.

Modification et test des droits d'accès d'un fichier

chmod modifie les droits d'accès à un fichier ref ou descripteur du fichier desc déjà ouvert. Les nouveaux droits dans les deux cas sont donnés par le paramètre mode et ils prennent les mêmes valeurs que le troisième paramètre de la primitive open().

```
#include<sys/stat.h>
int chmod(const char *ref, mode_t mode) ;
int fchmod(int desc, mode_t mode) ;
```

access permet à un processus de tester ses droits sur un fichier de nom ref. La syntaxe est la suivante :

```
#include<unistd.h>
int access(const char * ref, int mode) ;
```

mode indique les droits d'accès à tester :
R_OK, W_OK, X_OK, F_OK (droit de lecture, écriture, exécution, existance).
L'appel système retourne 0 si le droit est acquis pour le mode indiqué.

4.2. Opérations sur les répertoires

Ces appels système permettent de créer un répertoire, de détruire un répertoire vide, de lister un répertoire.

Création d'un répertoire

Un répertoire est crée par un appel à la primitive **mkdir()** dont le prototype est :

```
#include <sys/types.h>
#include <dirent.h>
#include<unistd.h>
int mkdir(const char *ref, mode_t mode);
```

Le premier paramètre ref spécifié le nom du répertoire à créer. Le champ mode spécifie les droits d'accès associé à ce répertoire et prend les mêmes valeurs que le troisième paramètre de la primitive open().

Il retourne 0 ou -1 en cas d'erreur.

Suppression d'un répertoire

```
#include <sys/types.h>
int rmdir(const char *ref);
```

rmdir() efface le répertoire s'il est vide. Si le répertoire n'est pas vide, on ne l'efface pas. Le paramètre ref correspond au nom du répertoire. Il retourne 0 ou -1 en cas d'erreur.

Changement de répertoire courant

chdir() change le répertoire courant qui devient ref.

```
int chdir(const char *ref) ;
```

Connaître le nom de répertoire courant

getcwd() permet de connaître le nom de son répertoire courant :

```
#include<unistd.h>
char *getcwd(char *buf, size_t taille)
```

Le nom absolu du répertoire courant du processus appelant est retourné dans le tampon buf dont la taille est spécifiée dans le paramètre taille. En cas de succès, la primitive renvoie un pointeur sur le tampon buf et sinon la valeur NULL.

Renommer un répertoire

```
#include <unistd.h>
```

int rename(char *old, char *new);

rename() change le nom du répertoire old. Le nom nouveau est new.
Le paramètre old nom d'un répertoire existant, et new le nom nouveau du répertoire. Il retourne 0 ou -1 en cas d'erreur.

Exploration d'un répertoire

Trois primitives permettent aux processus d'accéder aux entrées d'un répertoire. Ce sont les appels systèmes **opendir()**, **readdir()** et **closedir()** qui permettent respectivement d'ouvrir un répertoire, de lire une entrée du répertoire et de fermer le répertoire. Les prototypes de ces trois fonctions sont :

```
#include <sys/types.h>
#include <dirent.h>
#include<unistd.h>
DIR *opendir(const char *ref);
struct dirent *readdir(DIR *rep);
int closedir(DIR *rep);
```

Le type **DIR** défini dans le fichier <dirent.h> représente un descripteur de répertoire ouvert. La structure struct dirent correspond à une entrée de répertoire. Elle comprend les champs suivants :

- **long d_ino**, le numéro d'inode de l'entrée,
- **unsigned short d_reclen**, la taille de la structure retournée,
- **char [] d_name**, le nom de fichier de l'entrée.

La primitive **opendir()** ouvre le répertoire ref en lecture et retourne un descripteur pour ce répertoire ouvert.
La primitive **readdir()** lit une entrée du répertoire désigné par le descripteur rep. Cette entrée est retournée dans une structure de type struct dirent.
La primitive **closedir()** ferme le répertoire désigné par le descripteur rep.

Exemple :
Dans cet exemple, le répertoire courant du processus est ouvert et chacune des entrées de celui-ci est lue et les informations associées (numéro de l'inode, nom du fichier) sont affichées.

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<dirent.h>
int main()
{
char nom[20];
struct dirent *entree;
DIR *fd;
getcwd(nom,20);
```

```
printf("Mon répertoire courant est %s \n",nom);
fd=opendir(nom);
entree=readdir(fd);
while(entree!=NULL)
{
printf("le numéro d'inode de l'entrée est %ld et le nom de fichier correspondant est
%s \n",entree->d_ino,entree->d_name);
entree=readdir(fd);
}
closedir(fd);
return 0;
}
```

4.3. Opérations sur les liens symboliques

Les primitives **symlink()** et **readlink()** permettent respectivement de créer un lien symbolique et de lire le nom du fichier sur lequel il pointe. Les prototypes de ces 2 fonctions sont :

```
#include<unistd.h>
int symlink(const char *ancref, const char *nvref) ;
int readlink(const char *ref, char *buf, size_t taille) ;
```

La primitive **symlink()** crée un lien symbolique nommé **nvref**, qui pointe sur le fichier **ancref**. La commande **ln –s ancref nvref** réalise la même action depuis l'interpréteur de commandes.

La primitive **readlink()** retourne dans le tampon **buf** dont la taille est spécifiée dans le paramètre **taille**, le nom de fichier pointé par le lien **ref**. Elle rend par ailleurs le nombre de caractères placés dans **buf**. Si le premier argument à **readlink()** pointe vers un fichier qui n'est pas un lien symbolique, **readlink** donne **errno = EINVAL** et retourne -1.

Exemple 1 : Le programme « lien_sym.c » imprime le chemin d'un lien symbolique spécifié sur la ligne de commande :

```
#include<errno.h>
#include<stdio.h>
#include<unistd.h>
int main(int argc, char *argv[])
{
char target_path[256];
char *link_path=argv[1];
// Essai de lire le chemin du lien symbolique
int len=readlink(link_path,target_path,sizeof(target_path));
if(len==-1)
{
// L'appel échoué
if(errno==EINVAL)
printf("%s n'est pas un lien symbolique\n",link_path);
else
// Un autre problème a eu lieu
```

```
perror("readlink");
return 1;}
else
{
// NUL terminaison du chemin objectif
target_path[len]='\0';
// Imprimer
printf("%s \n",target_path);
return 0;
}}
```

Regardez l'utilisation de lien_sym.c:
etudiant> ln -s /usr/bin/wc my_link
etudiant> ./lien_sym my_link
/usr/bin/wc

Exemple 2. Voici un autre exemple sur les différences entre liens symboliques et physiques. L'utilisation de l'option "**-li**" (listage des i-noeuds) de **ls** sera utilisée. Montrons d'abord les liens symboliques :
etudiant> cat > ladiff.txt
exemple d'un fichier. ^D
etudiant> ls -l ladiff*
-rw-r--r-- 1 etudiant 48 Nov 18 15:06 ladiff.txt
etudiant> ln ladiff.txt ladiff.ln
pascal> ls -li ladiff*
3052239 -rw-r--r-- 2 etudiant 48 Nov 18 15:06 ladiff.ln
3052239 -rw-r--r-- 2 etudiant 48 Nov 18 15:06 ladiff.txt
etudiant > rm ladiff.txt
etudiant > cat ladiff.ln
exemple d'un fichier.
etudiant > ls -li ladiff*
3052239 -rw-r--r-- 1 etudiant 48 Nov 18 15:06 ladiff.ln

4.4. Opérations sur les partitions

Les primitives **mount()** et **umount()** permettent de monter ou démonter une partition sur l'arborescence de fichiers de façon à la rendre accessible ou inaccessible. Les prototypes de ces primitives sont :

```
#include<sys/mount.h>
#include<linux/fs.h>
int mount(const char *fichierspecial, const char *dir, const char *sgf, unsigned long flag, const void *data) ;
int umount(const char *fichierspecial) ;
int umount(const char *dir) ;
```

La primitive **mount()** monte, sur le répertoire **dir**, le système de gestion de fichiers présent sur le périphérique dont le nom est donné dans le paramètre **fichierspecial**. Le paramètre **sgf** indique le type du système de gestion de fichiers et peut prendre les valeurs « minix »,

« ext2 », « proc », etc. Le paramètre **flag** spécifie les options de montages sous forme de constantes qui peuvent être combinées par l'opération de OU binaire « | ». Les valeurs admises pour ces constantes sont :
MS_RONLY, les fichiers montés sont accessible en lecture seule,
MS_NOSUID, les bits setuid et setgidne ne sont pas utilisés,
MS_NODEV, les fichiers spéciaux ne sont pas accessible,
MS_NOEXEC, les programmes ne peuvent pas exécutées,
MS_SYNCHRONOUS, les écritures sont synchrones.

Le paramètre **data** spécifié d'autres options dépendants du type de système de gestion de fichiers.

La primitive **umount()** démonte un système de gestion de fichiers. Elle prend comme paramètre soit le nom du système de gestion de fichiers **fichierspecial**, soit le nom du point de montage **dir**. Ces deux primitives ne peuvent êtres exécutées que par le super-utilisateur.

Chapitre 5 Gestion de la mémoire

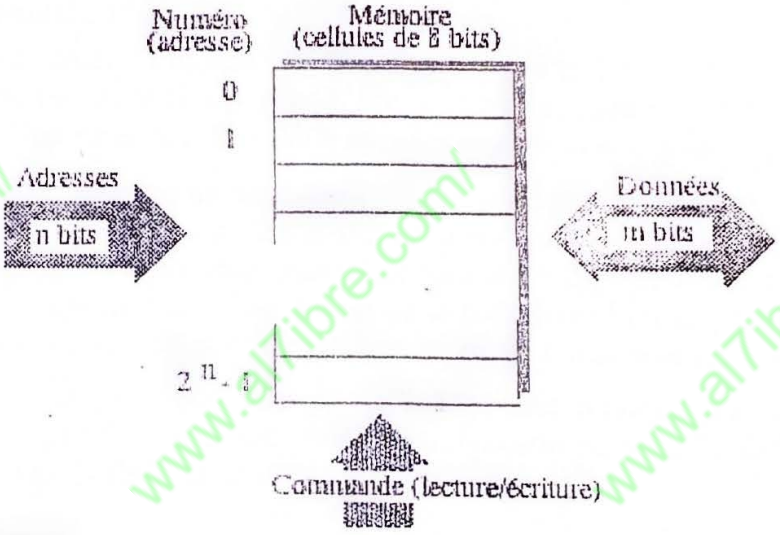
1. Introduction

Pourquoi la mémoire ?

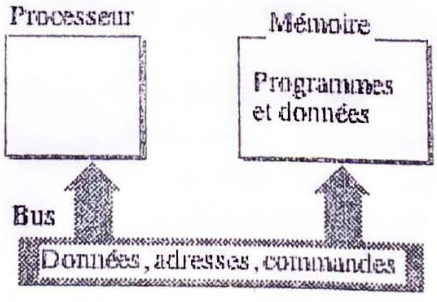
Le processeur va chercher les informations dont il a besoin, c'est-à-dire les instructions et les données, dans la mémoire. Il ne va jamais les chercher sur le disque, ce dernier sert de support d'archivage aux informations, en effet la mémoire s'efface lorsqu'on coupe l'alimentation électrique de la machine.

La mémoire, ressource du S.E

- La mémoire est assemblage de cellules repérées par leur numéro, ou **adresse**.
- Gestionnaire de mémoire : gère l'allocation de l'espace mémoire au système et aux processus utilisateurs.



n = largeur du bus d'adresses
m = largeur du bus de données



2. Espace d'adressage d'un processus

2.1. Notion d'espace d'adressage

L'espace d'adressage d'un processus est constitué de l'ensemble des adresses auxquelles le processus a accès au cours de son exécution. Cet espace d'adressage est au moins constitué de trois parties qui sont le code et les donnés du processus ainsi que sa pile d'exécution.

Cet espace d'adressage est représenté par l'ensemble des adresse qui le constitue, dont la forme dépend de la structuration de cet espace. Cet ensemble d'adresse est appelé **espace d'adresses logiques ou virtuelles**.

2.2. Adresse logique et adresse physique

L'unité centrale manipule des **adresses logiques** (emplacement relatif).
Les programmes ne connaissent que des adresses logiques, ou virtuelles.
L'espace d'adressage logique (virtuel) est donc un ensemble d'adresses pouvant être générées par un programme.
L'unité mémoire manipule des **adresses physiques** (emplacement mémoire). Elles ne sont jamais vues par les programmes utilisateurs. L'espace d'adressage physique est un ensemble d'adresses physiques correspondant à un espace d'adresses logiques.

3. Pagination de la mémoire centrale

3.1. Principe de la pagination

Dans le mécanisme de pagination, l'espace d'adressage du programme est découpé en morceaux linéaires de même taille appelés **pages**. L'espace de la mémoire physique est lui-même découpé en morceaux linéaires de même taille appelés **cases** ou **cadres de pages**.

Dans ce contexte, charger un programme en mémoire centrale consiste à placer les pages dans n'importe quelle case disponible. Pour connaître à tout moment quelles sont les cases libres en mémoire centrale à un instant t, le système maintient une table appelée **table cadres de pages** ou **table des cases** qui indique pour chaque case de la mémoire physique, si la case est libre ou occupée, et si elle est occupée, quelle page et quel processus la possèdent.

La figure 3.1 donne un exemple d'application de ce mécanisme d'allocation pour deux processus P1 et P2 dont les espaces d'adressage sont respectivement égaux à 16 Ko et 7 Ko. Les pages et les cases ont une taille de 4 Ko.

3.2. Adresse logique et table des pages

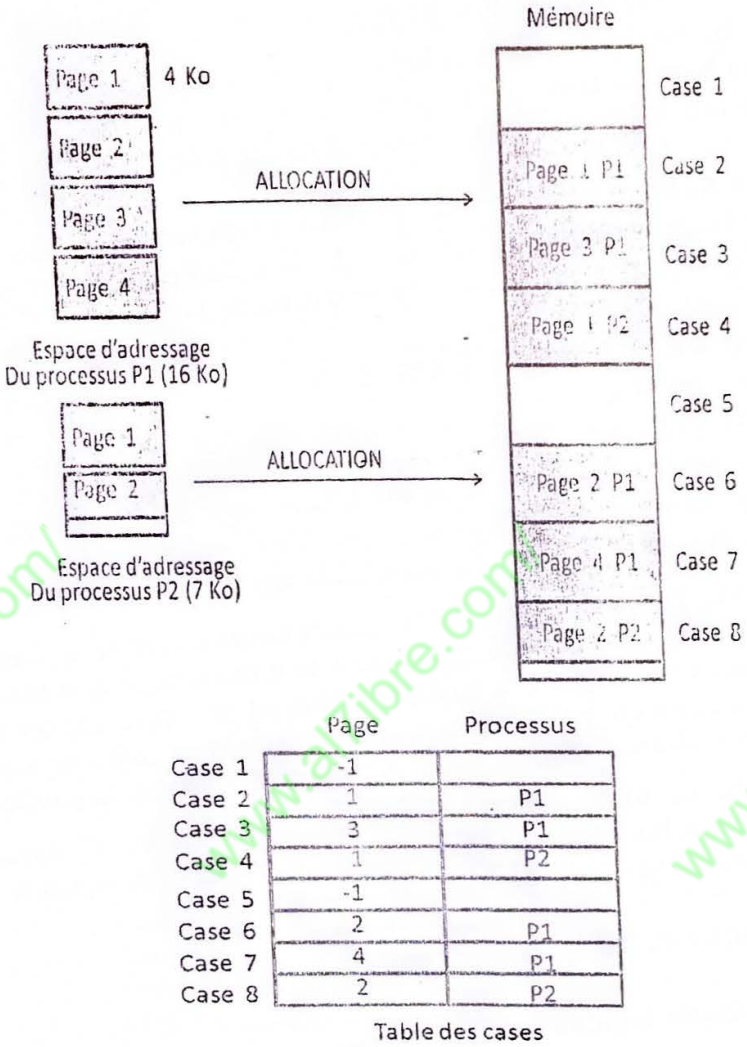
3.2.1. Conversion adresse logique- adresse physique

L'espace d'adressage du processus étant découpé en pages, les adresses générées dans cet espace d'adressage sont des **adresses paginées**, c'est-à-dire qu'un octet est repéré par son emplacement relativement au début de la page à laquelle il appartient.
L'adresse d'un octet est donc formée par le couple « numéro de page à laquelle appartient l'octet, déplacement relativement au début de cette page ».

Les octets dans la mémoire physique ne peuvent être adressés au niveau matériel que par leur adresse physique.

Pour toute opération concernant la mémoire, il faut donc convertir l'adresse paginée 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 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**.



	Page	Processus
Case 1	-1	
Case 2	1	P1
Case 3	3	P1
Case 4	1	P2
Case 5	-1	
Case 6	2	P1
Case 7	4	P1
Case 8	2	P2

Table des cases

FIG. 3.1 – Principe de pagination

3.2.2. La table des 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 la table est un couple « numéro de page, numéro de case physique dans laquelle la page est chargée ».

Dans l'exemple de la figure 3.2, le processus P1 a 4 pages dans son espace d'adressage, donc la table des pages a 4 entrées. Chaque entrée établit l'équivalence numéro de page, numéro de case relativement au schéma de la mémoire centrale. Le processus P2 a un espace d'adressage composé de 2 pages, donc sa table de pages a 2 entrées. Par ailleurs, l'espace d'adres... du processus P2 étant de 7 Ko, la page 2 du processus a une table égale à 3 Ko. Il s'en... un

phénomène de **fragmentation interne** en mémoire physique au niveau de la case qui n'est pas totalement occupée.

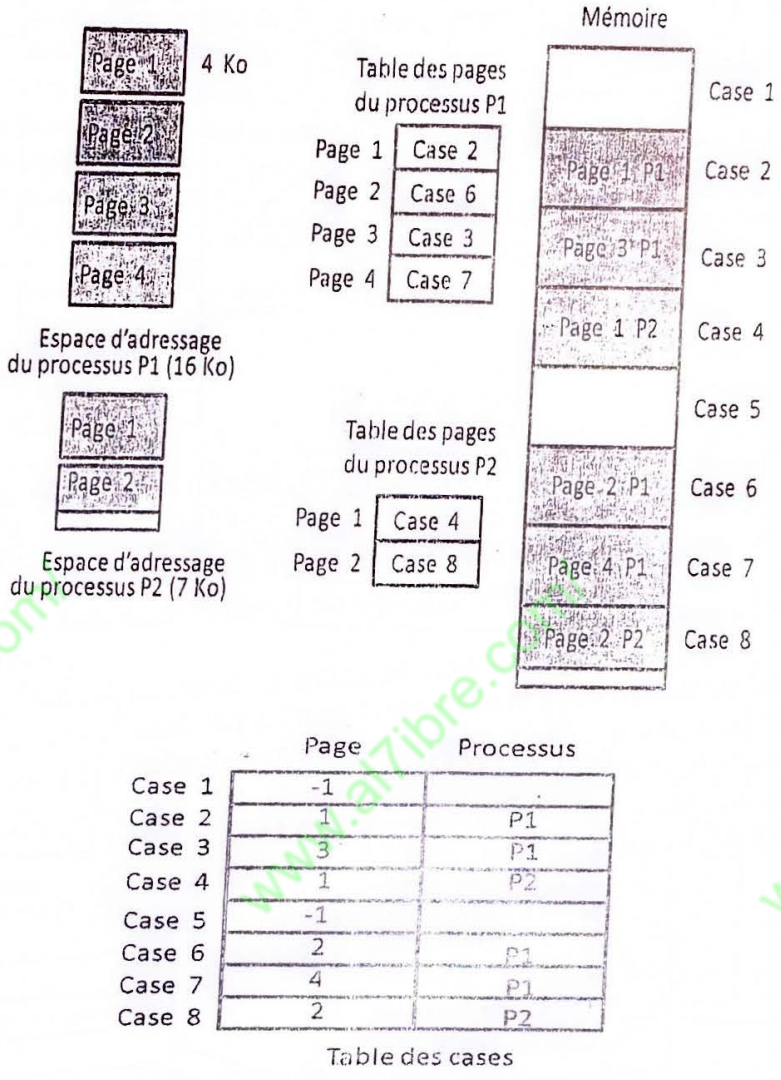


FIG. 3.2 – Table des pages

Deux approches existent pour la réalisation de la table des pages :

- à l'aide de registres du processeur MMU: la table des pages est sauvegardée avec le contexte processeur dans le PCB (Process Control Block) du processus.
- placer les tables des pages en mémoire centrale : la table active est repérée par un registre spécial du processeur le PTBR (page-table base register).

Accéder à un emplacement mémoire à partir d'une adresse paginée <p,d> nécessite deux accès à la mémoire (figure 3.3) :

- un premier accès permet de lire l'entrée de la table des pages correspondant à la page cherchée et délivre une adresse physique c de la case dans la mémoire centrale.
- un second accès est nécessaire à la lecture ou l'écriture de l'octet recherché à l'adresse c+d.

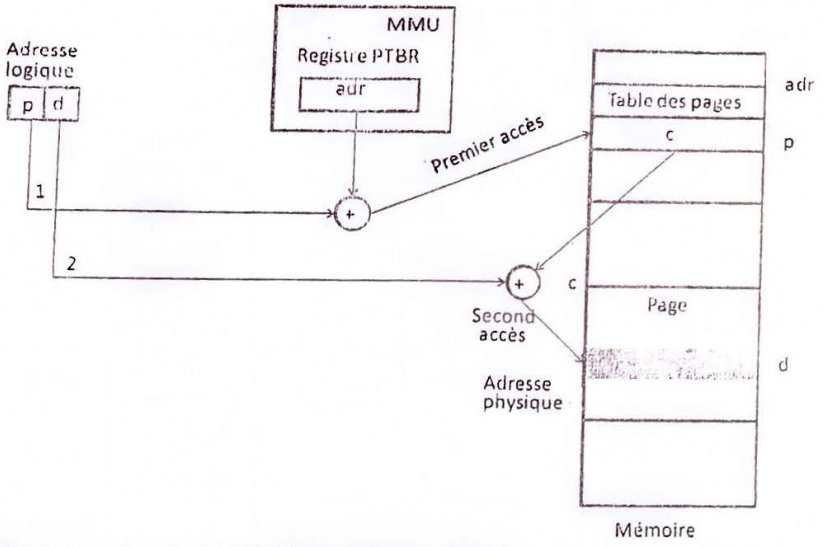


FIG. 3.3 – Traduction d'une adresse paginée en adresse physique

Pour accélérer les accès à la mémoire centrale et compenser le coût lié à la pagination :

- un cache associatif (TLB look-aside buffers) est placé en amont de la mémoire centrale qui contient un couple <numéro de page, adresse d'implantation de la case> les plus récemment accédés.
- la MMU cherche tout d'abord dans le cache si la correspondance numéro de page, adresse d'implantation de la case recherchée n'est pas dans le cache.
- Si non, elle accède à la table des pages en mémoire centrale et place le nouveau couple référencé dans le cache.
- Si oui, elle effectue directement la conversion : un seul accès mémoire est alors nécessaire pour accéder à l'octet recherché (figure 3.4).

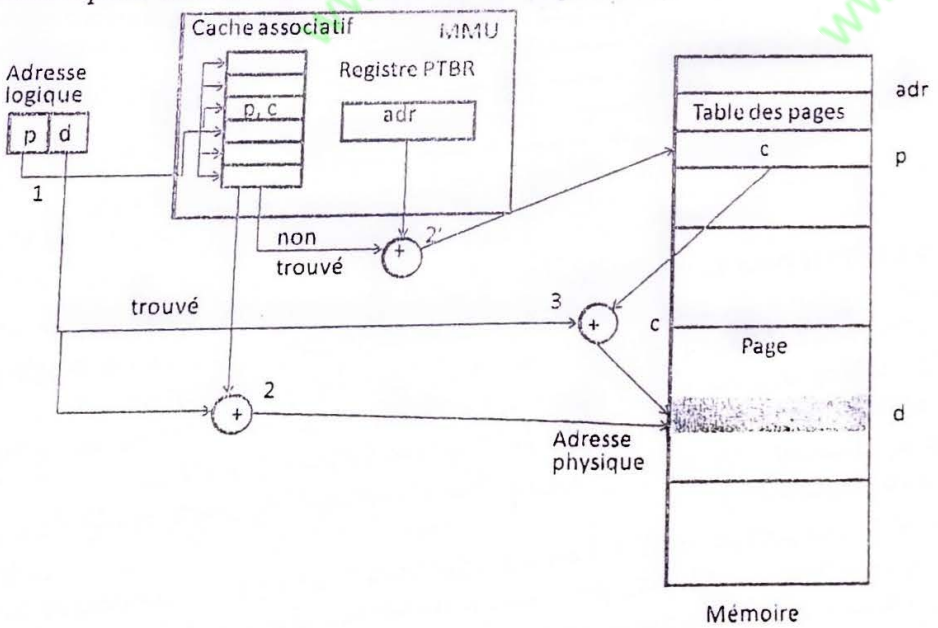


FIG. 3.4 – Traduction d'une adresse paginée en adresse physique avec ajout d'un cache associatif

3.3. Protection de l'espace d'adressage des processus

Des bits de protection sont associés à chaque page de l'espace d'adressage du processus et permettent ainsi de définir le type d'accès autorisés à la page. Ces bits de protection sont mémorisés pour chaque page, dans la table des pages du processus.

Classiquement, 3 bits sont utilisés pour définir respectivement l'autorisation d'accès en lecture (r), écriture (w) et exécution (x).

Il peut être souhaitable que des pages soient accessibles par plusieurs processus.

Pour que deux processus puissent partager un ensemble de pages, il faut que chacun des deux processus référence cet ensemble de pages dans sa table des pages respective. Ainsi, sur la figure 3.5, Les processus P1 et P2 référencent tous les deux la même page 1, situé dans la case 2 de la mémoire centrale.

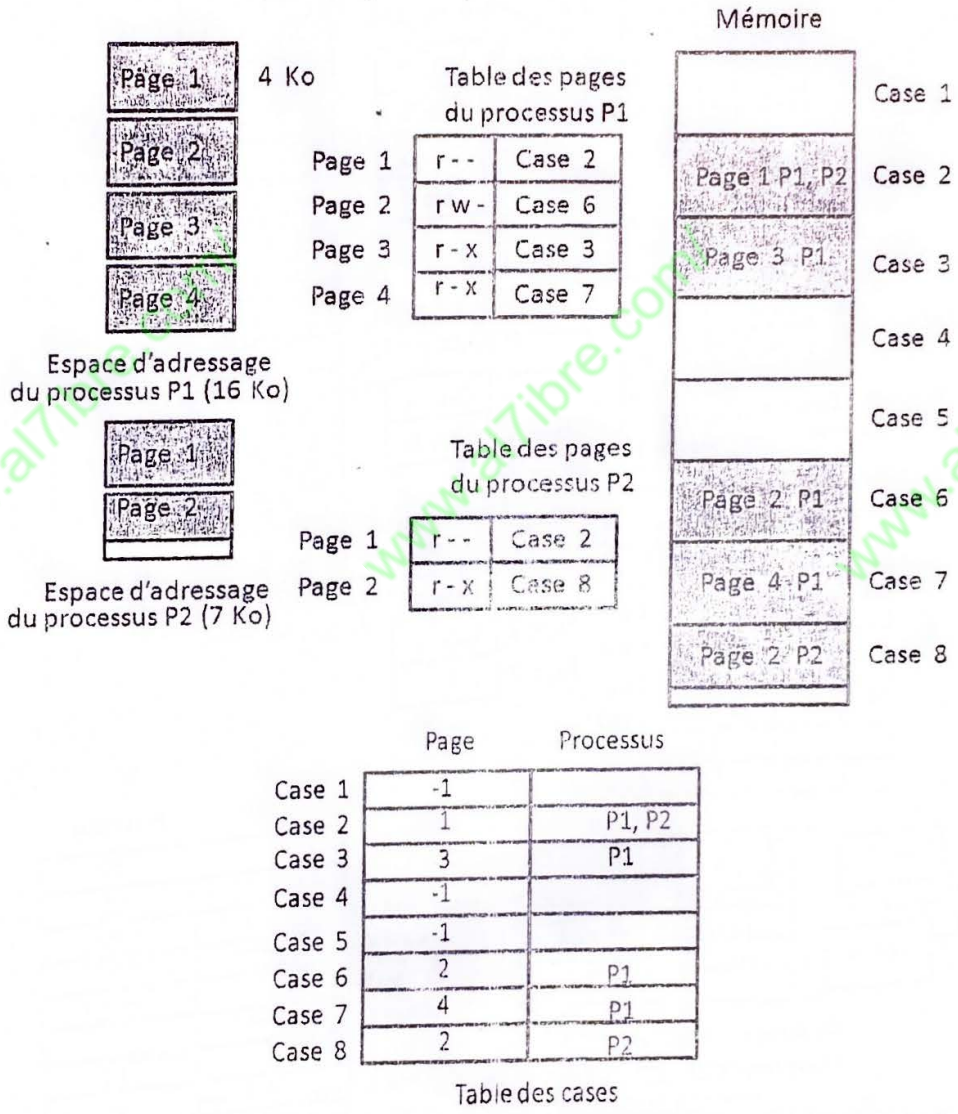


FIG. 3.5 – Entrées de la table des pages avec les bits de protection et partage des pages entre processus

4. La mémoire virtuelle

4.1. Principe de la mémoire virtuelle

La multiprogrammation implique de charger plusieurs programmes en mémoire centrale de manière à obtenir un bon taux d'utilisation du processeur.
Supposons comme sur la figure 4.1 que l'exécution des processus P1, P2 et P3 soit nécessaire pour obtenir ce taux d'utilisation du processeur satisfaisant. On peut remarquer qu'une fois les pages des processus P1 et P2 chargés dans la mémoire, toutes les cases sont occupées : le programme P3 ne peut pas être chargé.

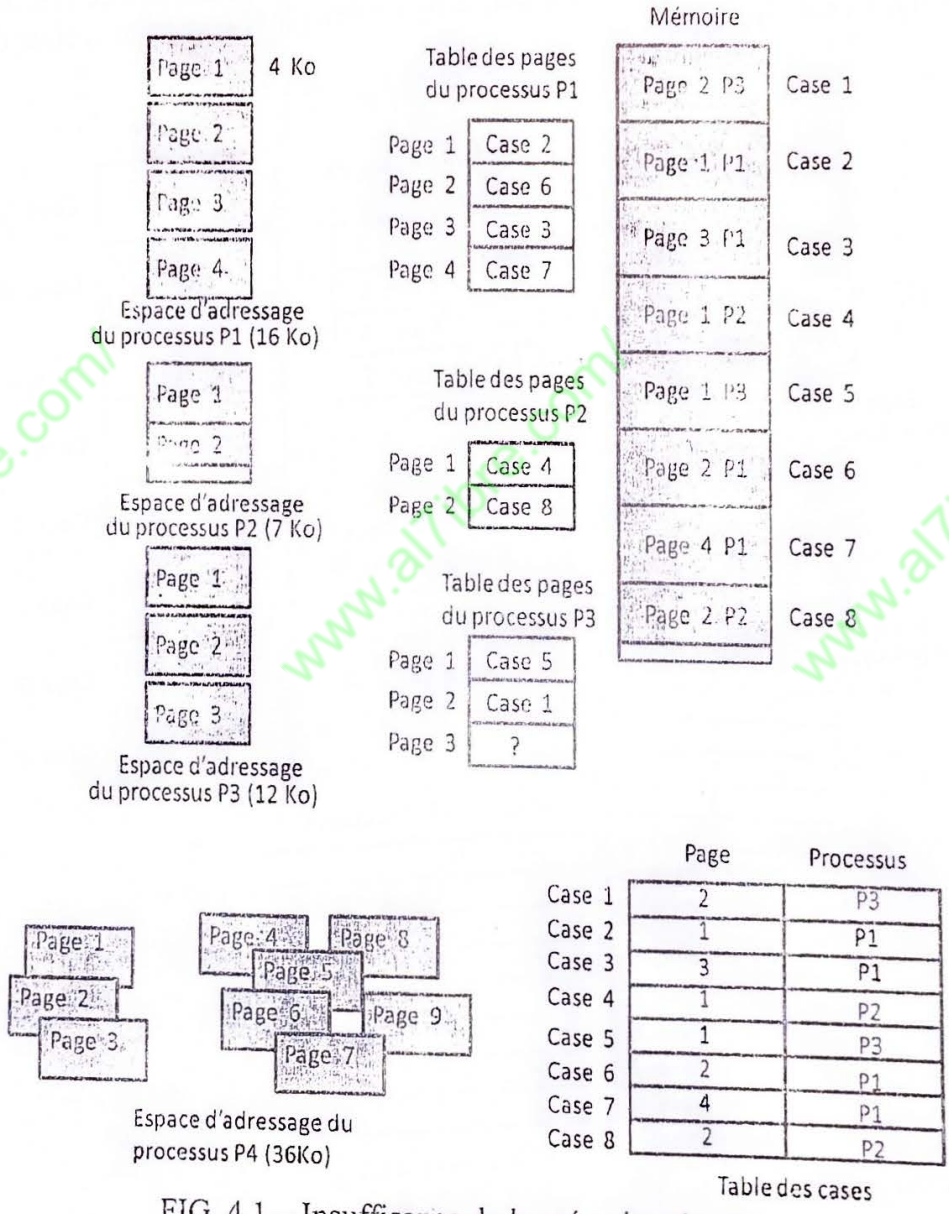


FIG. 4.1 – Insuffisance de la mémoire physique

Le processus n'accède qu'à une partie de son espace d'adressage, les autres pages de l'espace d'adressage ne sont pas accédées et sont donc inutiles en mémoire centrale.

Une solution pour pouvoir charger plus de programmes dans la mémoire centrale est donc de ne charger pour chaque programme que les pages couramment utilisées.
Ainsi, sur la figure 4.2, seules les pages 1 et 3 du processus P1 sont chargées ainsi que la page 1 du processus P2, et les pages 1 et 2 du processus P3.
Par ailleurs, on remarquera sur cette même figure, que l'espace d'adressage du processus P4 est plus grand que la mémoire physique disponible pour les programmes utilisateurs.

Le principe de la **mémoire virtuelle**, qui consiste à ne charger à un instant donné en mémoire centrale que la partie couramment utile de l'espace d'adressage des processus.

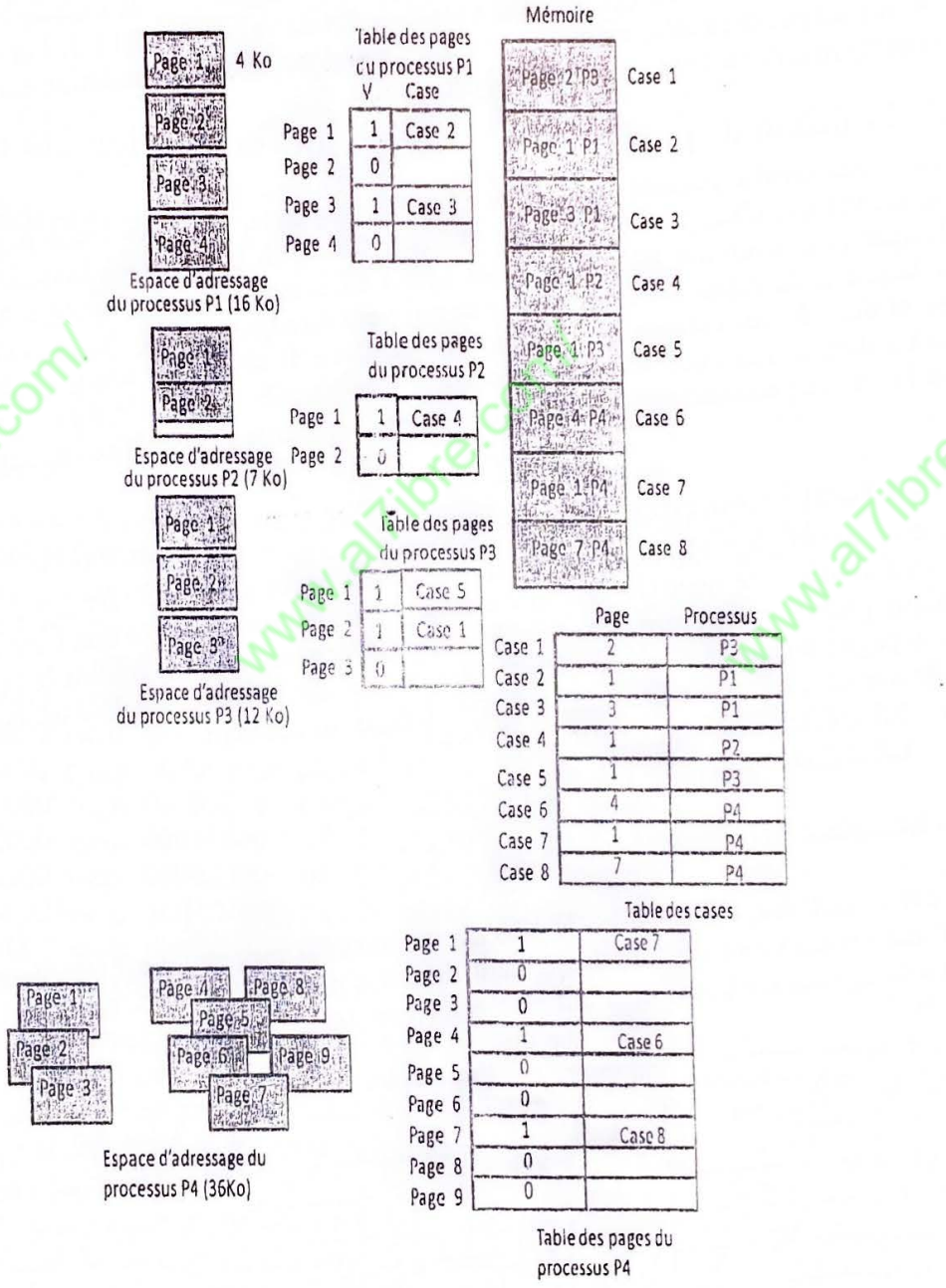


FIG. 4.2 – Mémoire virtuelle

Puisque les pages d'un espace d'adressage de processus ne sont pas toutes chargées en mémoire centrale, il faut que le processeur puisse détecter leur éventuelle absence lorsqu'il cherche à effectuer une conversion d'adresse paginée vers l'adresse physique. Chaque entrée de la table des pages comporte alors un champ supplémentaire, le bit validation V, qui est vrai (1 ou V) si la page est effectivement présente en mémoire centrale, 0 sinon.

La figure 4.2 montre les valeurs des bits de validation pour les tables des pages des quatre processus P1, P2, P3 et P4, en tenant compte des chargements de leurs pages en mémoire centrale. Ainsi pour le processus P1, la page 1 est chargée dans la case 2, le bit de validation est à 1. La page 3 est chargée dans la case 3, le bit de validation est à 1. Par contre les pages 2 et 4 ne sont pas présentes en mémoire centrale et donc le bit de validation est à 0 : dans ce cas, le champ numéro de case n'a pas de signification.

4.2. Le défaut de pages

Que se passe-t-il à présent lorsque qu'un processus tente d'accéder à une page de son espace d'adressage qui n'est pas en mémoire centrale ? La MMU accède à la table des pages pour effectuer la conversion adresse logique vers l'adresse physique et teste la valeur du bit de validation. Si la valeur du bit V est à 0 ce qui veut dire que la page n'est pas chargée dans une case et donc la conversion ne peut pas être réalisée. Il se produit alors un **défaut de page** : c'est un déroutement qui oblige le processeur à suspendre l'exécution du programme en cours pour lancer une entrée/sortie qui charge la page manquante en mémoire centrale dans une case libre.

4.3. Le remplacement de pages

Lors d'un défaut de page, la page manquante est chargée dans une case libre. La totalité des cases de la mémoire centrale peut être occupée : il faut donc libérer une case de la mémoire physique globalement (parmi l'ensemble des cases) ou localement (parmi les cases occupées par les pages du processus en défaut). Le système d'exploitation utilise un algorithme pour choisir une case à libérer. Les deux principaux algorithmes sont :

- FIFO (First In, First Out)
- LRU (Least Recently Used)

4.3.1. Remplacement FIFO

Avec cet algorithme, c'est la page la plus anciennement chargée qui est remplacée. La figure 4.3 donne un exemple du fonctionnement de cet algorithme où l'on suppose une mémoire centrale composée de trois cases initialement vides. La lettre D signale l'occurrence de défaut de pages.

Chaine de référence	8	1	2	3	1	4	1	5	3	4	1	4	3
Case 1	8	8	8	3	3	3	3	5	5	5	1	1	1
Case 2		1	1	1	1	4	4	4	3	3	3	3	3
Case 3			2	2	2	2	1	1	1	4	4	4	4
Défaut	D	D	D	D		D	D	D	D	D	D		

FIG. 4.3 – Remplacement FIFO

4.3.2. Remplacement LRU

Avec cet algorithme, c'est la page la moins récemment accédée qui est remplacée.

Chaine de référence	8	1	2	3	1	4	1	5	3	4	1	4	3
Case 1	8	8	8	3	3	3	3	5	5	5	1	1	1
Case 2		1	1	1	1	1	1	1	1	4	4	4	4
Case 3			2	2	2	4	4	4	3	3	3	3	3
Défaut	D	D	D	D		D		D	D	D	D		

FIG. 4.4 – Remplacement LRU

5. Gestion de mémoire sous Linux

5.1. Espace d'adressage d'un processus Linux

L'espace d'adressage d'un processus Linux est composé de cinq régions principales :

- Zone de code,
- Zone des données initialisées,
- Zone des données non initialisées
- Zone contenant le tas
- Zone de la pile.

On peut obtenir la liste des régions virtuelles d'un processus en utilisant le système de fichiers /proc. Un cat /proc/[pid]/maps (man proc) donnera la liste des régions du processus [pid].
Exemple pour un processus cat :

```
etudiant> cat /proc/self/maps
# address              perms   offset     dev    inode   pathname
08048000-0804b000 r-xp  00000000  08:02   52439  /bin/cat
0804b000-0804c000 rw-p  00003000  08:02   52439  /bin/cat
0804c000-0804d000 rwxp  00000000  00:00   0
40000000-40011000 r-xp  00000000   08:02  16236  /lib/ld-2.3.1.so
40011000-40012000 rw-p  00011000   08:02  16236  /lib/ld-2.3.1.so
40026000-4012e000 r-xp  00000000   08:02  16241  /lib/libc-2.3.1.so
4012e000-40134000 rw-p  00107000   08:02  16241  /lib/libc-2.3.1.so
40134000-40137000 rw-p  00000000   00:00  0
bfffe000-c0000000   rwxp  fffff000     00:00  0
```

La première région est le code du programme exécuté, son adresse de début est indiquée dans l'exécutable, et en général est située à l'adresse virtuelle 0x08048000 sur x86 (info ld). Cette région est en lecture et exécution seulement.
La deuxième région contient les données du programme exécuté (en lecture et écriture). La région qui suit est le tas.
Ensuite, on trouve le code et les données de toutes les bibliothèques partagées dont le programme a besoin. La première bibliothèque partagée, ld-2.3.1.so, est chargée en premier, et dans la première région disponible à partir de l'adresse virtuelle 0x40000000 sur x86.

Cette bibliothèque est le chargeur de bibliothèques dynamiques : elle va charger en mémoire les autres bibliothèques. Les autres régions sont le code et les données des bibliothèques partagées. La dernière région correspond à la pile.

La zone de la pile est visible uniquement à l'exécution, et elle contient les variables locales et les arguments des fonctions, entre autres. Chaque processus sur une machine 80x86 de 32-bits a 3 Go d'espace d'adressage virtuel, le Go restant est réservé aux Tables de pages et certaines données du Kernel du système d'exploitation.

Commande vmstat

La commande **vmstat** fourni des renseignements et des statistiques sur l'usage de la mémoire virtuelle du système :

```
etudiant~ vmstat
procs      memory                    swap    io      system      cpu
r b w    swpd free    buff   cache   si  so  bi  bo    in   cs    us sy  id
1 0 0    1480 79980 20192   99232    0   0   3   4   226   90    88  1  11
```

où (d'après man vmstat) :

procs

- r: Processus en attente d'UCT.
- b: Nombre de processus en sleep.
- w: Nombre de processus swapped out mais prêts.

memory

- swpd : Quantité de mémoire virtuelle utilisée (Ko).
- free : Quantité de mémoire libre (Ko).
- buff : Quantité de mémoire utilisée comme buffers (Ko).

swap

- si : Quantité de mémoire swapped in depuis le disque (Ko/s).
- so : Quantité de mémoire swapped au disque (Ko/s).

io

- bi : Blocs envoyés (blocks/s).
- bo : Blocs reçus (blocks/s).

system

- in : Interruptions à la seconde, incluant l'horloge.
- cs : Changements de contexte à la seconde.

cpu (Pourcentages de l'utilisation de l'UCT)

- us : Temps utilisateur.
- sy : Temps du système.
- id : Temps non utilisé.

Exemple :

Lors de l'exécution d'un programme, l'écriture des pointeurs (format « %p » du printf en C) fournit des adresses virtuelles comme l'indiquent les résultats de l'exécution du programme ad_virt.c suivant. Le compilateur sépare les différents objets dans des zones de mémoires virtuelles (code, données, librairies, pile).

```
/* ad_virt.c pour écriture des adresse virtuelles */
```

```
#include <stdio.h>
#include <stdlib.h>
int main (void) {
short int i=0;
char c='a';
short int j=1;
char *pc=&c;
short int *p=&i;
short int *q=&j;
int r;
r=p-q;
printf("pc = %p \n",pc);
printf("p = %p \n",p);
printf("q = %p \n",q);
printf("r = %d \n",r);
return 0 ;
}
```

Résultats de l'exécution du programme ad_virt.c:

```
etudiant>gcc -o ad_virt ad_virt.c
etudiant>./ad_virt
pc = 0x8004a024
p = 0x8004a024
q = 0x8004a26
r = 2
```

5.2. Projection d'un fichier en mémoire centrale

Les primitives **mmap()** et **mumap()** permettent respectivement de projeter un fichier en mémoire centrale et de libérer cette projection. Les prototypes de ces fonctions sont :

```
#include<unistd.h>
#include <sys/mman.h>
void  *mmap(void *debut, size_t longueur, int prot, int flags, int desc, off_t offset);
int munmap(void *debut, size_t longueur);
```

debut: indique l'adresse où on projettera le fichier. Normalement on met la valeur NULL pour laisser au système le choix.

longueur: indique le nombre d'octets à projeter en mémoire.

prot: spécifie le type de protection à appliquer à la région. Il peut prendre l'une des 4 valeurs suivantes :

- PROT_NONE, la région est marquée comme étant inaccessible.
- PROT_READ, la région est accessible en lecture,
- PROT_WRITE, la région est accessible en écriture,
- PROT_EXEC, la région est accessible en exécution.

desc : descripteur du fichier.

offset: permet de définir la position dans une zone mémoire.

flags : permet de spécifier certaines propriétés affectées à la région. Ce paramètre peut prendre l'une des valeurs suivantes :

- MAP_SHARED: Zone partagée. Les modifications vont affecter le fi . Un processus fils partage cette zone avec le père.
- MAP_PRIVATE: Zone privée. Les modifications n'affectent pas le fichier. Un processus fils ne partage pas cette zone avec son père, il obtient un dupliqué.
- MAP_FIXED: Le fichier va se projeter exactement dans l'adresse spécifiée par le premier paramètre **debut** s'il est différent 0.

Exemple : Programme « proj_fich.c » qui projette un fichier en mémoire et lit son conten... Le fichier contient 4 enregistrements de type correspondant.

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<sys/mman.h>
main(){
struct correspondant {
char nom[10];
char telephone[10];
};
int fd,i;
struct correspondant *un_correspondant;
c..r *adresse, nom[10];
fd=open("/home/etudiant/tp_SE2/fichnoms",O_RDWR);
if(fd==-1)
perror("prob open");
adresse=(char*)mmap(NULL,4*sizeof(struct
correspondant),PROT_READ,MAP_SHARED,fd,(off_t)0);
close(fd);
// les données du fichier sont accédées à la manière d'un tableau
// dont le premier élément est situé au premier-octets de la région
i=0;
while(i<4*sizeof(struct correspondant)){
putchar(adresse[i]);
i=i+1;}}
```

5.3. Allocation de mémoire dynamique

Les fonctions de la famille **malloc()** permettent d'allouer ou de libérer de la mémoire dynamique. Leurs prototypes sont :

```
#include<stdlib.h>
void *malloc(size_t taille) ;
void *calloc(size_t nombre, size_t taille) ;
void *realloc(void *adresse, size_t taille) ;
void free(void *adresse) ;
```

La fonction **malloc()** alloue une zone mémoire de **taille** octets. Elle retourne en cas de succès un pointeur sur la zone allouée et la valeur NULL sinon.

La fonction **calloc()** alloue un tableau de nombre zones mémoire de **taille** octets. Elle retourne en cas de succès un pointeur sur la zone allouée et la valeur NULL sinon.

La fonction **realloc()** permet de modifier la taille d'une zone préalablement allouée par un appel à la fonction **malloc()**. Le paramètre **adresse** correspond à l'adresse de la zone mémoire retournée par la fonction **malloc()** et **taille** spécifie la nouvelle taille de la zone. La fonction retourne en cas de succès un pointeur sur la nouvelle zone allouée et la valeur NULL sinon.

La fonction **free()** libère une zone mémoire dont l'adresse est donnée par le paramètre adresse.

Exemple : Le programme « cre_stock_fich.c » crée un fichier et stocke des données de type correspondant dans le fichier.

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/mman.h>
main()
{
struct correspondant {
char nom[10];
char telephone[10];
};
int fd,i;
struct correspondant *un_correspondant;
fd=open("/home/etudiant/tp_SE2/fichnoms",O_RDWR|O_CREAT,S_IRUSR|S_IWU
SR);
if(fd==-1)
perror("prob open");
i=0;
while(i<4){
un_correspondant=(struct correspondant*) malloc(sizeof(struct correspondant));
printf("Donnez un nom: \n");
scanf("%s",un_correspondant->nom);
printf("donnez un numéro de téléphone: \n");
scanf("%s",un_correspondant->telephone);
write(fd,un_correspondant,sizeof(struct correspondant));
i=i+1;}
close(fd);}
```