Université Sidi Mohamed Ben Abdellah

Faculté des sciences

Filière SMI S4

Systèmes d'exploitation II

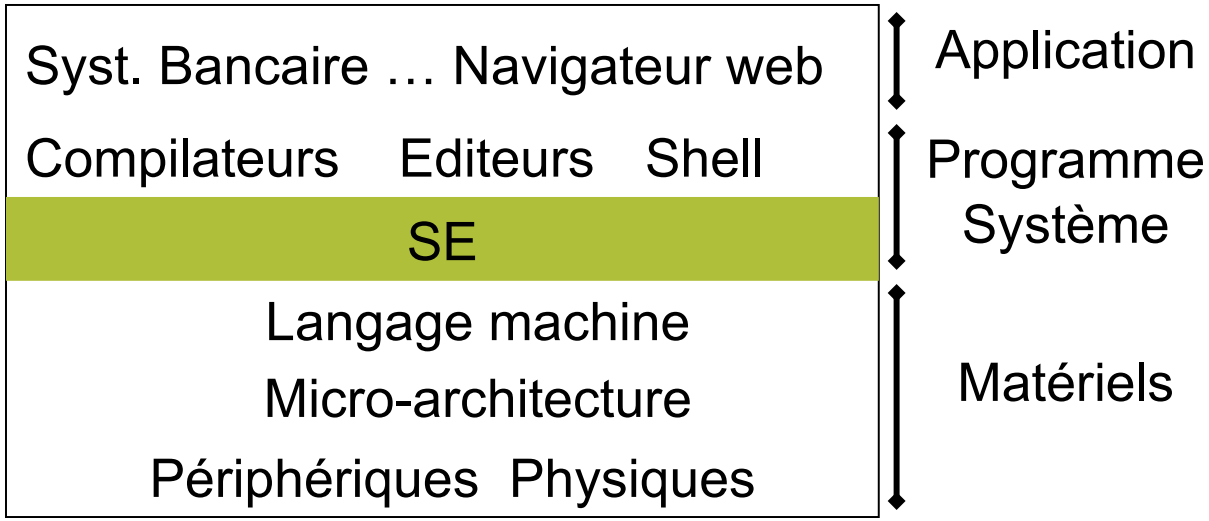
Enseignante: Fatima EL HAOUSSI

1 INTRODUCTION AUX SE

- Qu'est ce qu'un système d'exploitation?
- Les fonctionnalités du SE
- Les différents composants
- Les différentes classes de SE
- Présentation d'Unix
- Caractéristiques
- Le noyau Unix

Qu'est ce qu'un système d'exploitation?

- Le système d'exploitation peut être noté SE ou OS ("Operating System" en anglais).
- Deux visions :
  - Servir d'interface entre l'utilisateur et le matériel.
  - Un gestionnaire de ressources : un programme qui gère les ressources de l'ordinateur (processeur, mémoire, périphériques, etc…).



Les fonctionnalités du SE

- Gestion:
  - des processus
  - de la mémoire
  - des périphériques de stockage auxiliaires
  - des Entrées Sorties (E/S)
  - des fichiers
  - …
- Sécurité
- Réseau
- Interpréteur de commandes

Les composantes du SE

Le SE est composé d'un ensemble de logiciels permettant de gérer les interactions avec le matériel.

- **Le noyau** (en anglais kernel):
  - Une partie importante du système d'exploitation.
  - Une interface entre le logiciel et le matériel de l'ordinateur.
  - La gestion des ressources de l'ordinateur et des fonctionnalités de communication.
- Type de noyau:
  - Les **noyaux monolithiques** placent un maximum de programmes systèmes dans l'espace noyau.
  - Les **micro-noyaux** préfèrent au contraire placer le plus de choses dans l'espace utilisateur.
  - Les **noyaux hybrides**, sont un intermédiaire entre les deux précédents.

Les composantes du SE (suite)

- **L'interpréteur de commande** (en anglais shell):

  • permet la communication avec le système d'exploitation par l'intermédiaire d'un langage de commandes.

  • Avantages par rapport aux environnements graphiques:

  - Précision et simplicité d'automatisation des tâches (mode batch)
  - Contrôle à distance
  - Uniformité
  - Stabilité
  - Faible de consommation des ressources.

Les composantes du SE (suite)

- **Le système de fichiers** (en anglais "file system"):
  - Stocker les informations et les organiser dans des fichiers sur des mémoires secondaires (disque dur, CD-ROM, clé USB, disquette…).
  - Il offre à l'utilisateur une vue abstraite sur ses données et permet de les localiser à partir d'un chemin d'accès.
  - Un fichier est identifié par un nom qui a la forme suivante :

    nom_du_fichier.extention
  - L'extension sert à regrouper les fichiers de même nature.
  - Les fichiers peuvent être regroupés dans des répertoires.

Les différentes classes du SE

- **Systèmes multi-tâches:**
  - L'exécution de plusieurs processus en même temps.
- **Systèmes multi-processeurs:**
  - Plusieurs processeurs reliés au bus de l'ordinateur.
  - Ils se caractérisent par leur capacité de traitement et leur fiabilité.
  - La panne d'un processeur n'arrêtera jamais le système.
- **Systèmes multi-utilisateurs:**
  - Capacité à pouvoir gérer un panel d'utilisateurs utilisant simultanément les mêmes ressources matérielles.
- **Systèmes temps réel:**
  - Essentiellement utilisés dans l'industrie.
  - Respect des contraintes temporelles.

Les différentes classes du SE (suite)

- **Systèmes embarqués :**
  - Systèmes électroniques et informatique autonomes.
  - Spécialisés dans une tâche bien précise.
  - Caractéristiques:

    gestion avancée de l'énergie.
    capacité à fonctionner avec des ressources limitées.
  - Domaines d'applications:

    Equipement médical
    Guichet automatique bancaire
    Télécommunication: téléphone portable
    Transport: automobile
- **Systèmes distribués :**
  - Permettre l'exécution d'un seul programme sur plusieurs machines,

    distribuer les processus et les remettre ensemble.
  - Ils sont pour gros calculs,
  - Exemple: inversion de grandes matrices.

Présentation d'Unix

Historique de Unix/ Linux

1969 : Ken Tompson écrit la première version d'Unix.
1984 : développement de Système 1 d'Apple.
1991 : Naissance de Linux LinusThorvalds conçu pour PC et Internet.
1992 : Développement de FreeBSD qui est un système Unix.
Après 1992 : apparition de multiples **distributions** de Linux (**Slackware** et **Debian** en 93, **SuSE** en 94, **RedHat** en 95 ...)
2001 : Mac OS X système d'exploitation d'Apple qui est un système Unix et qui est en partie dérivé de FreeBDS.

Caractéristiques

Unix

• Un système interactif et multi-utilisateurs.
• Un système multi-tâches, ce qui signifie que plusieurs programmes peuvent s'exécuter en même temps sur la même machine .
• Un système d'exploitation ouvert, portable et disponible sur différentes plate-formes.

Linux

• Linux est gratuit et on a le droit d'étudier et de modifier le code source.

Le noyau Unix

Le **noyau** est le programme qui assure :

- la gestion de la mémoire,
- le partage du processeur entre les différentes tâches à exécuter,
- le partage des entrées/sorties de bas niveau.

Il est lancé au démarrage du système (le *boot*) et s'exécute jusqu'à son arrêt. C'est un programme relativement petit, qui est chargé en mémoire principale.

Le noyau Unix (suite)

Il s'exécute en mode **superviseur** (maître), c'est-à-dire que toutes les instructions sont autorisées.

Tous les autres programmes qui s'exécutent sur la machine fonctionnent en mode **utilisateur** (esclave): ils leur sont interdits d'accéder directement au matériel et d'utiliser certaines instructions.

2 INTRODUCTION A LA PS

- Qu'est ce que la programmation système?
- Les bases de la PS
- Outils de développement
- Éditeurs de texte
- La compilation sous Linux
- Gestion des erreurs
- Installer gcc
- Programmer et compiler sous Linux

Qu’est-ce que la programmation système?

- Les **programmes d'application:** sont réalisés lors de la programmation classique, celle que nous avons fait par exemple sur le langage C.

- Les **programmes systèmes** 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...

Les bases de la programmation système?

Tous les programmes qui fournissent des services au système d'exploitation Unix 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.
- ❑ accès aux informations du système.

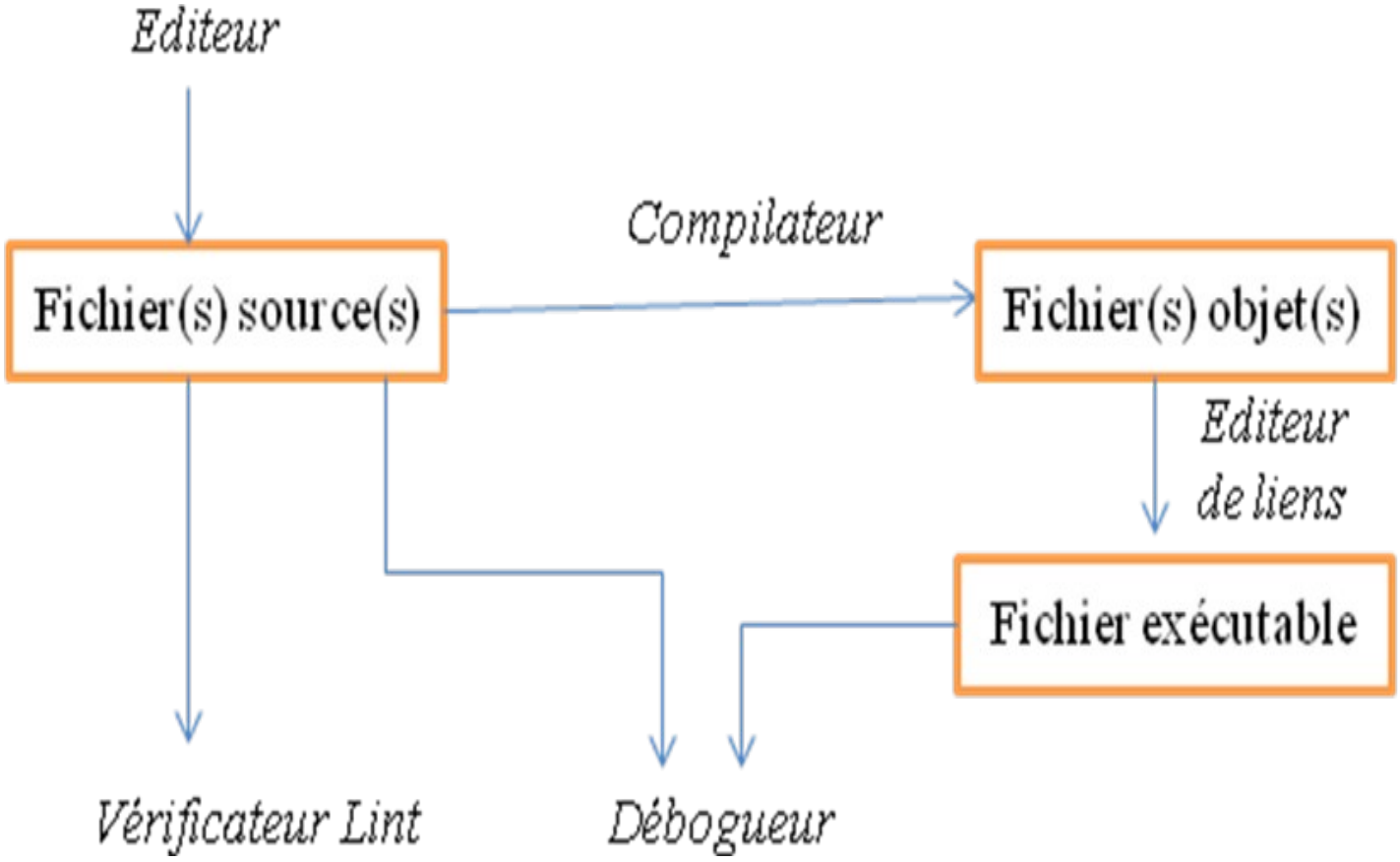
Pour accéder à ces informations, nous allons utiliser des fonctions qui permettent de communiquer avec le noyau. Ces fonctions s'appellent des **appels-systèmes**.

Les **appels systèmes** sont directement appelables depuis un programme.

Outils de développement

- L'éditeur de texte, qui nous permet de créer et de modifier les fichiers source.
- Le compilateur, qui permet de passer d'un fichier source à un fichier objet.
- 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 un fichier exécutable.
- Le débogueur, qui peut 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.

Outils de développement (suite)



Éditeurs de texte

Emacs:

c’est une énorme machine capable d'offrir l'essentiel des commandes dont un développeur peut rêver.

Vi :

- Il est plus léger
- Il offre moins de fonctionnalités et de possibilités d'extensions qu’Emacs.

Les avantages de Vi:

- La disponibilité sur toutes les plates formes Unix
- la possibilité de l'utiliser même sur un système très réduit pour réparer des fichiers de configuration.

La compilation sous Unix

Le compilateur C utilisé sous Linux est gcc (Gnu C Compiler).

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.

La compilation sous Unix (suite)

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

Les options les plus utilisées:

- Le chemin de recherche des bibliothèques supplémentaires, précédé de l'option –L.
- Le nom d'une bibliothèque supplémentaire à utiliser lors de l'édition des liens, précédé du préfixe -l. Par exemple la commande –lm permet d'inclure le fichier libm.so indispensable pour les fonctions mathématiques.
- Le nom du fichier exécutable, précédé de l'option –o.

La compilation sous Unix (suite)

Options :

- Le chemin de recherche des fichiers d'en-tête, 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
# avec la librairie math en dynamique
gcc -o mon_fichier mon_fichier.c -lm
# avec la librairie math en statique
gcc -o mon_fichier mon_fichier.c -static -lm
```

Gestion des erreurs

La variable globale errno

- **S**ignaler une erreur, les fonctions renvoient une valeur spéciale qui est généralement **-1**.
- Alerter l'appelant de la survenance d'une erreur.
- Cette variable est définie dans <errno.h> comme suit :

```
#include<errno.h>
extern int errno ;
```

La fonction perror

- Permettre une description de l'erreur qui s'est produite.

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

Installer gcc

- Ouvrir un terminal.
- Passer en mode super-utilisateur.

smi@smi-desktop:~$ sudo su
Password: <- tapez votre password utilisateur
root@smi-desktop:~#

- Remarque que l'invite est # à la place de $, tapez :

root@smi-desktop:~# apt-get install gcc-4.7

- Taper ensuite ctrl-d ou exit pour retrouver le terminal utilisateur (invite $)

smi@smi-desktop:~$

- gcc est maintenant complètement installé.

Programme C sous Ubuntu

- On va essayer un premier programme.
- Créer un répertoire par la commande **mkdir tpc** et placer vous dans ce répertoire

smi@smi-desktop:~$ cd tpc

- Lancer un éditeur vi par la commande vi hello.c
- Taper le programme suivant dans l'éditeur vi

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

Compiler C sous Ubuntu

On peut maintenant le compiler par gcc:

smi@smi-desktop:/~$ gcc -o hello hello.c
smi@smi-desktop:/~/tpc$

Lancer le fichier exécutable hello :

smi@smi-desktop:/~/tpc$ ./hello
Hello World!
smi@smi-desktop:/~/tpc$

3 PROCESSUS ET THREADS

- Introduction aux processus
- Identification des processus
- Création de processus
- Terminaison d'un processus
- Synchronisation entre père et fils
- Arguments en ligne de commande
- Variables d'environnement
- L'appel système exec
- Qu'est ce qu'un thread ?
- Primitives de gestion des threads
- Création, Suspension et terminaison d'un thread
- Ordonnancement

Qu’est ce qu’un processus?

- Un processus est un programme en cours d'exécution dans le système.
- Chaque processus peut être identifié par son numéro de processus, ou PID (Process IDentifier).
- Chaque processus (le processus fils) doit être lancé par un autre (le processus père). La racine de cette **hiérarchie** est le **programme initial**.
- Le **processus inactif** du système lance le premier processus (le **programme initial**) que le noyau exécute, et il a le PID 0.
- Le **programme initial** se nomme **init**, et il a le PID 1.
- Les **démons** (les processus système, le terme anglais daemon) sont en activité et fournit des services au système.

Les états d'un processus

Exécution (**R** pour running) : le processus est en cours d'exécution ;

Sommeil (**S** pour sleeping) : Endormi, en attente d'un évènement, comme la fin d'une entrée sortie (écriture sur un terminal, …).

Arrêt (**T** pour stopped) : le processus a été temporairement arrêté par un signal.

Zombie (**Z** pour … zombie) : le processus s'est terminé, mais son père n'a pas encore lu son code de retour.

Implémentation des processus

Le SE utilise une **table des processus** qui comprend une entrée par processus, c'est le **bloc de contrôle du processus** (Process Control Block, souvent abrégé **PCB**).

Ce bloc contient 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.

Gestion des processus

❑ Pour afficher ses propres processus en cours d'exécution, on utilise la commande

```
$ ps
   PID  TTY          TIME     CMD
21693   pts/8      00:00:00     bash
21694   pts/8      00:00:00     ps
```

• Le premier processus « bash » est le shell s'exécutant au sein du terminal.
• Le second est l'instance de ps en cours d'exécution.
• La première colonne, PID, indique l'identifiant de chacun des processus.

Gestion des processus (suite)

❑ Pour afficher tous les processus en cours d'exécution, on peut utiliser l'option **aux** (a : processus de tous les utilisateurs, u : affichage détaillé, x : démons).
$ ps –aux

• **USER** correspond à l'utilisateur qui a lancé le processus.
• **PID** indique le numéro de PID du processus.
• **STAT** correspond à l'état du processus.
• **START** correspond à l'heure du lancement du processus.
• **COMMAND** correspond au chemin complet de la commande lancée par l'utilisateur .

❑Tuer un processus en cours d'exécution grâce à la commande **kill**.
❑ Spécifier tout simplement sur la ligne de commande l'identifiant du processus à tuer.

Identification des processus

- Pour connaître l'identifiant PID, on utilise l'appel-système **getpid( )** déclaré dans <unistd.h>, qui ne prend pas d'argument et renvoie une valeur de type pid_t.

```
pid_t getpid(void);  /* processus courant */
```

- Le processus fils peut aisément accéder au PID de son père (noté PPID pour Parent PID) grâce à l'appel-système **getppid( )**, déclaré dans <unistd.h> :

```
pid_t getppid (void);   /*processus père */
```

Identification des processus (suite)

Exemple :

```
#include <stdio.h>
#include<stdlib.h>
#include <unistd.h>
int main(void)
{
printf("Processus %d , de père %d , taille d'un pid = %ld
octets \n", getpid(),getppid(),sizeof(pid_t));
return 0;
}
```

Création de processus

- L'appel-système **fork** permet de créer dynamiquement un nouveau processus (le processus fils) qui s'exécute en parallèle avec celui qui l'a créé (processus père).
- L'appel-système fork( ) est déclaré dans <unistd.h>, ainsi :

pid_t fork(void);

- Le processus père et le processus fils ont le même code source, mais la valeur retournée par fork permet de savoir si on est dans le processus père ou fils.
- La fonction **fork** retourne une valeur de type **pid_t**. Il s'agit généralement d'un int, il est déclaré dans **<sys/types.h>.**

Création de processus (suite)

La valeur renvoyée par **fork** est de :

- **-1** si il y a eu une erreur ;
- **0** si on est dans le processus fils ;
- Le PID du fils si on est dans le processus père. Cela permet ainsi au père de connaître le PID de son fils.

Le code de l'erreur est contenue dans la variable globale **errno**, déclarée dans le fichier **<errno.h>.** Ce code peut correspondre à deux constantes :

ENOMEM : le noyau n'a plus assez de mémoire disponible pour créer un nouveau processus;
EAGAIN : ce code d'erreur peut être dû à deux raisons : soit il n'y a pas suffisamment de ressources systèmes pour créer le processus, soit l'utilisateur a déjà trop de processus en cours d'exécution.

Création de processus (suite)

Exemple : Le programme suivant permet de créer un processus fils

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>       /* Pour fork() */
#include <sys/types.h>   /* Pour le type pid_t */
int main(void) {
pid_t pid_fils;
pid_fils = fork();
if (pid_fils == -1) {
printf("Erreur de création du nouveau processus \n");
exit (1); }
if (pid_fils == 0) {
printf("Nous sommes dans le fils\n");
printf("Le PID du fils est %d \n", getpid());
printf("Le PID de mon père (PPID) est %d \n", getppid());
} else {
printf("Nous sommes dans le père\n");
printf("Le PID du fils est %d\n", pid_fils);
printf("Le PID du père est %d\n", getpid());
printf("PID du grand-père : %d \n", getppid());}
return 0;}
```

Création de processus (suite)

Exercice 1 : Ecrire le programme de l'exemple précédent avec switch

Exercice 2 : Combien de processus sont lancés par le programme suivant

```
#include<stdio.h>
#include<unistd.h>
int main()
{
int p1, p2;
p1 = fork();
p2 = fork();
exit(0);
}
```

Création de processus (suite)

Exercice 3 :

- Quel est le processus de pid 1 ?
- Écrire un programme qui crée un fils. Le père doit afficher « Je suis le père » et le fils doit afficher « Je suis le fils ».
- utiliser la primitive **sleep** avec 20s pour voir les résultats suivants:

1) Le fils se termine avant le père,
2) Le père se termine avant le fils.

- Taper ps -la dans un autre terminal avant la fin du père, avant la fin du fils.
- Quels sont les ppid du père et du fils ?
- Donner une explication.

Création de processus (suite)

Solution de l'exercice 3: Utiliser les options l a x de ps pour pouvoir voir le processus init.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main() {
pid_t pid_fils;
pid_fils = fork();
switch(pid_fils ) {
case (-1):
printf("Erreur fork  \n ");
break;
case (0):
/*sleep(20)*/
printf("Le PID du fils est %d  et mon  (PPID) est %d\n", getpid(), getppid());
break;
default:
/*sleep(20)*/
printf("Le PID du père est %d et mon PPID est %d \n", getpid(), getppid());
} return 0;}
```

Terminaison d'un processus

Deux façons différentes pour la terminaison d'un programme:

- laisser le processus finir la fonction **main** avec l'instruction **return** suivie du code de retour du programme.
- terminer le programme grâce à la fonction :

#include <stdlib.h>
void exit(status);

```
#include <stdio.h>
#include <stdlib.h>
void quit(void) {
printf(" Nous sommes dans la fonction quit().\n");
exit(0); }
int main(void) {
quit();
printf(" Nous sommes dans le main.\n");
return 0;}
```

Le résultat est :
Nous sommes dans la fonction quit().

Synchronisation entre père et fils

L'appel système **wait** sert à attendre la mort d'un fils :
pid_t wait(int *status)
Cet appel a le comportement suivant :

- le processus courant n'a aucun fils, il retourne -1 avec errno ;
- le processus courant a au moins un fils zombie, un zombie est détruit et son pid est retourné par wait ;
- aucun des fils du processus courant n'est un zombie, wait bloque en attendant la mort d'un fils.

Extraction des informations sur la façon dont s'est terminé le processus fils se fait par :

WIFEXITED(status) retourne vrai si le fils est mort de façon normale.
WEXITSTATUS(status), retourne le paramètre de **exit** utilisé par le fils, cette macro n'est valide que lorsque WIFEXITED(status) est vrai.

Synchronisation entre père et fils (suite)

L'appel système **waitpid** est une version étendue de wait :
pid_t waitpid(pid_t pid, int *status, int flags) ;

Le paramètre pid indique le processus à attendre ; lorsqu'il vaut -1, waitpid attend la mort de n'importe quel fils (comme wait).

Le paramètre flags peut avoir les valeurs suivantes :
– 1 : dans ce cas waitpid attend la mort du fils (comme wait) ;
– WNOHANG : dans ce cas waitpid récupère le zombie si le processus est mort, mais retourne 0 immédiatement dans le cas contraire.

Synchronisation entre père et fils (suite)

Exemple. Voici un exemple où le père récupère le code renvoyé par le fils dans la fonction exit.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>  /* Pour wait() */
#include <errno.h> /* permet de récupérer les codes
d'erreur */
int main(void)
{
pid_t pid_fils ;
int status;
switch (pid_fils=fork())
{
case -1 : perror("Problème dans fork()\n");
exit(errno); /* retour du code d'erreur */
break;
```

Synchronisation entre père et fils (suite)

```
case 0 : puts("Je suis le fils");
puts("Je retourne le code 3");
exit(3);
default : puts("Je suis le père");
puts("Je récupère le code de retour");
wait(&status);
printf("Code de sortie du fils %d : %d\n", pid_fils,
WEXITSTATUS(status));
}
return 0;
}
```

La trace de ce programme est la suivante :

Je suis le fils
Je retourne le code 3
Je suis le père
Je récupère le code de retour
Code de sortie du fils 3444 : 3

Arguments en ligne de commande

La fonction main d'un programme peut prendre des arguments en ligne de commande.
Par exemple, si un fichier monprog.c a permis de générer un exécutable monprog à la compilation.
$ gcc monprog.c -o monprog

On peut invoquer le programme monprog avec des arguments
$ ./monprog argument1 argment2 argument3

Exemple : La commande cp du bash prend deux arguments :
$ cp nomfichier1 nomfichier2

Arguments en ligne de commande (suite)

- Pour récupérer les arguments dans le programme C, on utilise les paramètres argc et argv du main.
- L'entier argc donne le nombre d'arguments rentrés dans la ligne de commande **plus 1.**
- Le paramètre argv est un tableau de chaînes de caractères qui contient comme éléments :

• Le premier élément argv[0] est une chaîne qui contient le nom du fichier exécutable du programme ;

• Les éléments suivants argv[1], argv[2],etc... sont des chaînes de caractères qui contiennent les arguments passés en ligne de commande.

Arguments en ligne de commande (suite)

Exemple : Ecrivez un programme "monprog" qui prend des arguments et qui affiche : Argument 1 : ...
Argument 2 : ...

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
int i;
if (argc==1)
printf("Le programme n'a reçu aucun argument");
if (argc>=2){
printf("Le programme a reçu les arguments suivants :");
for (i=1;i<argc;i++)
printf("Argument %d = %s\n", i, argv[i]);}
return 0;}
```

Résultat :

```
$ ./monprog  Je mange
Argument 1 = Je
Argument 2 = mange
```

Variables d'environnement

- Les variables d'environnement sont de la forme:

  NOM=VALEUR

- Lorsqu'un programme en C démarre, ces variables sont automatiquement copiées dans un tableau de char.

- Pour accéder en déclarant la variable externe globale **environ** au début de votre fichier, comme ceci :

  extern char **environ;

- Ce tableau contient des chaînes de caractères se terminant par un pointeur **NULL** pour marquer la fin de la liste.

Variables d'environnement (suite)

Exemple:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
extern char **environ;
int main(void)
{
int i;
for (i=0;environ[i]!=NULL; i++)
printf(" %s \n",environ[i]);
return 0;
}
```

Variables d'environnement (suite)

Variables d'environnement classiques

HOME : contient le répertoire personnel de l'utilisateur ;
PATH : contient la liste des répertoires où le shell va chercher les commandes exécutables ;
PWD : contient le répertoire de travail ;
USER (ou **LOGNAME**) : nom de l'utilisateur ;
TERM : type de terminal utilisé ;
SHELL : shell de connexion utilisé.

Exemple:
On peut afficher La valeur d'une variable avec la commande :
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games

L’appel système exec

- Un programme peut se faire remplacer par un autre code source ou un script shell en faisant appel à **exec**.
- Il y a en fait plusieurs fonctions de la famille exec qui sont légèrement différentes.

• La fonction **execl** prend en paramètre une liste des arguments à passer au programme (liste terminée par NULL).

int execl(char*path, char*arg0, char*arg1,..., char*argn,NULL)

• La fonction **execlp** permet de rechercher les exécutables dans les répertoires apparaissant dans le PATH, ce qui évite souvent d’avoir à spécifier le chemin complet.

int execlp(char*file,char*arg0,char*arg1,...,char*argn,NULL)

• La fonction **execv** n’a pas besoin de connaître la liste des arguments à l’avance (ni même leur nombre).

int execv(char*path,char*argv[])

L’appel système exec (suite)

- Le premier paramètre est une chaîne de caractère qui doit contenir le chemin d’accès complet au fichier exécutable ou au script shell à exécuter.

- Les paramètres suivants sont des chaînes de caractère qui représentent les arguments passés en ligne de commande.

- La chaîne argv[0] doit donner le nom du programme (sans chemin d’accès).
- Les chaînes suivantes argv[1], argv[2], ... donnent les arguments.

L’appel système exec (suite)

Exemple 1 : exécution de la commande ls

```
/* execls.c exec de ls */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void){
execlp("ls", "ls", NULL);
/* execlp("lss", "ls", NULL); */
/* erreur : lss inconnu */
perror("Erreur sur execlp");
return 0;}
```

Exemple de résultats dans le répertoire /home/etudiant/tpc. Le message d’erreur n’est écrit par perror() qu’en cas d’erreur du execlp.

```
./execls
fichier1.c fichier2.c
```

L’appel système exec (suite)

Exemple 2 : Le programme suivant édite le fichier .c du répertoire de travail avec vi.
Dans le programme, le chemin d’accès à la commande vi est donné à partir de la racine /usr/bin/vi.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void){
char * argv[] = {"vi", "fichier.c", NULL};
/* dernier élément NULL, obligatoire */
execv("/usr/bin/vi", argv);
puts("Problème : cette partie du code ne doit jamais être exécutée");
return 0;
}
```

L’appel système exec (suite)

Exemple 3 : Ecrire un programme execv_ps.c qui lance la commande ps. Vous pouvez la trouver dans le dossier /bin.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void){
/* Tableau de char contenant les arguments (là aucun : le
nom du
programme et NULL sont obligatoires) */
char * arguments[] = {"ps", NULL};
/* dernier élément NULL, obligatoire */
execv("/bin/ps", arguments);
perror("Problème : execv");
return 0;}
```

Exemple de résultats : ./execv_ps

```
PID   TTY  TIME     CMD
1762  pts/0 00:00:00 bash
1845  pts/0 00:00:00 ps
```

L’appel système exec (suite)

Exemple 4 : Ecrire un programme prog_execl qui crée un processus fils qui exécute le fichier « affichez » avec le contenu «salut » . On utilisera la commande cat qui se trouve dans le dossier /bin.

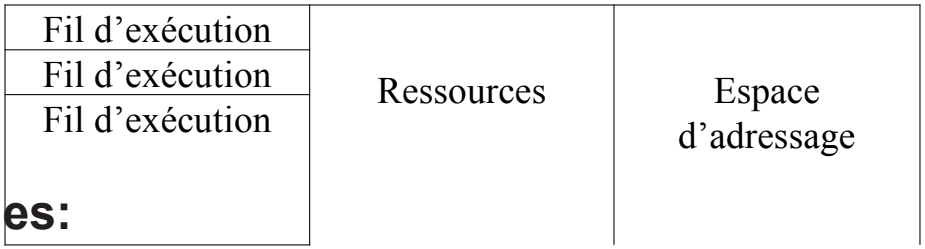
```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
int main() {
pid_t pid;
if((pid = fork()) < 0)
perror("fork error");
else if(pid == 0) {
if(execl("/bin/cat", "cat","affichez",NULL) < 0)
perror("execl error");}
return 0;}
```

Exemple de résultats : ./prog1
salut

Qu'est-ce qu'un thread?

- **Le processus léger** (thread) constitue une extension du modèle traditionnel de processus, appelé en opposition **processus lourd**.
- Un processus classique est constitue d'un espace d'adressage avec un seul fil d'exécution, ce fil d'exécution étant représenté par une valeur de compteur ordinal et une pile d'exécution.
- Un processus léger (thread) consiste à admettre plusieurs fils d'exécution indépendants dans un même espace d'adressage, chacun de ces fils d'exécution étant caractérisés par une valeur de compteur ordinal et une pile d'exécution privée.

processus classique (lourd)



processus léger (thread)

Avantages:

- **Réactivité**. Le processus léger s'exécute même si certaines de ses parties sont bloquées.
- **Partage de ressources**.
- **Économie d'espace mémoire et de temps**.

Primitives de gestion des threads

❑ Les primitives de la gestion des threads sont très semblables à celles de la gestion des processus.

- Inclure l'en-tête **<pthread.h>** dans les fichiers sources
- Ajouter l'option **–lpthread** lors de l'appel à l'éditeur des liens
- Chaque thread est identifié par un type **pthread_t**
- La primitive **pthread_self()** permet à chaque thread de connaître son propre identifiant.

❑ Pour créer un thread, il faut créer une fonction qui va s'exécuter dans le thread, qui a pour prototype :
void *ma_fonction_thread(void *arg);
Dans cette fonction, on met le code qui doit être exécuté dans le thread.

Création d'un thread

- Créer un thread par un appel à la fonction **pthread_create** qui permet de passer en argument la fonction ma_fonction_thread dans un pointeur de fonction.
- La fonction **pthread_create** a pour prototype :

int pthread_create(pthread_t *thread, pthread_attr_t *attributs,void *(*ma_fonction_thread)(void *arg), void *arg);

- La fonction renvoie une valeur de type int.
- renvoyer 0 si la création a été réussie ou une autre valeur si il y a eu une erreur.
- Le premier argument est un pointeur vers l'identifiant du thread (valeur de type pthread_t).
- Le deuxième argument attributs désigne les attributs du thread, et on peut mettre NULL pour avoir les attributs par défaut.
- Le troisième argument est un pointeur sur la fonction à exécuter dans le thread (par exemple ma_fonction_thread).
- Le quatrième argument est l'argument de la fonction de thread.

Suspension d'un thread

- La fonction **pthread_join** (similaire à la fonction wait dans le fork) permet aussi de récupérer la valeur retournée par la fonction ma_fonction_thread du thread.
- Le prototype de la fonction pthread_join est le suivant :

int pthread_join(pthread_t thread, void **retour);

- Le premier paramètre est l'identifiant du thread
- Le second paramètre est un passage par adresse d'un pointeur qui permet de récupérer la valeur retournée par ma_fonction_thread.

Terminaison d'un thread

- La fonction **pthread_exit** termine le thread en cours d'exécution.

int pthread_exit(void *valeur_de_retour);

- Une valeur de retour peut être communiqué par un autre thread par l'intermédiaire de la fonction **pthread_join()**.
- NULL en paramètre indique une absence de valeur de retour.
- Si le thread est détruit avant sa fin normale par un autre thread, la valeur de retour est **PTHREAD_CANCEL**.

Exemple 1: Le programme threadpid0.c montre l'implantation de threads dans GNU/Linux, ainsi que la récupération du pid du thread.

```
#include <unistd.h> // pour sleep
#include <pthread.h> // pthread_create , pthread_join ,
pthread_exit
#include <stdio.h>
void *fonction(void *arg)
{
printf ( "pid du thread fils = %d\n", (int)getpid ());
while(1);
return NULL;}
int main(){
pthread_t thread1;
printf ("pid de main = %d\n", (int)getpid ());
pthread_create(&thread1, NULL, &fonction, NULL);
while(1);
return 0;}
```

smi@smi-desktop:~$ gcc -o pthreadpid0 threadpid0.c -lpthread
smi@smi-desktop:~$./pthreadpid0
pid de main = 24136
pid du thread fils = 24136

Exemple 2: Le programme suivant illustre le partage de l'espace d'adressage entre les threads d'un même processus.

Le thread fils qui exécute la fonction addition() se partage une même variable i avec le thread principal, qu'ils incrémentent chacun de leur côté.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
int i ;
void addition(){
i=i+10 ;
printf("hello, thread fils %d\n",i) ;
i=i+20 ;
printf("hello, thread fils %d\n",i) ;}
main() {
pthread_t num_thread;
i=0 ;
if(pthread_create(&num_thread,NULL,(void *(*)())addition,NULL)==-
1)
perror("pb pthread_create \n") ;
i=i+1000 ;
printf("hello, thread principal %d\n",i) ;
i=i+2000 ;
printf("hello, thread principal %d\n",i) ;
pthread_join(num_thread,NULL) ;}
```

Exemple 3: (lourd)

Le processus père crée un processus fils et chacun des processus incrémente également une variable i des mêmes valeurs que précédemment.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>  /* Pour wait() */
main(){
int pid;
int i=0;
pid=fork();
if(pid==0){
i=i+10;
printf("hello, fils %d\n",i);
i=i+20;
printf("hello, fils %d\n",i);
}
else{
i=i+1000;
printf("hello, père %d\n",i);
i=i+2000;
printf("hello, père %d\n",i);
wait();}}
```

Ordonnancement

La fonction d'ordonnancement gère le partage du processeur entre les différents processus en attente pour s'exécuter, c'est-à-dire entre les différents processus qui sont dans l'état prêt.
L'opération d'élection consiste à allouer le processeur à un processus.
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 l'utilisateurs (systèmes interactifs).
Temps d'exécution : chaque programme doit s'exécuter le plus vite possible.
Equité : chaque processus reçoit sa part du temps processeur.

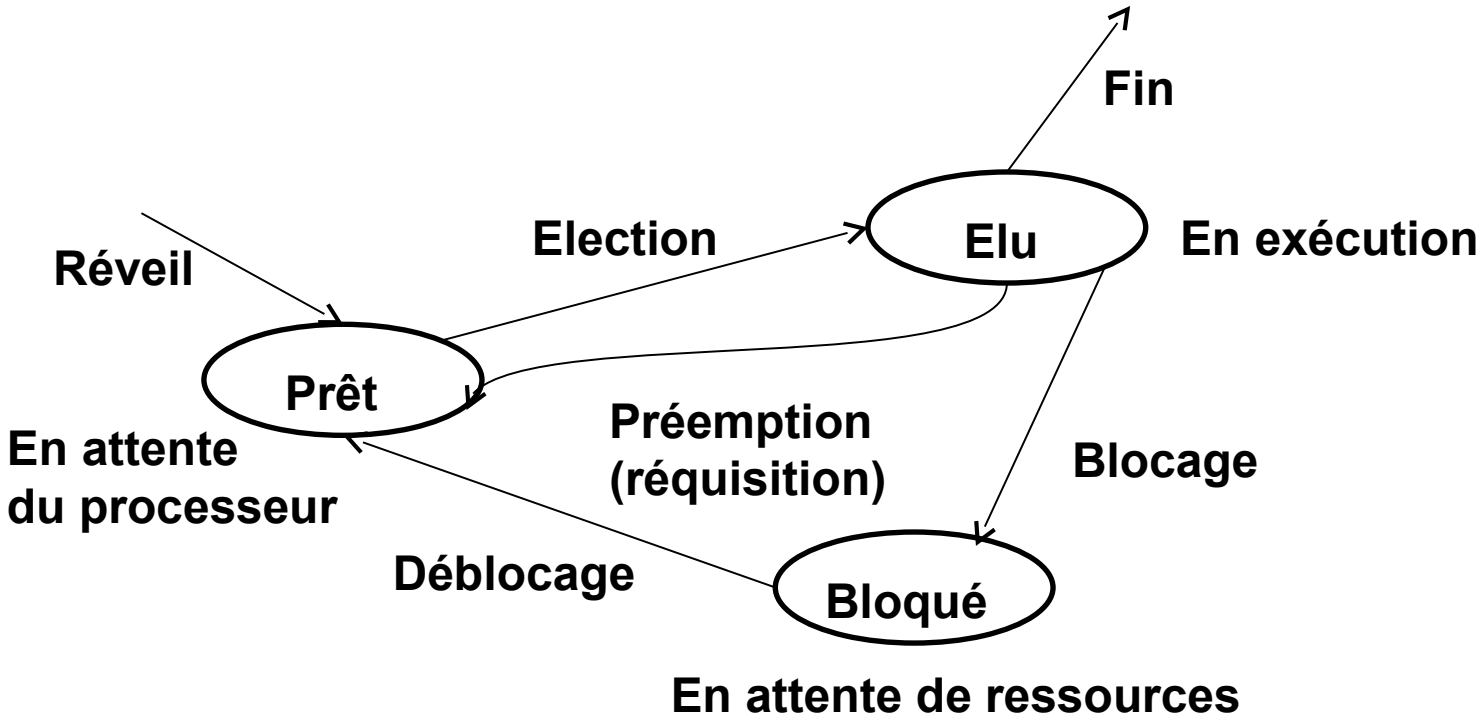
Ordonnancement (suite)

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.

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

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

Ordonnancement (suite)



Il existe deux familles d'algorithmes :

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

Préemptif: à intervalle régulier, l'ordonnanceur reprend la main et élit un nouveau processus actif.

Algorithmes d'ordonnancement

Premier arrivé, premier servi (FIFO: First in first out)

- Algorithme non préemptif
- File d'attente FIFO pour les processus prêts
- Cet algorithme est facile à implanter, mais il est loin d'optimiser le temps de traitement moyen.

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

- Algorithme non préemptif
- Le prochain cycle le plus court est sélectionné
- En cas d'égalité, on revient au FIFO
- Temps de réponse plus rapide que le précédent

Plus petit temps de séjour (SRT : Shortest Remaining Time)

- 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.

Algorithmes d'ordonnancement (suite)

Tourniquet (Round Roubin)

- Il s'agit d'un algorithme ancien, simple et fiable.
- Le processeur gère une liste circulaire de processus.
- Algorithme non 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.
- Conçu spécialement pour le temps partagé.
- La performance dépend de la taille du quantum.
- Un quantum trop grand augmente les temps de réponse.
- Un quantum trop petit multiplie les commutations de contexte jusqu'à les rendre non négligeables.

Algorithmes d'ordonnancement (suite)

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.

Files d'attente à plusieurs niveaux

- Découpage de la file d'attente des processus prêts en plusieurs files.
- Ordonnancement spécifique au sein de chaque file(RR, FIFO)
- Ordonnancement des files entre elles (priorités fixes, allocation de tranches de temps etc.)

Ordonnancement dans les systèmes Linux

Le système linux ordonnance directement les threads.
sched_setscheduler() définit à la fois la politique d'ordonnancement et ses paramètres pour le thread dont l'identifiant est indiqué par pid. Il définit plusieurs classes de threads :

- 2 classes dites temps réel
  - Les FIFO (SCHED_FIFO)
  - Les Round Robin (SCHED_RR)

- Les threads en temps partagé :
  - Les threads standards (SCHED_OTHER)
  - Les threads de calculs (SCHED_BATCH)
  - Les threads de tâches de fond (SCHED_IDLE)

- Sont déclarée dans le fichier <**sched.h**>.
- Les threads en temps réel sont toujours plus prioritaires que des threads en temps partagé.

4 SYSTÈME DE GESTION DE FICHIERS

- Introduction
- Fichier
- Modes d'accès du SGF
- Gestion de l'espace disque
- Gestion de l'espace libre
- Le répertoire
- Fichier Linux
- Structure d'un répertoire Linux
- Partition Linux
- Les opérations sur les fichiers
- Les opérations sur les répertoires
- Les opérations sur les liens symboliques
- Les opérations sur les partitions

Introduction

- La mémoire centrale est une mémoire volatile.
- En effet la mémoire s'efface lorsqu'on coupe l'alimentation électrique de la machine.
- 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).
- Un SGF a pour principal rôle de gérer les fichiers et d'offrir les primitives pour manipuler ces fichiers.

Fichier

- Une unité de stockage logique.
- Ensemble d'informations en relation entre elles, qui est enregistré sur la mémoire auxiliaire.
- Le SE établit une correspondance entre les fichiers et les dispositifs physiques.
- Deux visions d'un système de fichiers:

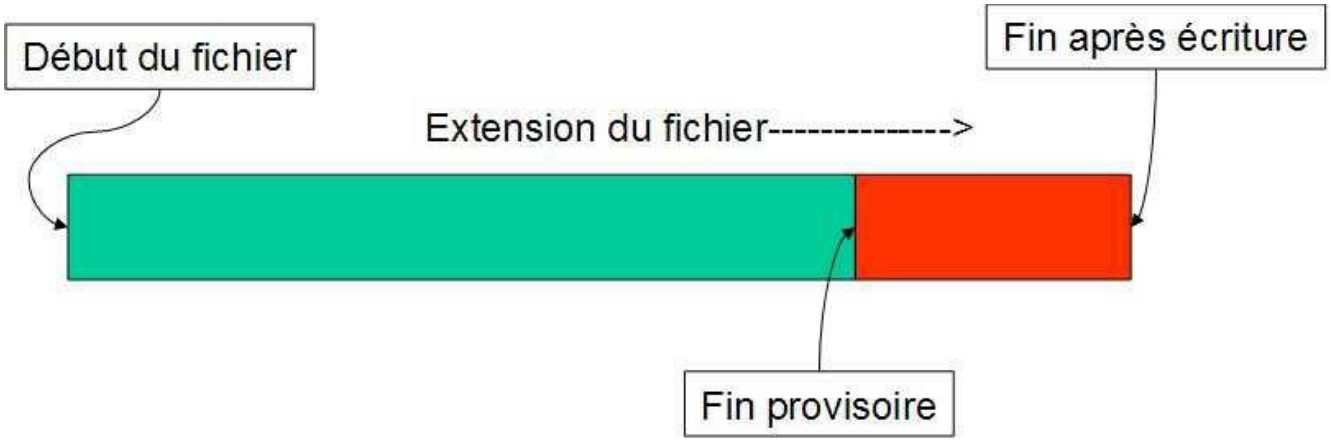
• **Point de vue de l'utilisateur:** nommage des fichiers, protection et droit d'accès, opération autorisées, etc.

• **Point de vue de l'implantation:** organisation physique d'un fichier sur un disque, gestion des blocs et manipulation des blocs physiques attribués à un fichier, gestion de l'espace libre du disque.

Modes d'accès du SGF

Accès séquentiel:

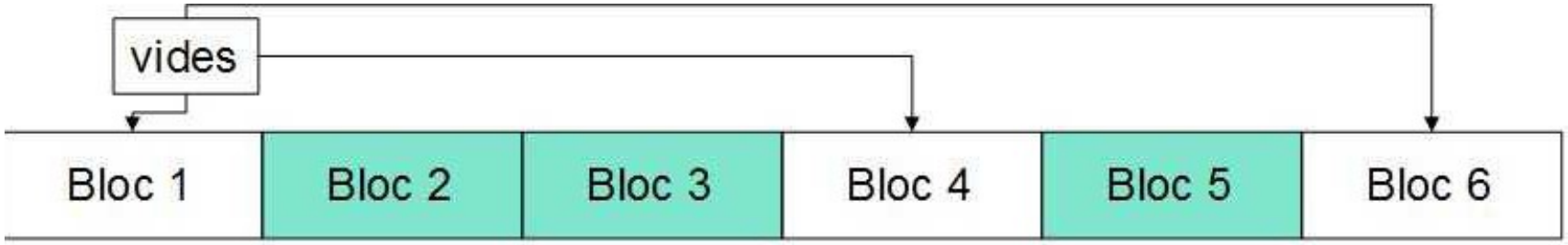
- L'information dans le fichier est traitée en ordre, un enregistrement après l'autre.
- Éditeurs et compilateurs utilisent cette méthode. Pratique quand le support de stockage était une bande magnétique.



Modes d'accès du SGF (suite)

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)

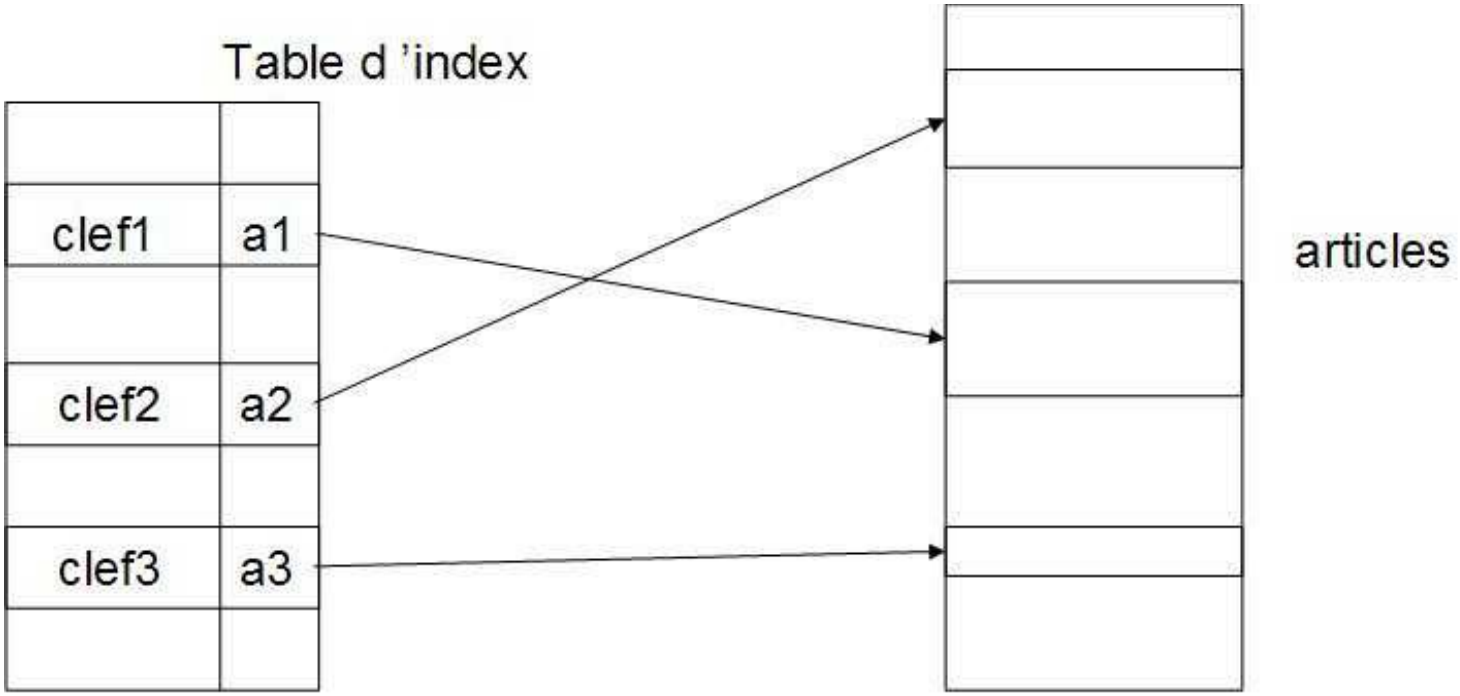


- On peut accéder directement au bloc 5 sans que les précédents ont été remplis au préalable

Modes d'accès du SGF (suite)

Accès indexé :

- Nécessite d'avoir un ensemble de clés ordonnées
- Relation entre clés et articles établie au moyen d'une table d'index.



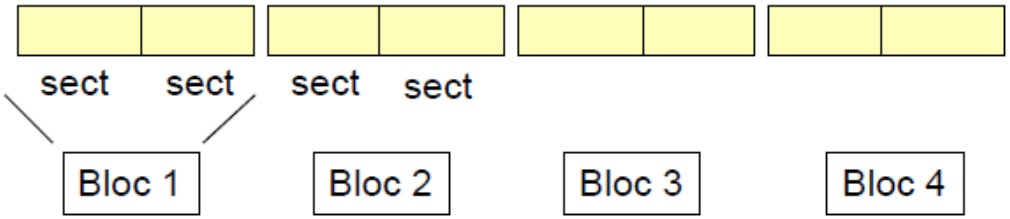
Gestion de l'espace disque

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

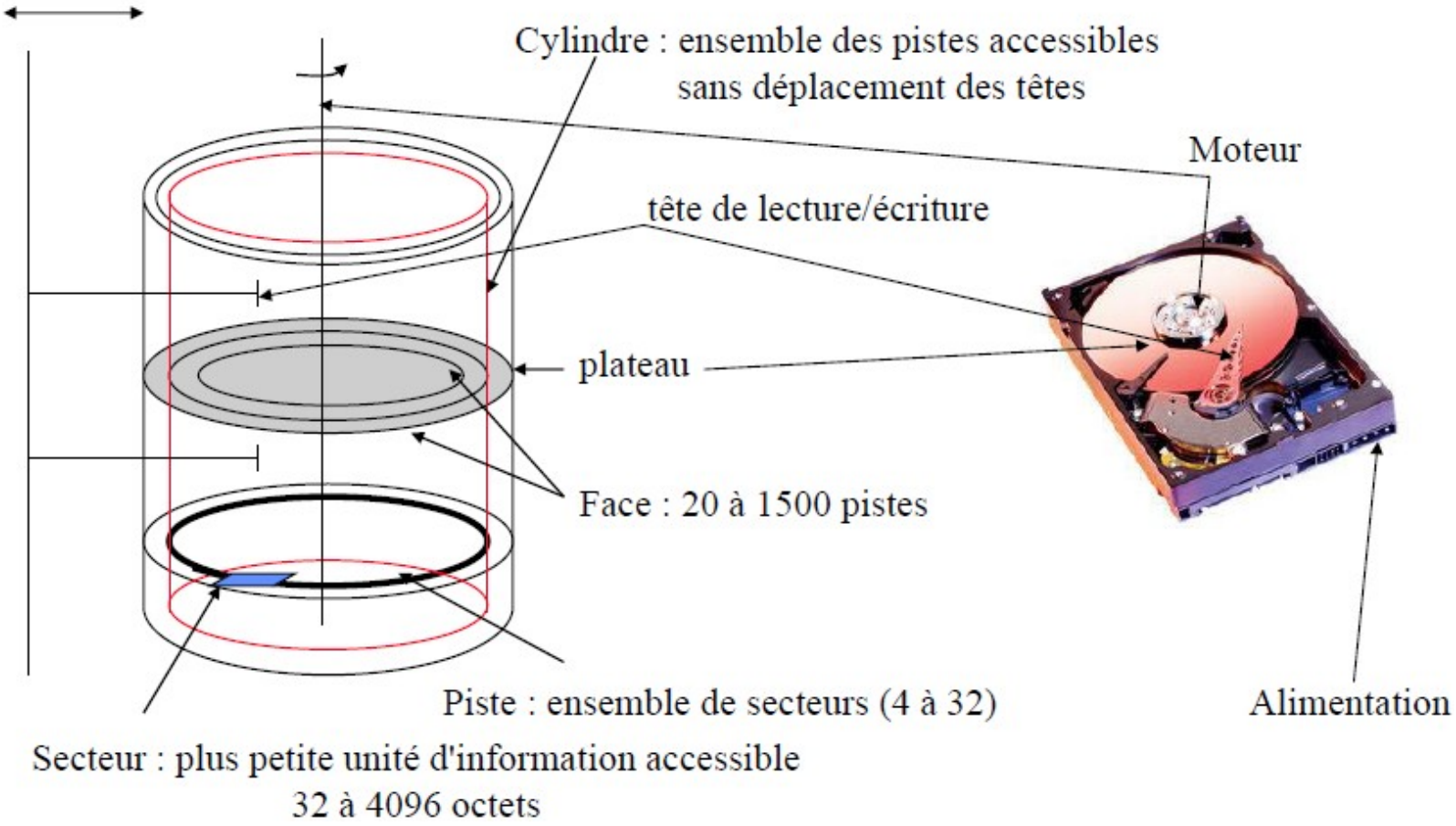
- Un fichier physique est constitué d'un ensemble de blocs physique. Il existe plusieurs méthodes d'allocation des blocs physiques :

1. allocation contiguë (séquentielle simple)
2. allocation par blocs chainés
3. allocation indexée

Les opérations de lecture et d'écriture du SGF se font bloc par bloc
1 bloc = 2 secteurs de 512 octets soit 1KO



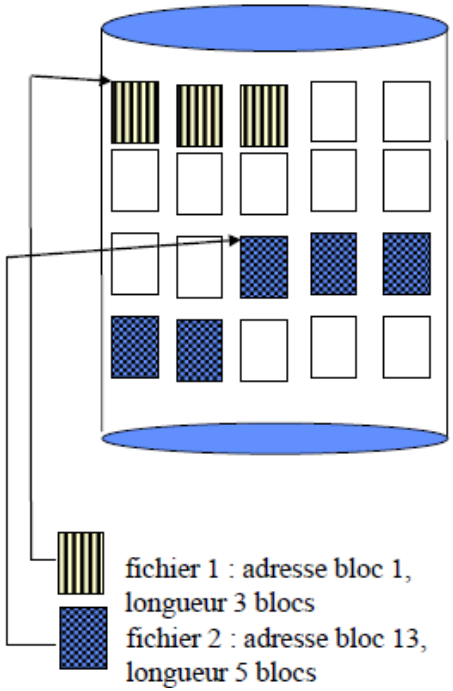
Structure du disque dur

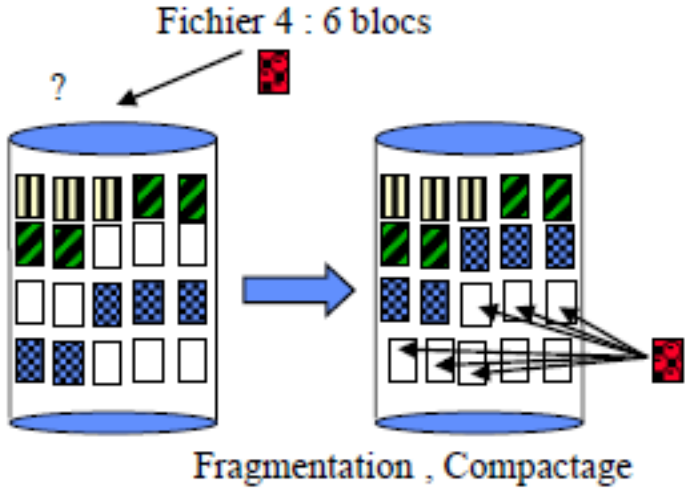


Allocation contiguë

- Un fichier occupe un ensemble de blocs contigus sur le disque
- Difficultés:

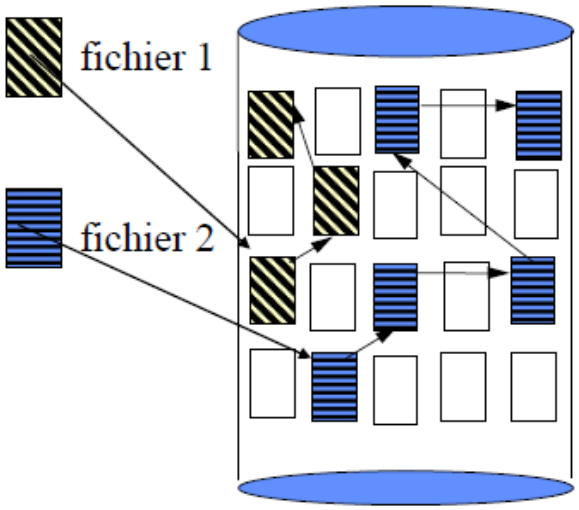
Fragmentation: trouver des espaces suffisants à une nouvelle allocation.
Compactage: regrouper les espaces libres dispersés en un seul espace libre exploitable.





Allocation par bloc chainée

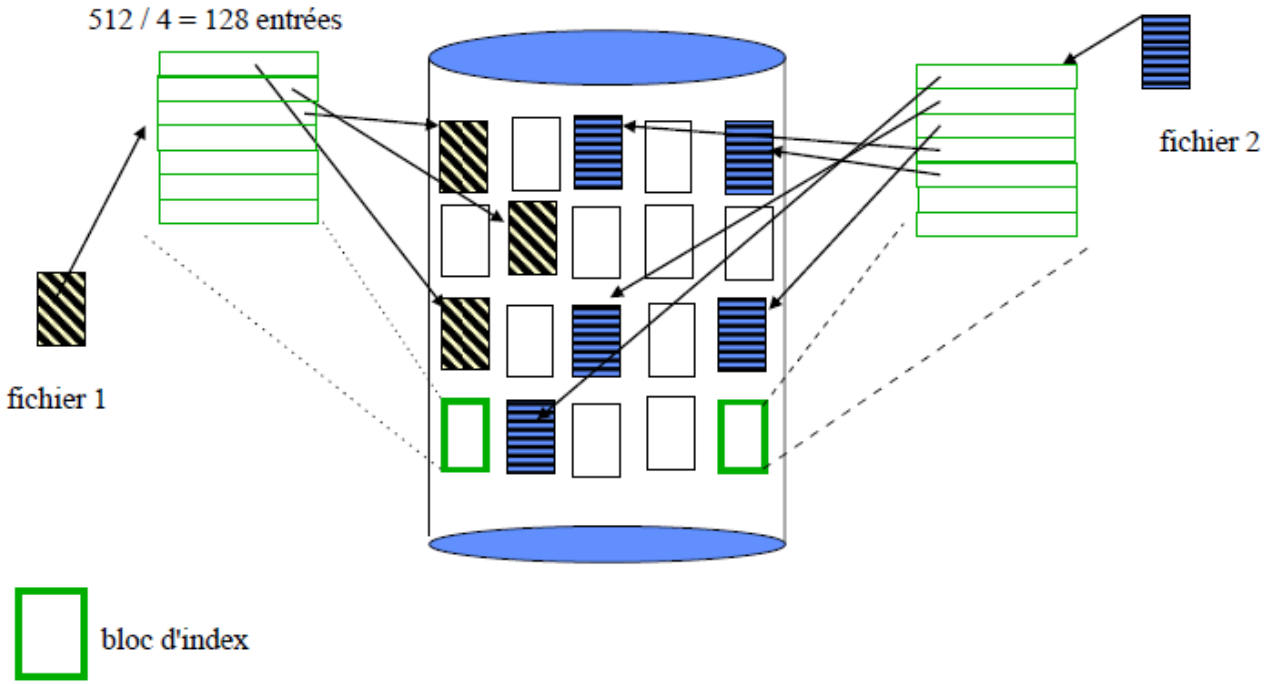
❑ Un fichier est constitué comme une liste chainée de blocs physiques, qui peuvent être dispersés n'importe où.



❑ Extension simple du fichier : allouer un nouveau bloc et le chainer au dernier
❑ Pas de fragmentation
❑ Difficultés :
✓ Le seul mode d'accès utilisable est le mode d'accès séquentiel.
✓ La perte d'un chaînage entraîne la perte de tout le reste du fichier.

Allocation indexée

Les adresses des blocs physiques constituant un fichier sont rangées dans une table appelée **index**, elle-même contenue dans un ou plusieurs blocs disque.



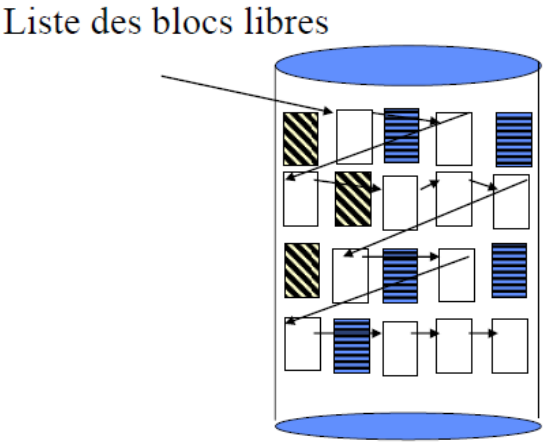
- Supporte bien l'accès direct
- «Perte de place » dans le bloc d'index

Gestion de l'espace libre

Les systèmes d'exploitation utilisent deux approches pour mémoriser l'espace libre : une statique et une dynamique.

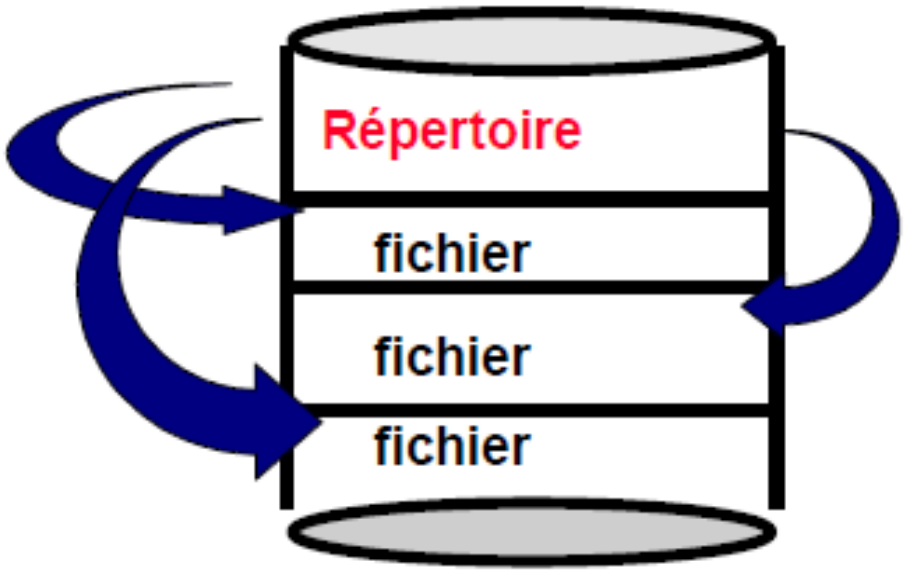
- Approche statique utilise une table de bits. A chaque bloc du disque, correspond un bit dans la table, positionné à 1 si le bloc est occupé, à 0 si le bloc est libre.
- Liste chainée: Approche dynamique utilise une liste chaînée constituée d'éléments, chacun mémorisant des numéros de blocs libres.

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



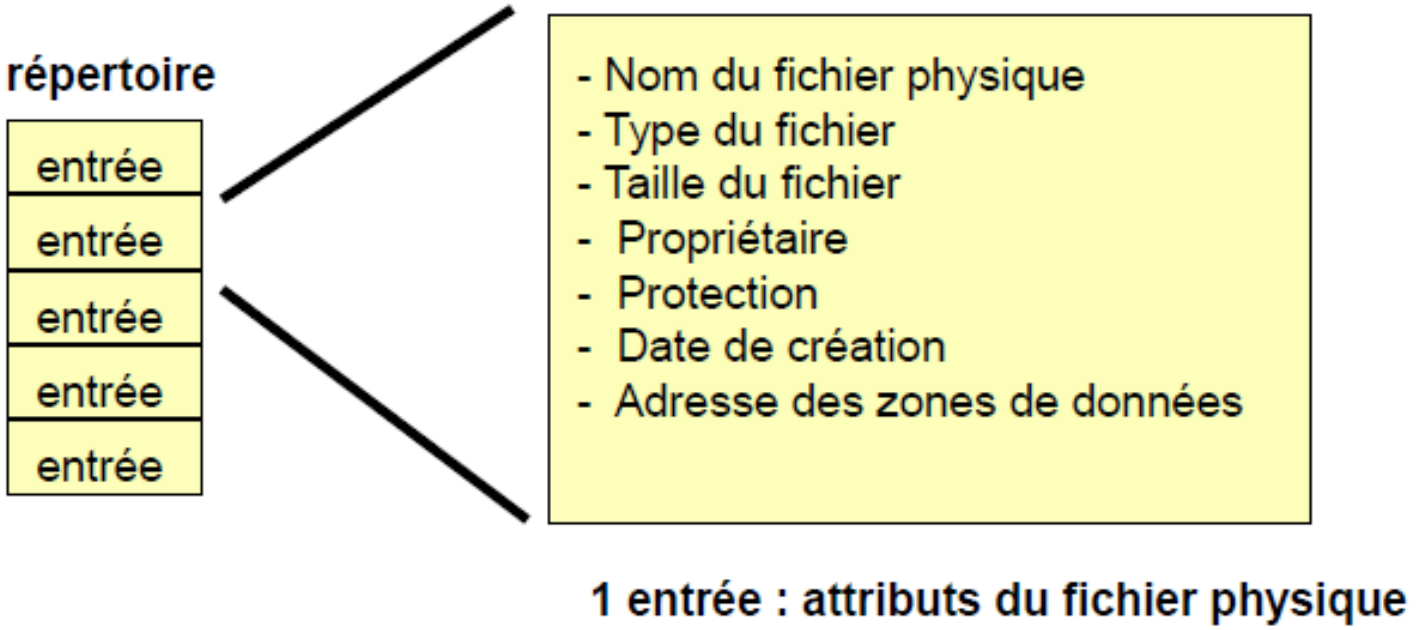
Le répertoire

- Le répertoire est une table sur le support permettant de référencer tous les fichiers existants du SGF avec leur nom et leurs caractéristiques principales.
- Le répertoire stocke pour chaque fichier l'adresse des zones de données allouées au fichier.



Le répertoire (suite)

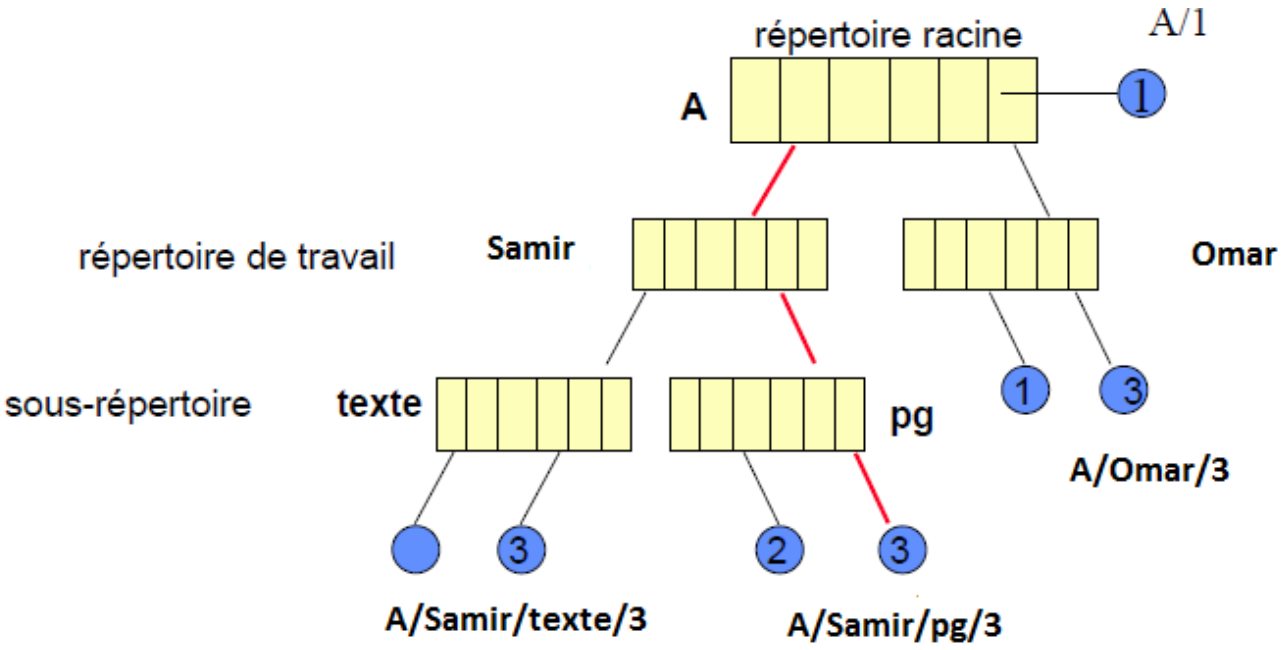
- Un répertoire est une zone disque réservée par le SGF.
- Le répertoire comprend un certain nombre d'entrées.
- Une entrée de répertoire concernant un fichier donné, contient les informations suivantes :



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.



Fichier Linux

- Identifier par un nom.
- La méthode d'allocation mise en œuvre est de type allocation indexée.
- Un fichier Linux est composé d'un descripteur appelé «**inode**» et de blocs physiques, qui sont soit des blocs d'index, soit 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).

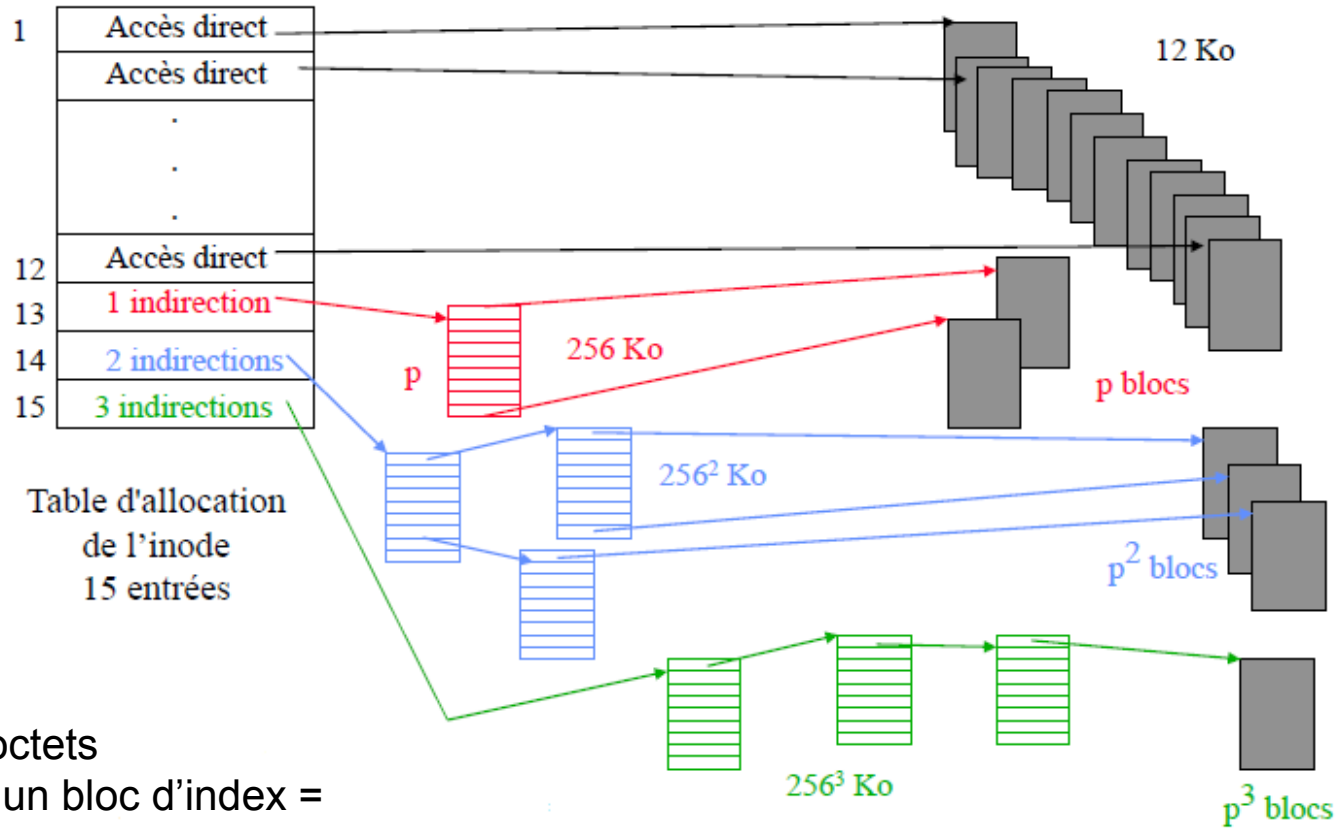
Fichier Linux: Inode

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

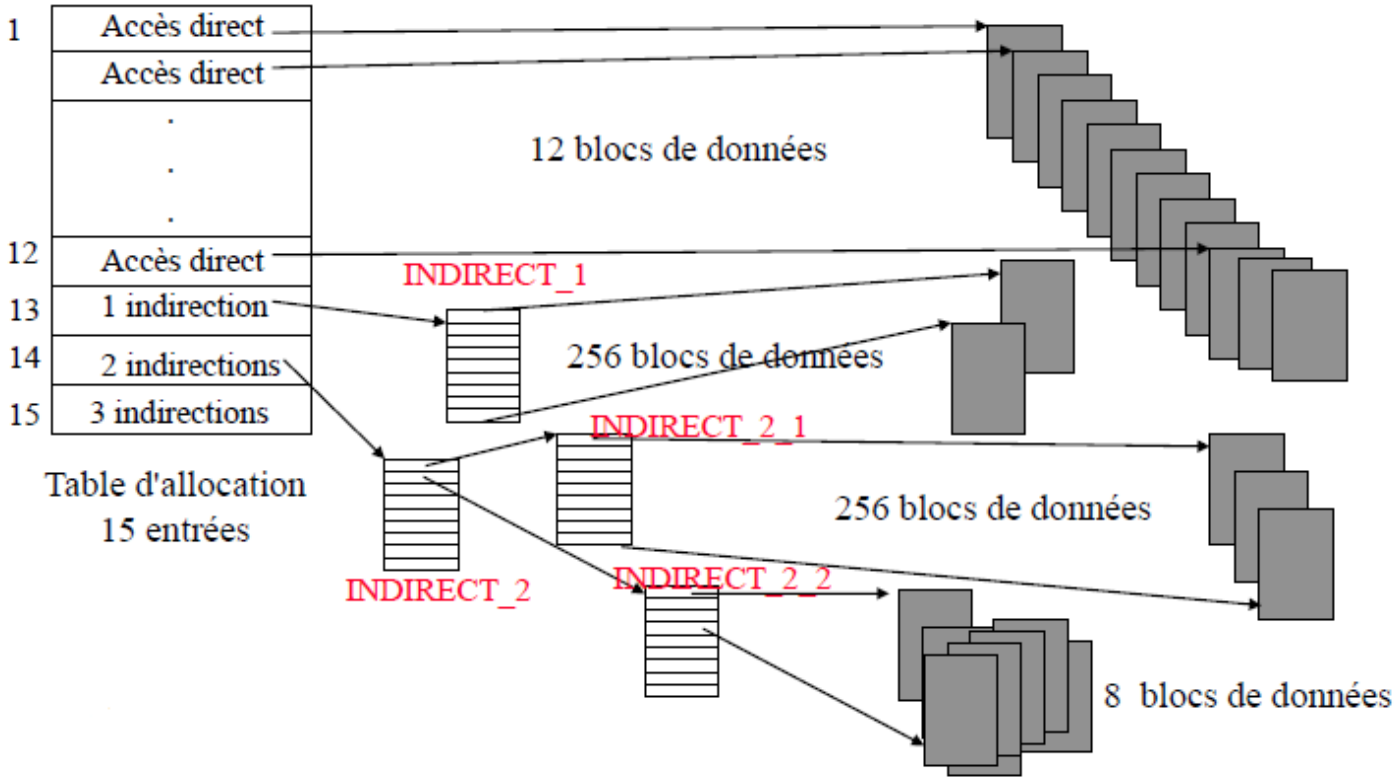
Fichier Linux: structure

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 indexée.



Bloc= 1024 octets
Numéro de blocs= 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 →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

Types de fichiers Linux

- Les **fichiers répertoires** contiennent des informations sur les fichiers et les sous-répertoires. **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

Types de fichiers Linux (suite)

- 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. Aucune commande n'est disponible pour créer de tels sockets : ils sont créés par chaque serveur utilisant ces sockets.
- Les périphériques sont des **fichiers spéciaux** qui appartiennent au sous-répertoire **/dev**.
  - ✓ 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.

Types de fichiers Linux (suite)

Le **b** précédant les droits d'accès de

- /dev/fd0 (floppy disk : lecteur de disquette)
- /dev/hdc1 indique que le périphérique échange des blocs de données.

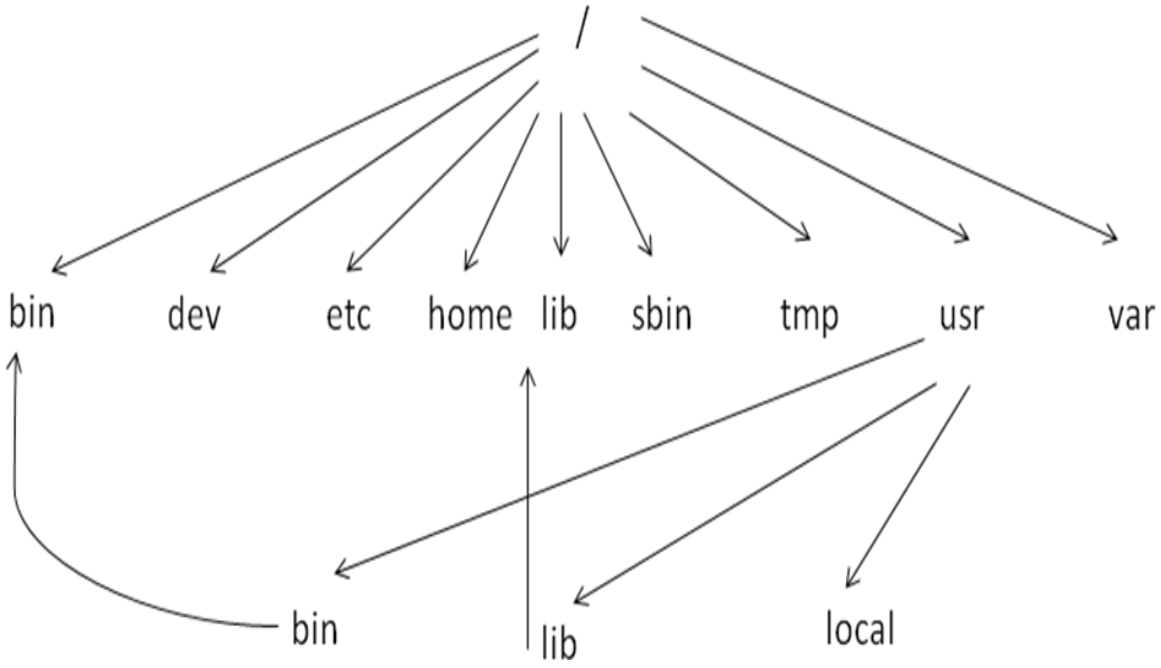
Le **c** devant les droits d'accès de

- /dev/tty (le terminal de l'utilisateur) indique que les échanges se font caractère.
- La mémoire physique peut être accédée 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
```

Structure d'un répertoire Linux

- 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.



Structure d'un répertoire Linux (suite)

/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

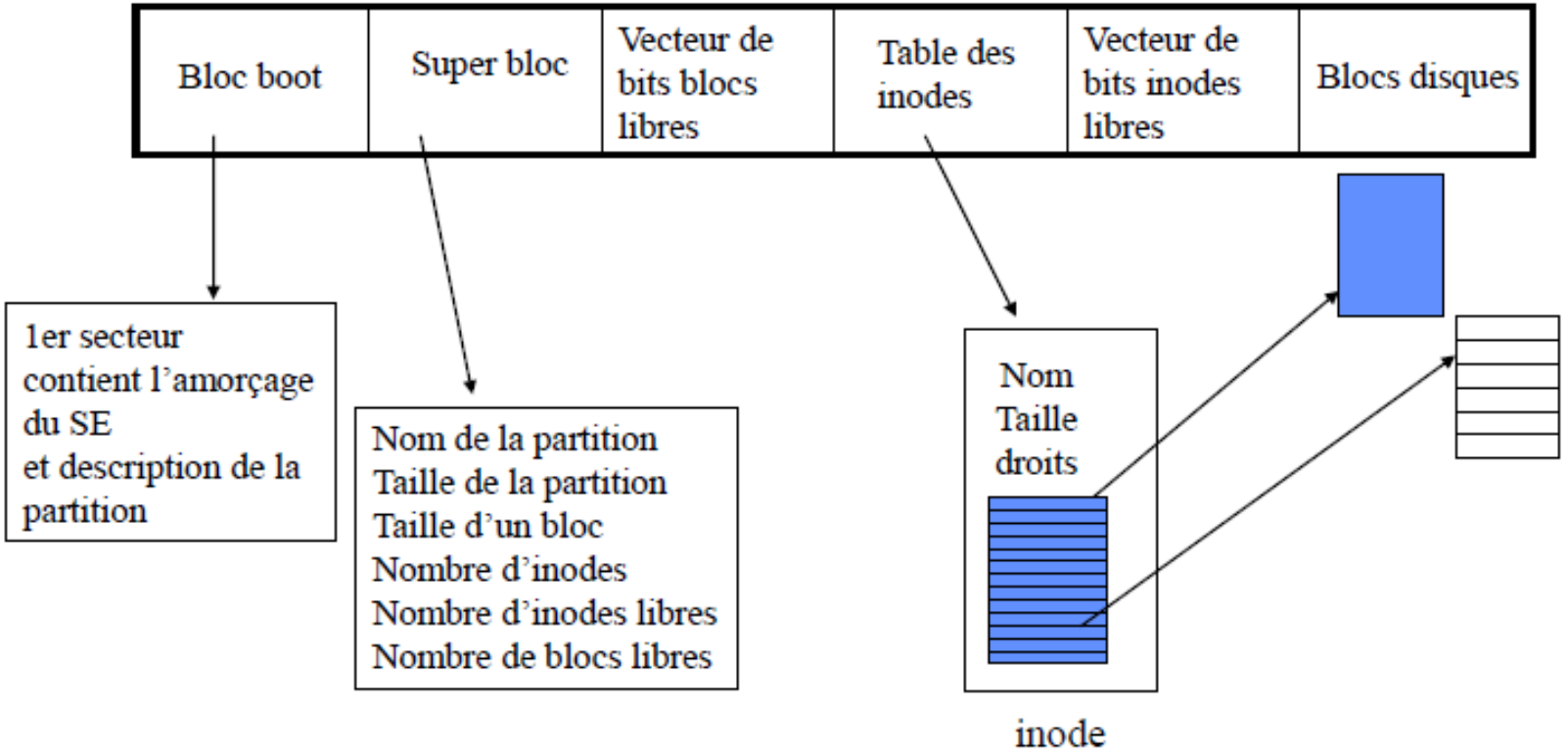
Structure d'un répertoire Linux (suite)

/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

Partition Linux : structure

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.

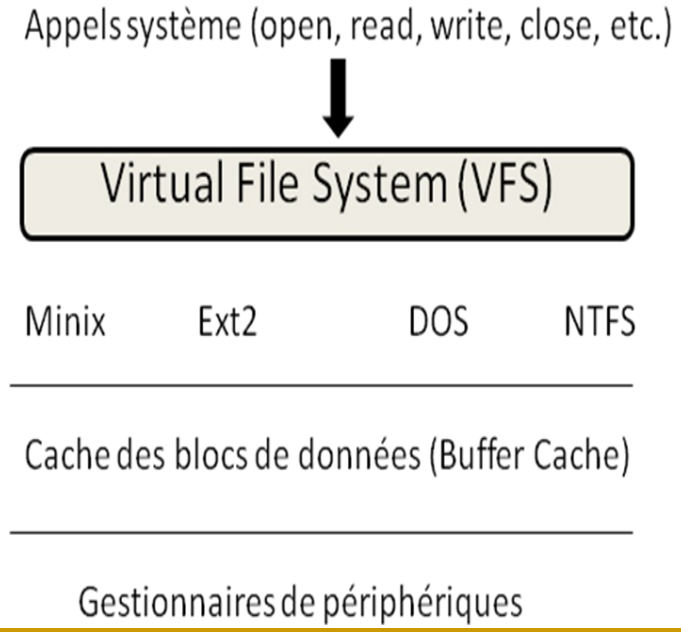


Virtual File system : VFS

Une couche logicielle appelée **VFS** est insérée dans le noyau Linux pour que l'accès aux SGF (Minix, Ext2, Dos, NTFS…) soit transparent et uniforme pour l'utilisateur.

Le VFS gère deux systèmes de cache :

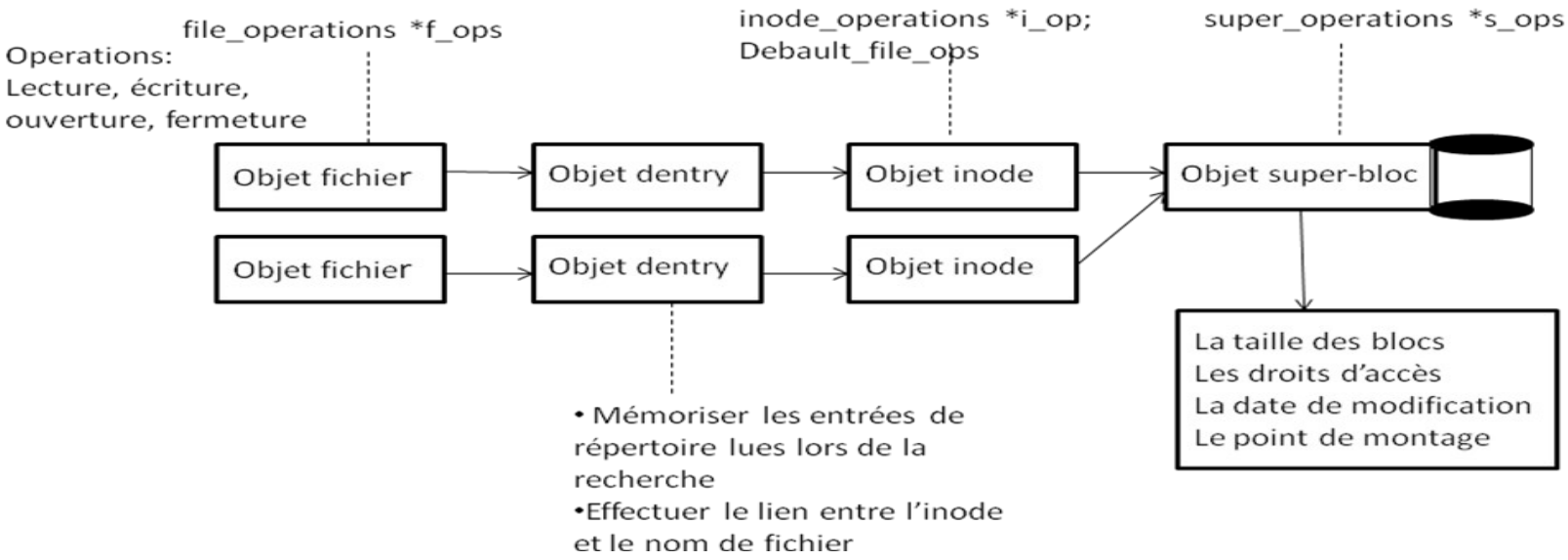
- Un cache de noms qui conserve les conversions les plus récentes des noms de fichiers en numéro de périphérique et numéro d'inode.
- Un cache de tampons disque appelé buffer cache, qui contient un ensemble de blocs de données lus depuis le disque.



Structure du VFS

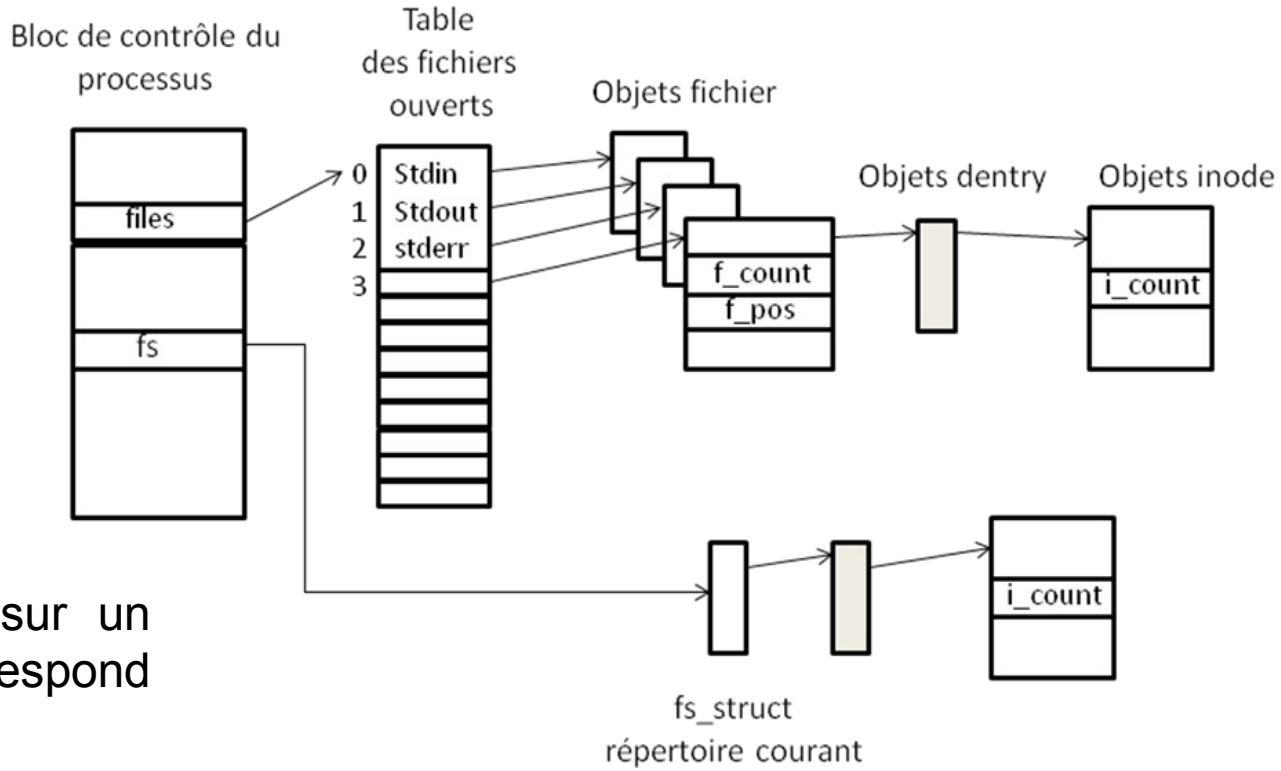
- La structure du VFS est organisée autour d'objets auxquels sont associés des traitements.
- Les objets sont : L'objet système de gestion de fichiers (super-bloc), l'objet inode et l'objet fichier (file) qui sont définit dans <linux/fs.h>.

Et l'objet nom de fichier (dentry).



Liaison du processus avec les objets du VFS

Les fichiers ouverts sont mémorisés dans une table des **fichiers ouverts,** pointée depuis le bloc de contrôle du processus.
Chacune des entrées de cette table (table fd), nommée **descripteur de fichier**, pointe vers un objet fichier. Chaque objet fichier pointe à son tour vers un objet dentry. Chaque objet dentry pointe vers l'objet inode du fichier.



Le champ fs pointe sur un objet fichier qui correspond au répertoire courant.

Ouverture d'un fichier par open(nom_fichier, mode_ouverture)

- La fonction **open()** bascule le processus appelant en mode superviseur, puis appelle la routine système **sys_open()**.
- Alloue une entrée dans la table **fd** et création d'un nouvel objet fichier.
- 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.
- 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

Lecture d'un fichier : le 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 un 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 copie le bloc disque dans le tampon libre.
  - ✓ Si tous les tampons sont occupés, il libère un tampon en choisissant le moins récemment accédé.

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);
```

*ref : spécifié le chemin du fichier à ouvrir dans l'arborescence du SGF.

mode_ouv : est une combinaison de l'opération de OU binaire « | » et des constantes permettant de spécifier les options sur le mode d'ouverture O_RDONLY O_WRONLY O_RDWR O_CREAT

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

Ouverture d'un fichier (suite)

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),

Fermeture, lecture et écriture dans un fichier

Fermeture d'un fichier:

L'appel système **close()** permet de fermer un descripteur de fichier desc.

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

Lecture d'un fichier:

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

L'appel système **read()** lit nb_octets octets au maximum et les copie dans le tampon ptr_buf.

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.

Il retourne le nombre d'octets réellement écrits. Sinon, il renvoie -1.

Exemple:

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 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)
write(fd,buf,nbcar);
return 0;}
```

Se positionner dans un fichier

Il est 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**.

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.

Exemple 1:

```
#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 2:

```
#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/fatima/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;
}
```

Accès aux attributs des fichiers

L'appel système **stat** permet d'obtenir certaines des informations de l'inode décrite dans le fichier d'en-tête **<sys/stat.h>**. Pour **fstat**, il faut utiliser le descripteur obtenu avec **open**.

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

Accès aux attributs des fichiers (suite)

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:

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_IWUSR ? '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:

Le programme « unlink_cs.c » permet d'effacer (et encore réutiliser) 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;}
```

Modification dans un fichier

Modification du nom d'un 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 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) ;

Opérations sur les répertoires

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.

Opérations sur les répertoires (suite)

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.

Opérations sur les répertoires (suite)

Exploration d'un répertoire:

opendir(), **readdir()** et **closedir()** permettent respectivement d'ouvrir un répertoire, de lire une entrée du répertoire et de fermer le répertoire.

```
#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.

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;}
```

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) ;
```

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:

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;
}}
```

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 SGF 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.

Opérations sur les partitions (suite)

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.

5 ENTRÉES/SORTIES

- Introduction
- Fichiers en C
- Les flux standards
- Fonctions d'Entrées/Sorties
- Interprétations
- sprintf et sscanf
- Fin de fichier en lecture
- Paramètres dans la fonction main
- Les fonctions atoi(), atof() et atol()

Introduction

- **Entrées/Sorties**: Opérations d'échanges d'informations dans un système informatique.

- les entrées/sorties sont des fonctions et des appels système du système d'exploitation sur lequel s'exécute le programme.

- La plupart de ces fonctionnalités se situent dans la librairie <stdio.h> (stdio : standard input output).

- Sans entrées/sorties, il est impossible de produire des programmes évolués ayant de grands impacts sur l'environnement, le système et la mémoire.

Introduction (suite)

❑Toutes interactions avec l'environnement et la mémoire (et leurs modifications) nécessite de récupérer ou de constituer un flux d'informations.

❑ Flux classiques:

- Texte tapé au clavier (entrée)
- Mouvement de la souris et clic (entrée)
- Fichiers sur le disque dur (entrée/sortie)
- Position de la souris sur l'écran (entrée/sortie)
- Texte affiché sur l'écran (sortie)
- Réseau (entrée/sortie)
- Webcam (entrée)
- …

Fichier en C

- Espace de stockage de données
- Il est possible de manipuler (créer/lire/écrire/détruire) des fichiers du disque dur via des fonctions d'Entrées/Sorties de la librairie <stdio.h>.

- **Primitives systèmes:**

int open(const char* pathname,int flags,mode_t mode);
int close(int fd);
ssize_t read(int fd,void* buf,size_t count);
ssize_t write(int fd,const void* buf,size_t count);

- Les déclarations des appels système se trouvent dans les fichiers suivants :

/usr/include/unistd.h
/usr/include/sys/types.h
/usr/include/sys/stat.h
/usr/include/fcntl.h

Fichier en C (suite)

- Les fichiers sont manipulables via un pointeur sur une structure FILE.
- FILE = structure décrivant un fichier

Manipulation de fichier

- Ouverture du fichier, constitution de l'objet (FILE *) et placement du pointeur.
- Fonctions d'Entrées/Sorties du langage C pour modifier des caractères sur le pointeur et déplacer ce pointeur dans le fichier ouvert.
- Fermeture du fichier.

Le dernier caractère d'un fichier ouvert est EOF pour (End Of File).

```
FILE* fopen(const char* path,const char* mode);
int fclose(FILE* stream);
size_t fread(void* ptr,size_t size,size_t n,FILE* stream);
size_t fwrite(const void* ptr,size_t size,size_t n,FILE* stream);
```

Les flux standards

- L'en-tête <stdio.h> fournit trois flux que l'on peut utiliser directement:

• stdin: (standard input) est l'entrée standard. Tout caractère tapé au clavier durant l'exécution d'un programme se retrouve dans ce flux.

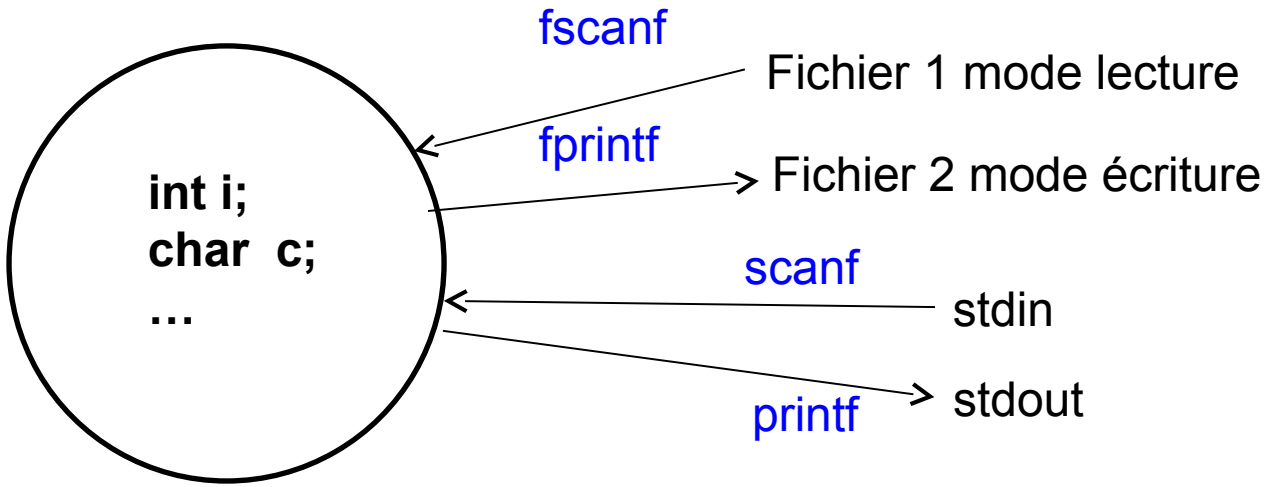
• stdout: (standard output) est la sortie standard. Tout caractère écrit dans ce flux sera affiché dans la console lors de l'exécution d'un programme.

• stderr: (standard error) est la sortie d'erreur standard.

Les fonctions d'Entrées/Sorties du langage C opérant sur les fichiers et flux standards sont très liées (printf - fprintf, scanf - fscanf, ...).

Fonctions d'Entrées/Sorties

- Les fonctions d'entrées/sorties servent à écrire et récupérer des données dans les fichiers ouverts et les flux standards.



printf : écriture formatée sur stdout

int printf(const char *format, ...);

- La liste de variables correspond aux différents types apparaissant dans le format.
- Valeur de retour = nombre de caractères écrits sur la sortie standard
- Retour négatif en cas d'erreur

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc,char* argv[]) {
int n=printf("commande=%s\n",argv[0]);
int i;
for (i=1;i<n;i++) {
printf("-");}
printf("\n");
return 0;}
```

```
SMI4$./test_printf
commande=./test_printf
- - - - - - - - - - - - - - - - - -
```

scanf: lecture formatée sur stdin

int scanf(const char *format, ...);

- On remplace les ... par une liste d'adresses de conteneurs mémoires adaptés
- **Exemple:**

```
char nom [50];
int age;
printf (”Saisissez votre nom, puis votre age ” ) ;
scanf (”%s %d ” , nom , &age );
```

- scanf retourne le nombre de variables affectées par la saisie, cela permet de vérifier le bon déroulement de la saisie.

fprintf: écriture formatée sur un flux

int fprintf(FILE *stream, const char *format, ...);

- La liste de variables correspond aux différents types apparaissant dans le format.

Exemple:

```
int age=12 ;
FILE ∗ fichier_sortie;
fichier_sortie= fopen ( ” nomage.txt” , ”w” ) ;
fprintf(fichier_sortie, ”%s%d ” , ” Amal” , age ) ;
```

- En cas de succès, fprintf retourne le nombre de caractères écrits sur le flux.
- printf(format, ...) est équivalent à fprintf(stdout, format,...).

fscanf: lecture formatée sur un flux

int fscanf(FILE *stream, const char *format, ...);

- On remplace les ... par une liste d'adresses de conteneurs mémoires adaptés. On récupère ainsi des données dans le flux qui se place en valeur des adresses de variables données.

Exemple:

```
char nom [80];
int age;
FILE ∗ fichier_entree;
fichier_entree= fopen ( ”nomage .txt”, ”r”);
fscanf (fichier_entree, ”%s %d ”,nom,& age);
```

- fscanf retourne le nombre de variables affectées par la saisie.
- Les flux standards se comportent comme des fichiers ouvert.
- scanf(format, ...) est équivalent à fscanf(stdin, format,...).

fgetc, fputc, fgets et fputs

- **int fgetc(FILE* stream);**
- **int fputc(int c, FILE* stream);**
- servent a lire ou écrire un seul caractère
- retournent **EOF en cas d'erreur**

- **char* fgets(char* s,int n,FILE* f);**
- **fgets lit des caractères jusqu'a:**

• un **\n (qui est copie dans le résultat s)**
• la fin de fichier
• avoir lu **n-1 caractères**

- retourne **NULL en cas d'erreur, s sinon**

- **int fputs(const char* s,FILE* f);**
- **fputs écrit la chaine s dans le fichier f**
- retourne **EOF en cas d'erreur, 0 sinon**

Interprétations

❑ Les noms de fonctions n'ont pas été choisis au hasard.

- print-f: écrire, format
- f-scan-f: flux, analyser, format
- f-get-c: flux, obtenir, caractère
- put-char: placer, caractère
- f-put-s: flux, placer, chaîne de caractères

❑ Lorsque le nom ne commence pas par un f, la fonction affecte les flux standards, sinon elle aura un flux parmi ses arguments.

❑ En fin de nom c et char désigne des caractères, s désigne une chaîne de caractères, f que la fonction peut prendre des écritures formatées (types différents).

sprintf

int sprintf (chaine , const char * format , ...)

- La fonction sprintf réalise le même traitement que la fonction fprintf avec la différence que les caractères émis par sprintf ne sont pas écrits dans un fichier, mais dans une chaîne de caractères.

```
#include<stdio.h>
#include<stdlib.h>
int main() {
char* nom="alami";
char* prenom="samira";
char email[256];
sprintf(email,"%s.%s@yahoo.fr",nom,prenom);
printf("email=%s\n",email);
return 0;
}
```

```
SMI4$./prog_sprintf
email=alami.samira@yahoo.fr
```

sscanf

int sscanf (chaine , const char * format , ...)

- La fonction sscanf réalise le même traitement que la fonction fscanf, avec la différence que les caractères lus par sscanf ne sont pas lus depuis un fichier, mais dans une chaîne de caractères.

```
#include<stdio.h>
#include<stdlib.h>
int main() {
char* date="Mardi Mars 10 14:50:00 2020";
char jours[10],mois[10];
int njours,annee;
sscanf(date,"%s %s %d %*s
%d",jours,mois,&njours,&annee);
printf("%s %d %s %d \n",jours,njours,mois,annee);
return 0;}
```

```
SMI4$./prog_sscanf
Mardi 10 Mars 2020
```

Fin de fichier en lecture

- fscanf : retourne EOF ((End Of File))
- fgetc : retourne EOF
- fgets : retourne NULL
- fread : retourne 0

```
void copy(FILE* src,FILE* dst) {
char buffer[4096];
size_t n;
while ((n=fread(buffer,sizeof(char),4096,src))>0) {
fwrite(buffer,sizeof(char),n,dst);
}}
```

Fin de fichier en lecture (suite)

- int feof(FILE* stream);
- renvoie 0 si la fin de fichier est atteinte.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
char s[400];
FILE* f=fopen("fich.txt","r");
if(f==NULL) exit(1);
while (!feof(f)) {
fscanf(f,"%s",s);
printf(" %s \n",s);}
fclose(f);
return 0;}
```

```
SMI4$ cat fich.txt
abc  smi
20
SMI4$./prog
abc
smi
20
```

Paramètres dans la fonction main

- Le prototype standard d'une fonction **main** contenant des arguments:

int main(int argc, char* argv[])

- argc est un entier, il donne le nombre d'arguments passés à l'exécutable.

- argv est un tableau de chaînes de caractères.

• argv[0]: le nom de l'exécutable (dépend du nom donné à la compilation).
• argv[1]: le nom du premier argument.
• argv[2]: le nom du second argument.
• ...

- Similarité avec le langage du bash:

$# se comporte comme argc
$0 se comporte comme argv[0]
$1 se comporte comme argv[1]

La fonction atoi()

- Cette fonction convertie une chaîne de caractères en une valeur entière «int».

int atoi(const char * theString);

retourne un entier à partir d'une chaîne de caractères.

- **Exemple:**

```
#include <stdio.h>
#include <stdlib.h>    // atoi
int main ( ) {
char  nombre=" 12345 " ;
printf("%d \n", atoi(nombre));
return 0;
}
```

```
SMI4$ gcc -o test test.c
SMI4$ ./test
12345
```

La fonction atof()

- Cette fonction convertie une chaîne de caractères en une valeur flottante «double».

double **atof**(const char * theString);

Exemple:

```
#include <stdio.h>       // printf, fgets
#include <stdlib.h>      // atof
#include <math.h>       // sin
int main () {
double n,m;
char buf[256];
printf ("Entrer  un  nombre ");
fgets (buf,256,stdin);
n = atof(buf);
m = sin(n);
printf ("Le sin de %f  est %f \n" , n, m);
return 0; }
```

Entrer un nombre: 45
Le sin de 45.000000 est 0.850904

La fonction atol()

- Cette fonction convertie une chaîne de caractères en une valeur entière longue «long».

long **atol**(const char * theString);

Exemple:

```
#include <stdio.h>       // printf, fgets
#include <stdlib.h>      // atol
int main () {
long int li;
char buf[256];
printf ("Entrer un nombre long: ");
fgets (buf, 256, stdin);
li = atol(buf);
printf ("La valeur entrée %ld. et le nombre long est %ld.\n",li,li*2);
return 0; }
```

```
Entrer un nombre long: 345567
La valeur entrÚe 345567. et le nombre long est 691134.
Appuyez sur une touche pour continuer...
```

6 GESTION DE LA MÉMOIRE

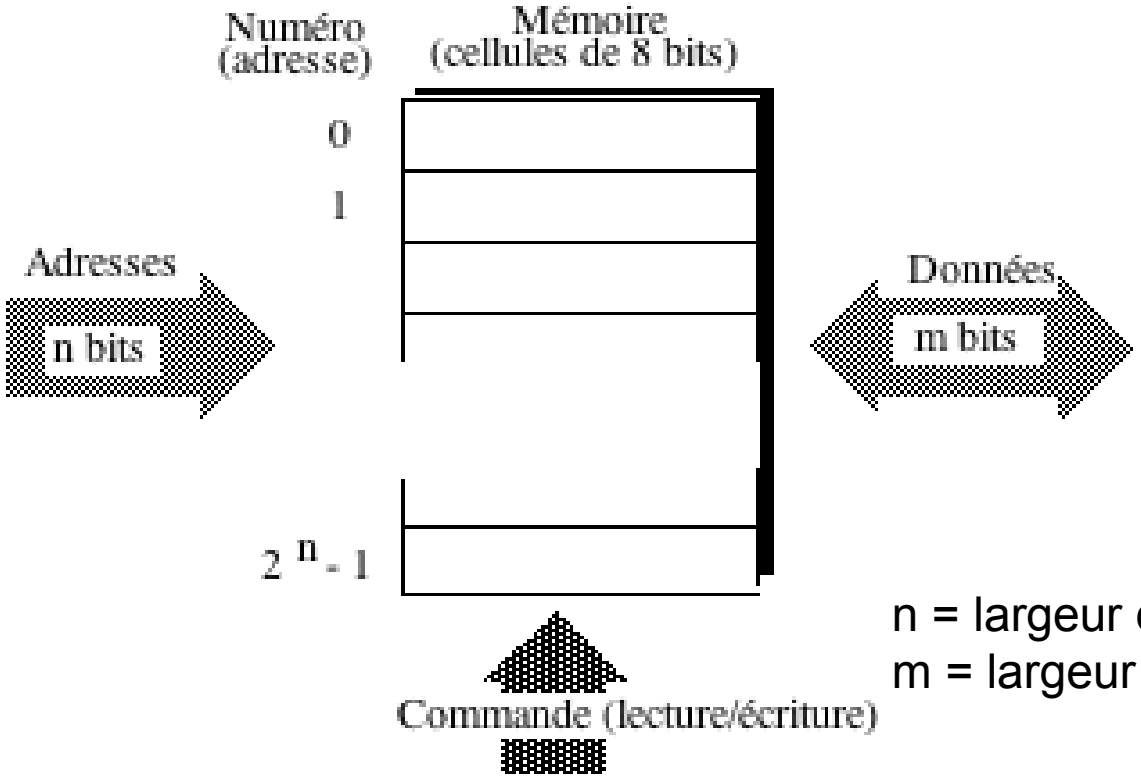
- Pourquoi la mémoire ?
- Notion d'espace d'adressage
- Adresse logique et adresse physique
- Principe de la pagination
- Table des pages
- Protection de l'espace d'adressage des processus
- Protection de l'espace d'adressage
- Le défaut de pages
- Gestion de mémoire sous Linux
- Projection d'un fichier en mémoire centrale

Pourquoi la mémoire ?

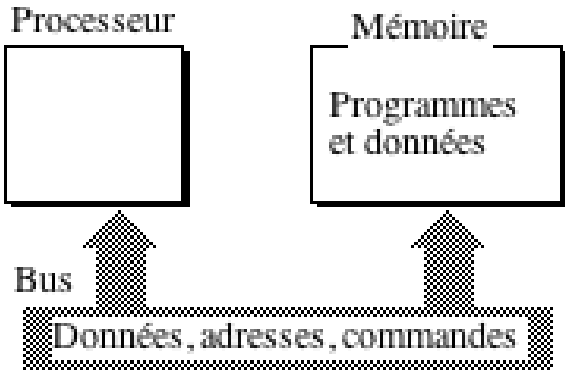
- Le processeur va chercher les informations dont il a besoin dans la mémoire.
- Il ne va jamais les chercher sur le disque, ce dernier sert de support d'archivage aux informations.

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



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 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 constitué, dont la forme dépend de la structuration de cet espace.

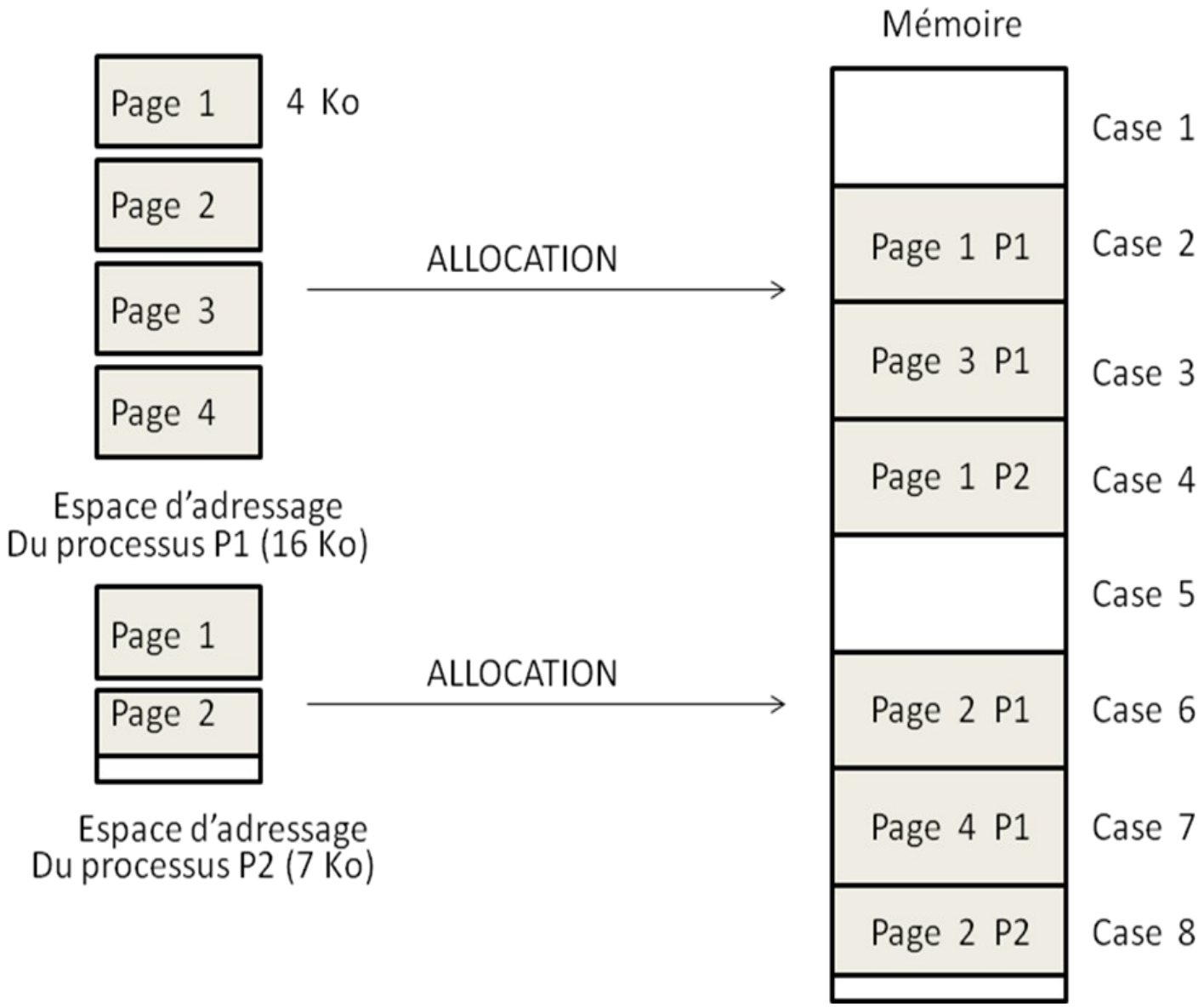
- Cet ensemble d'adresse est appelé **espace d'adresses logiques ou virtuelles**.

Adresse logique et adresse physique

- L'unité centrale manipule des **adresses logiques** (emplacement relatif).
- Les programmes ne connaissent que des adresses logiques.
- L'espace d'adressage logique (virtuel) est 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.

Principe de la 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**.
- Charger un programme en mémoire centrale consiste à placer les pages dans n'importe quelle case disponible.
- Pour connaître à tout moment quelle sont les case libres en mémoire centrale à un instant t.
- Le système maintient une table appelée **table des cases** qui indique pour chaque case de la mémoire physique, si la case est libre ou occupée.


Mémoire
Page 1
4 Ko
Page 2
Page 3
Page 4
ALLOCATION
Espace d'adressage
Du processus P1 (16 Ko)
Page 1
Page 2
ALLOCATION
Espace d'adressage
Du processus P2 (7 Ko)
Case 1
Page 1 P1
Case 2
Page 3 P1
Case 3
Page 1 P2
Case 4
Case 5
Page 2 P1
Case 6
Page 4 P1
Case 7
Page 2 P2
Case 8

	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

Table des pages

- Pour toute opération concernant la mémoire, il faut 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 savoir pour toute page, dans quelle case de la mémoire centrale celle-ci a été placée.
- Cette correspondance se fait grâce à une structure particulière appelée **la table de pages**.
- **La table des pages** est une table contenant autant d'entrées que de pages dans l'espace d'adressage d'un processus.
- Chaque entrée de la table est un couple « numéro de page, numéro de case physique dans laquelle la page est chargée ».

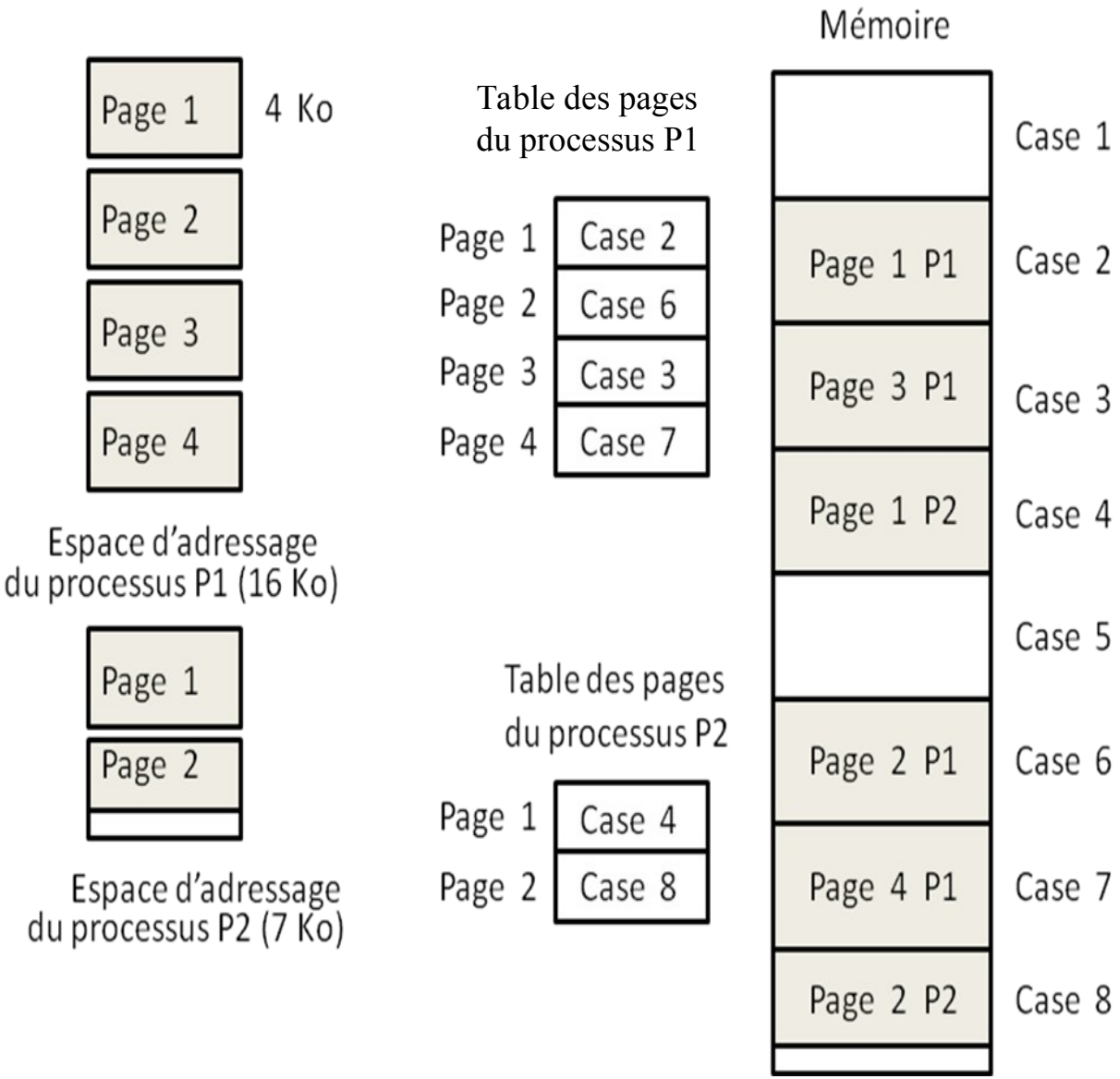

Page 1
4 Ko
Page 2
Page 3
Page 4
Espace d'adressage du processus P1 (16 Ko)
Page 1
Page 2
Espace d'adressage du processus P2 (7 Ko)
Table des pages du processus P1
Page 1 Case 2
Page 2 Case 6
Page 3 Case 3
Page 4 Case 7
Table des pages du processus P2
Page 1 Case 4
Page 2 Case 8
Mémoire
Case 1
Page 1 P1 Case 2
Page 3 P1 Case 3
Page 1 P2 Case 4
Case 5
Page 2 P1 Case 6
Page 4 P1 Case 7
Page 2 P2 Case 8

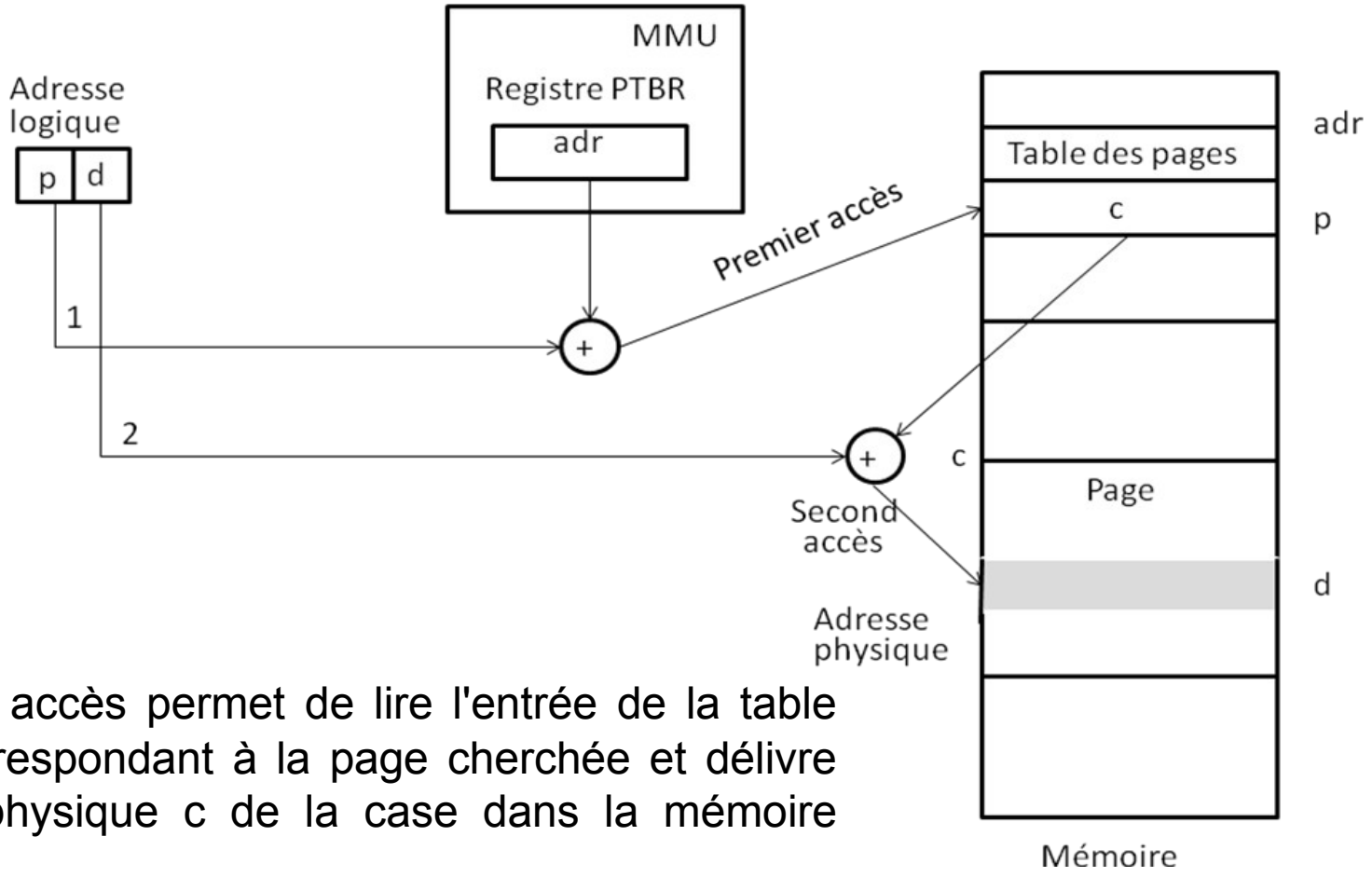
Table des pages (suite)

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

- Pour la réalisation de la table des pages :

• 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).

Table des pages (suite)



- 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.

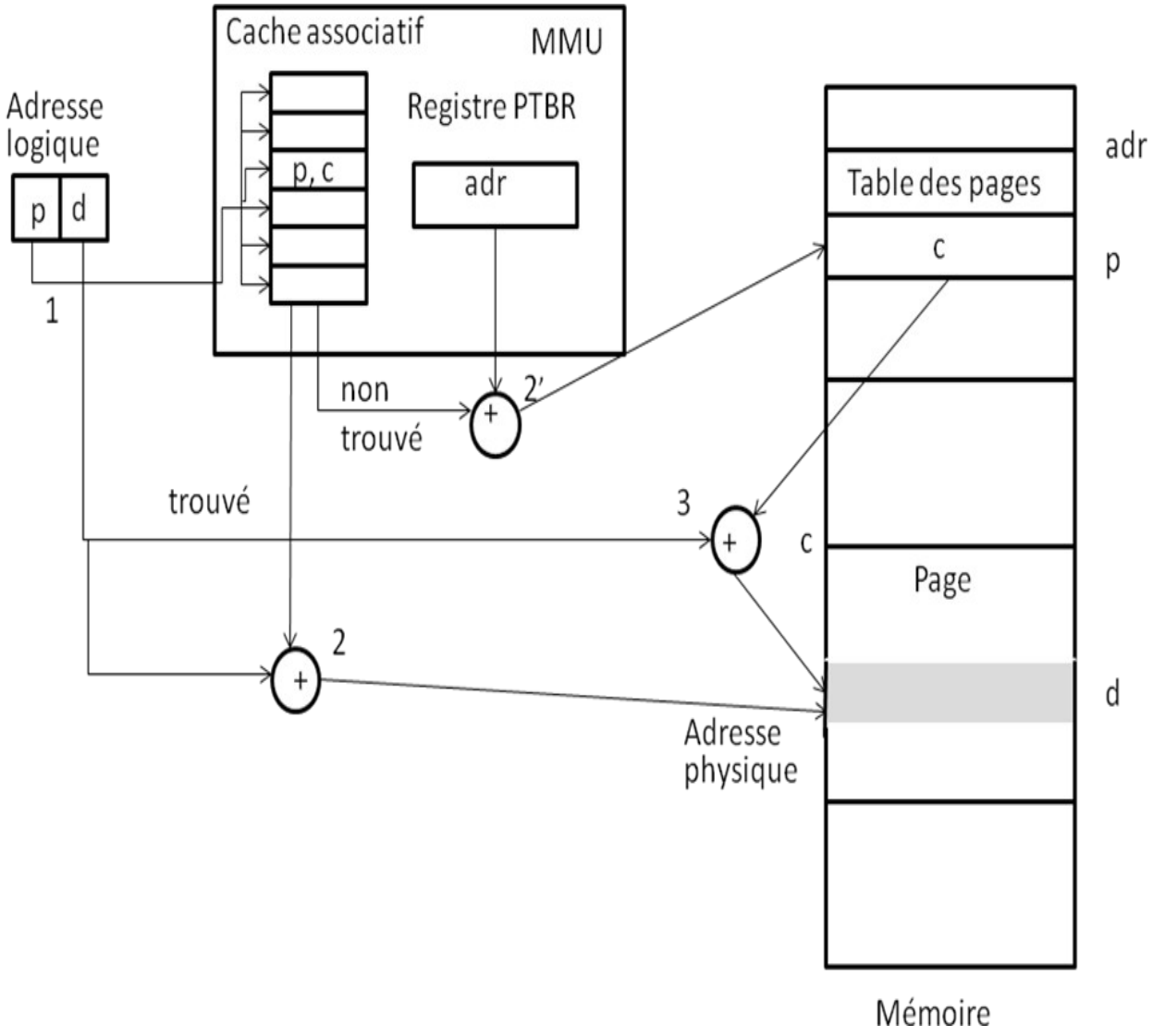
Table des pages (suite)

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

• Un cache associatif 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.

1)Si oui, elle effectue directement la conversion : un seul accès mémoire est alors nécessaire pour accéder à l'octet recherché.
2)Si non, elle accède à la table des pages en mémoire centrale et place le nouveau couple référencé dans le cache.


Cache associatif
MMU
Registre PTBR
Adresse logique
p
d
p, c
adr
1
non trouvé
2'
trouvé
3
2
Table des pages
c
adr
p
c
Page
d
Adresse physique
Mémoire

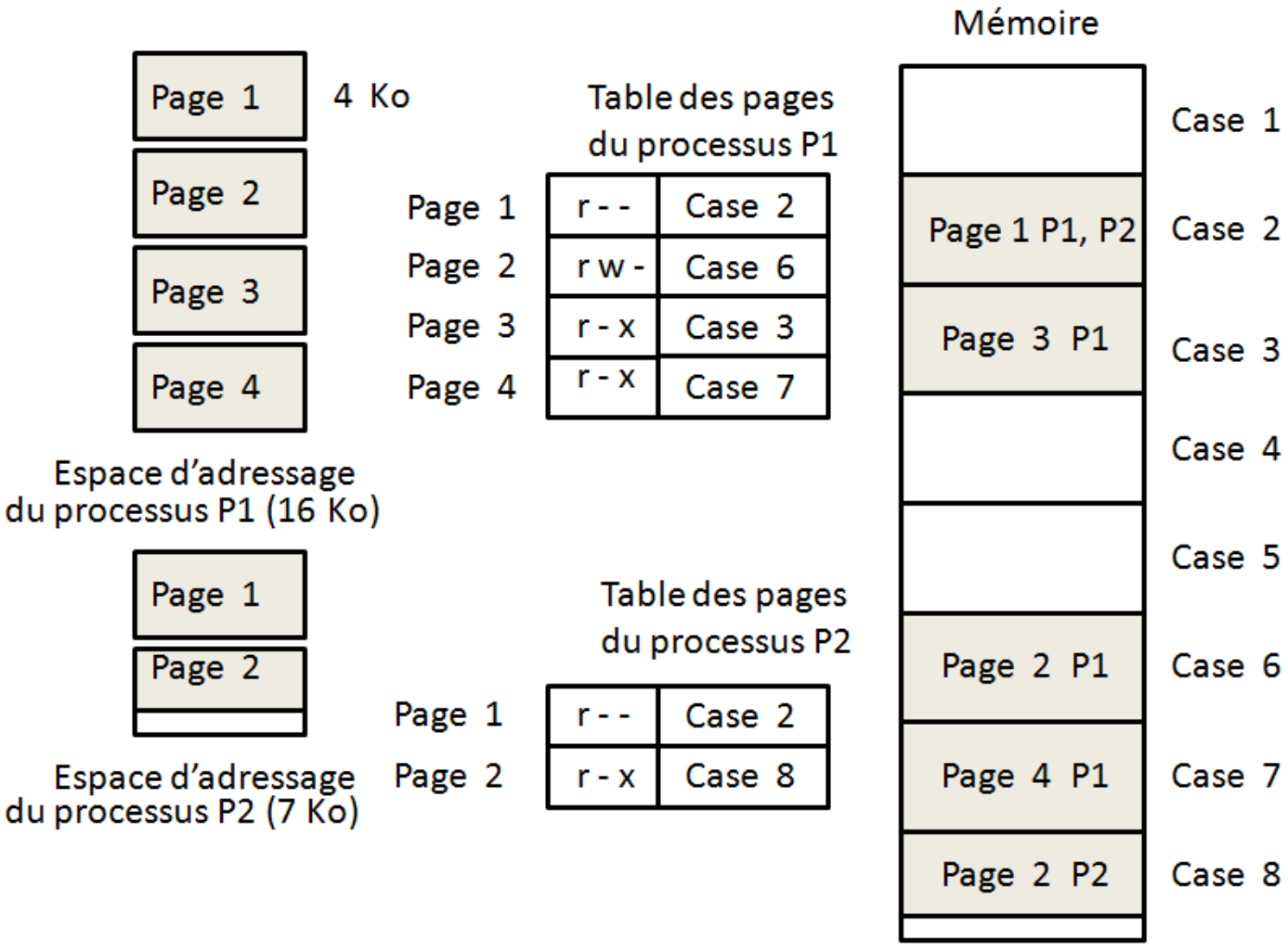
Protection de l'espace d'adressage

- Des bits de protection sont associés à chaque page de l'espace d'adressage du processus.

- Ces bits de protection sont mémorisés pour chaque page, dans la table des pages du processus.

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

- 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.


Mémoire
Page 1
4 Ko
Page 2
Page 3
Page 4
Espace d'adressage du processus P1 (16 Ko)
Table des pages du processus P1
Page 1
Page 2
Page 3
Page 4
r - -
Case 2
r w -
Case 6
r - x
Case 3
r - x
Case 7
Page 1
Page 2
Espace d'adressage du processus P2 (7 Ko)
Table des pages du processus P2
Page 1
Page 2
r - -
Case 2
r - x
Case 8
Case 1
Page 1 P1, P2
Case 2
Page 3 P1
Case 3
Case 4
Case 5
Page 2 P1
Case 6
Page 4 P1
Case 7
Page 2 P2
Case 8

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

Table des cases

Principe de la mémoire virtuelle

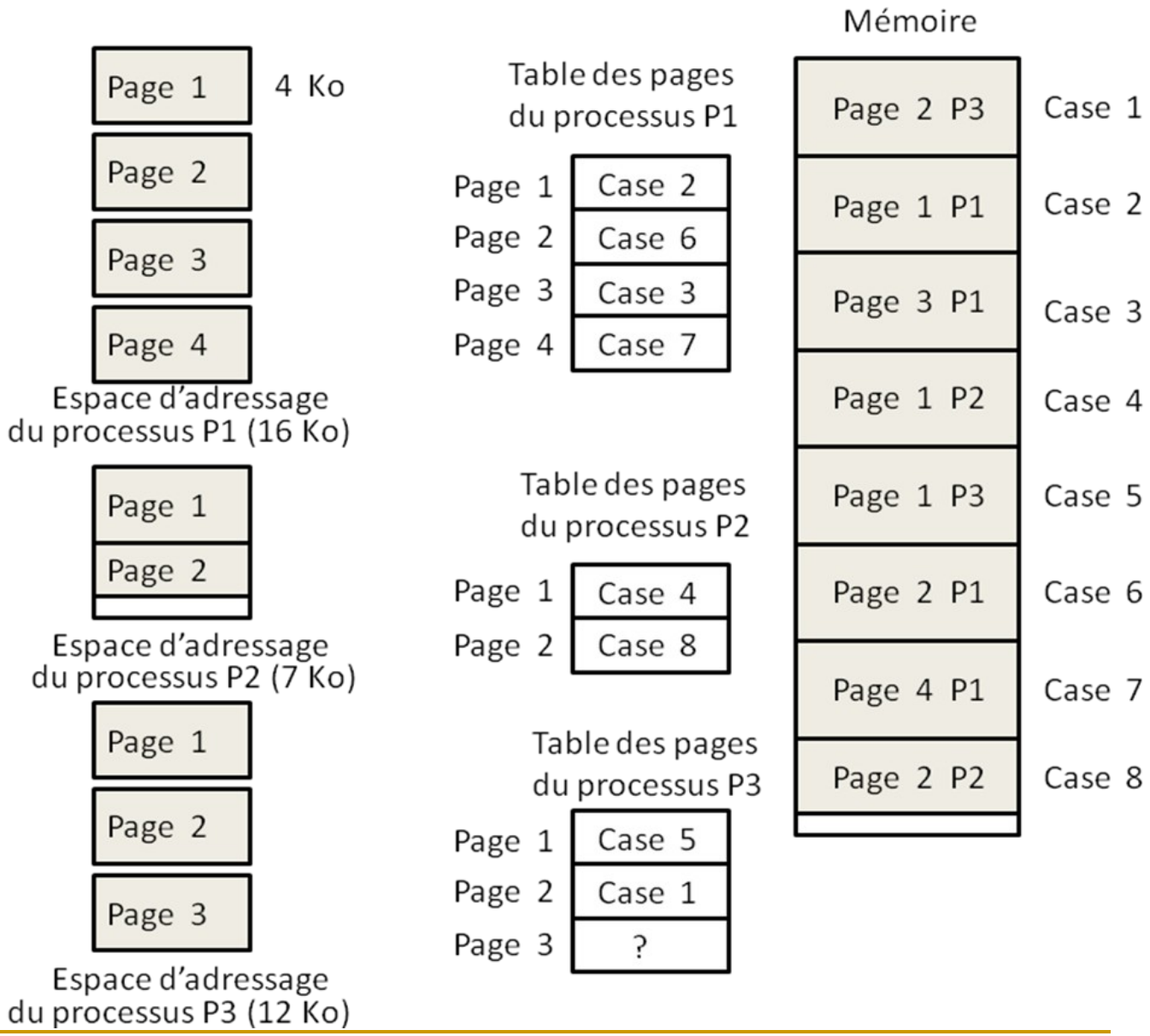
- La multiprogrammation implique de charger plusieurs programmes en mémoire centrale de manière à obtenir un bon taux d'utilisation du processeur.

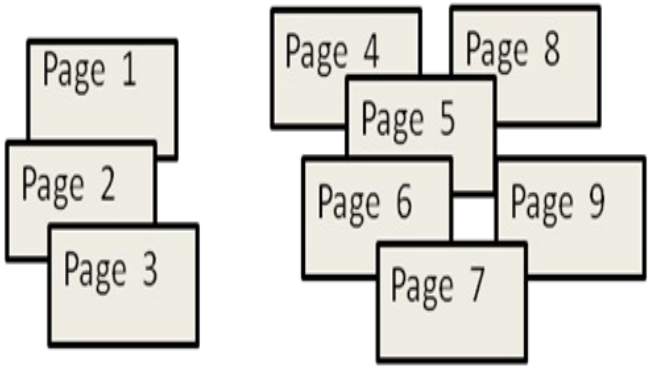
- On peut remarquer qu'une fois les pages des processus P1 et P2 chargés dans la mémoire, toutes les cases sont occupées : le programme P3 ne peut pas être chargé.

- 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.

- 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.

Insuffisance de la mémoire centrale





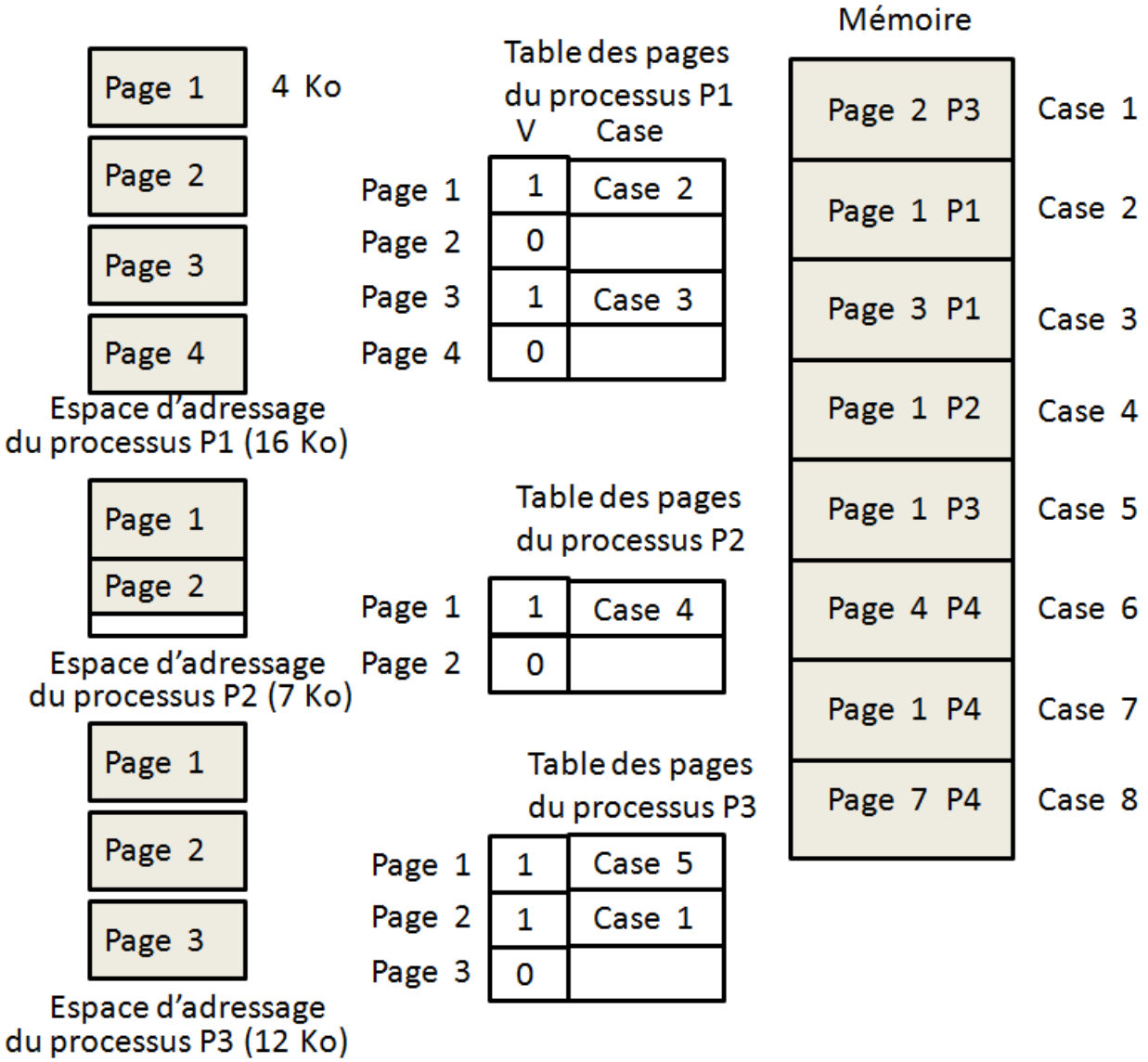
Espace d'adressage du processus P4 (36Ko)

	Page	Processus
Case 1	2	P3
Case 2	1	P1
Case 3	3	P1
Case 4	1	P2
Case 5	1	P3
Case 6	2	P1
Case 7	4	P1
Case 8	2	P2

Table des cases

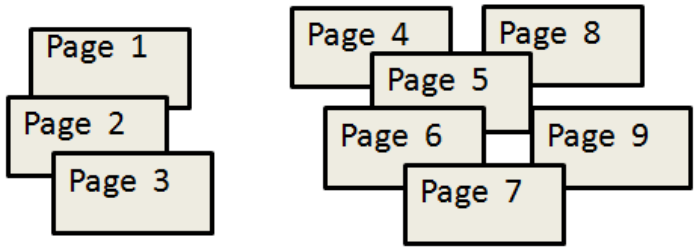
Mémoire virtuelle

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.



	Page	Processus
Case 1	2	P3
Case 2	1	P1
Case 3	3	P1
Case 4	1	P2
Case 5	1	P3
Case 6	4	P4
Case 7	1	P4
Case 8	7	P4

Table des cases



Espace d'adressage du processus P4 (36Ko)

Page 1	1	Case 7
Page 2	0	
Page 3	0	
Page 4	1	Case 6
Page 5	0	
Page 6	0	
Page 7	1	Case 8
Page 8	0	
Page 9	0	

Table des pages du processus P4

Le défaut de pages

- 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 égal à 0 ce qui veut dire que la page n'est pas chargée dans une case et donc la conversion ne peut être réalisée.

- Il se produit alors un **défaut de page**.

Le remplacement de pages

- La totalité des cases de la mémoire centrale peut être occupée : il faut donc libérer une case de la mémoire.

- Le SE utilise un algorithme pour choisir une case à libérer.

- Les deux principaux algorithmes sont :

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

Remplacement FIFO

- C'est la page la plus anciennement chargée qui est remplacée.

- **Exemple:**

on suppose qu'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		

Remplacement LRU

- C'est la page la moins récemment accédée qui est remplacée.

- **Exemple:**

On suppose qu'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	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		

Gestion de mémoire sous 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.

cat /proc/[pid]/maps (man proc) donnera la liste des régions du processus [pid].

Gestion de mémoire sous Linux (suite)

```
SMI4> 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
```

Gestion de mémoire sous Linux (suite)

La première région est le code du programme exécuté.
La deuxième région contient les données du programme exécuté .
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, avec une adresse virtuelle 0x40000000 sur x86.

Les autres régions sont le code et les données des bibliothèques partagées.

La dernière région correspond à la pile.

Gestion de mémoire sous Linux (suite)

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).

Gestion de mémoire sous Linux (suite)

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.

Gestion de mémoire sous Linux (suite)

Exemple :

L'écriture des pointeurs fournit des adresses virtuelles. 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 ;
}
```

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. 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.

Projection d'un fichier en mémoire centrale (suite)

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 fichier. 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 contenu.
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;
char *adresse, nom[10];
fd=open("/home/SMI4/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;}}
```

7 COMMUNICATION ENTRE PROCESSUS

- IPC
- Les principaux signaux
- Envoyer un signal
- Manipuler un ensemble de signaux
- Bloquer les signaux
- La primitive signal
- La primitive sigaction
- Qu'est ce qu'un tube anonyme?
- Création, écriture, lecture et la fermeture d'un tube anonyme
- Rediriger les entrées-sorties vers des tubes
- Les tubes nommés
- Création et ouverture d'un tube nommé

IPC (Inter-Process Communications)

- Les processus interagissent entre eux et doivent donc pouvoir communiquer entre eux.
- Le système doit pouvoir avertir les processus en cas de défaillance d'un composant du système.
- L'utilisateur doit pouvoir gérer les processus (arrêt ,suspension,..).
- Il existe de multiples moyens de réaliser une communication inter-processus par

• fichiers
• Signaux
• Sockets
• Files d'attente de message
• Pipe/tubes
• Named pipe/tubes nommés
• Sémaphores
• Mémoire partagée
• …

Les principaux signaux

- Un signal traduit l'apparition d'un événement asynchrone à l'exécution d'un processus.
- Un programme utilisateur peut envoyer un signal avec la fonction ou la commande kill, ou encore par certaines combinaison de touches au clavier (comme Ctrl-C).
- Un utilisateur ne peut envoyer un signal qu'à un processus dont il est propriétaire.
- Un signal est représenté dans le système par un nom de la forme SIGXXX.

Les principaux signaux (suite)

Nom du signal	Numéro du signal	Evénement associé
SIGHUP	1	Terminaison du processus leader de la session
SIGINT	2	Interruption au clavier (par Ctrl-C par défaut)
SIGQUIT	3	Quit par frappe au clavier (par Ctr − AltGr − \ par défaut)
SIGILL	4	Détection d'une instruction illégale
SIGTRAP	5	Point d'arrêt pour le debug mode (non POSIX)
SIGABRT	6	Terminaison anormale
SIGBUS	7	Erreur de bus
SIGFPE	8	Exception de calcul flottant (division par 0 racine carrées d'un nombre négatif, etc...)
SIGKILL	9	Processus tué (kill)
SIGUSR1	10	Signal 1 défini par l'utilisateur
SIGSEGV	11	Référence mémoire invalide
SIGUSR2	12	Signal 2 défini par l'utilisateur
SIGPIPE	13	Ecriture dans un tube sans lecteur
SIGALRM	14	Horloge temps réel, fin de temporisation
SIGTERM	15	Signal de terminaison
SIGSTKFLT	16	Erreur de pile du coprocesseur
SIGCHLD	17	Terminaison d'un fils

Les principaux signaux (suite)

SIGCONT	18	Reprise du processus
SIGSTOP	19	Suspension du processus
SIGTSTP	20	Émission vers le terminal du caractère de suspension (Ctrl Z).
SIGTTIN	21	Lecture par un processus en arrière-plan
SIGTTOU	22	Écriture par un processus en arrière-plan.
SIGURG	23	Données urgentes sur une socket
SIGXCPU	24	Limite de temps CPU dépassé
SIGXFSZ	25	Limite de taille de fichier dépassé
SIGVTALARM	26	Alarme virtuelle (non POSIX)
SIGPROF	27	Alarme du profileur (non POSIX)
SIGWINCH	28	Fenêtre redimensionnée (non POSIX)
SIGIO	29	Arrivée de caractère à lire (non POSIX)
SIGPOLL	30	Equivalent à SIGIO (non POSIX)
SIGPWR	31	Chute d'alimentation (non POSIX)
SIGUNUSED	32	Non utilisé

Envoyer un signal

La méthode la plus générale pour envoyer un signal est d'utiliser soit la commande shell kill(1), soit la fonction C kill(2).

La commande kill

La commande kill prend une option -signal et un pid.

Exemple :

```
$ kill -SIGINT 14764  # interrompe processus de pid 14764
$ kill -SIGSTOP 22765 # stoppe temporairement le processus 22765
$ kill -SIGCONT 22765 # reprend l'exécution du processus 22765
```

La fonction kill

Envoi d'un signal à n'importe quel processus

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

Retourne 0 si OK, -1 si erreur

Envoyer un signal (suite)

- Si pid>0, le signal num_sig est envoyé au processus pid.
- Si pid=0, le signal num_sig est envoyé à tous les processus appartenant au même groupe que le processus appelant.
- Si pid=-1, le signal num_sig est envoyé à tous les processus, sauf le processus (init ale pid 1) et le processus appelant.
- Si pid<-1, le signal num_sig est envoyé au groupe de processus identifié par la valeur absolue de pid.

Erreur : **EINVAL**, **ESRCH**, **EPERM**
EINVAL : Signal invalide
ESRCH : Le processus ou groupe de processus n'existe pas.
EPERM : Le processus ne possède pas de droits suffisants pour envoyer le signal.

Envoyer un signal (suite)

Exemple : Le processus père teste l'existence de son fils avant de lui envoyer le signal SIGUSR1.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
int main(void) {
pid_t pid,pidBis;
int status;
printf("Lancement du processus %d \n",getpid());
switch(pid = fork())
{
case -1: exit(1);
```

Envoyer un signal (suite)

```
case 0:
while(1) sleep(1);exit(1);
default:
printf("Processus fils %d creé \n",pid);
sleep(10);  /* On attend 10 secondes */
if(kill(pid,0) == -1) {
printf("processus fils %d inexistant \n",pid);
exit(1);
}
else {
printf("envoi de SIGUSR1 au fils %d \n",pid);
kill(pid,SIGUSR1);
}
pidBis=wait(&status);
printf("Mort de fils %d avec status=%d \n",pidBis,status);
exit(0);
}}
```

Manipuler un ensemble de signaux

Les variables de type **sigset_t** , ne sont manipulables que par ces fonctions :

```
#include <sys/types.h>
#include <signal.h>
int sigemptyset (sigset_t *ens_signaux);
/* permet de vider l'ensemble de signaux ens_signaux */
int sigfillset (sigset_t * ens_signaux);
/*permet de remplir l'ensemble de signaux ens_signaux avec tous les signaux du système */
int sigaddset (sigset_t * ens_signaux, int signum);
/* permet d'ajouter le signal num_sig à l'ensemble de signaux ens_signaux */
```

Manipuler un ensemble de signaux (suite)

int **sigdelset** (sigset_t * ens_signaux, int signum);
/* permet de retrancher le signal num_sig à l'ensemble de signaux ens_signaux */
int **sigismember** (const sigset_t * ens_signaux, int signum);
/* permet de savoir si le signal num_sig appartient à l'ensemble de signaux ens_signaux */

Les quatre premières fonctions renvoient 0 en cas de succès et -1 sinon.
La dernière fonction renvoie 1 si le signal num_sig est présent dans l'ensemble, 0 sinon et -1 en cas d'erreur.

Bloquer les signaux

La fonction **sigprocmask()** permet à un processus de bloquer (ou de débloquer) un ensemble de signaux, à l'exception des signaux SIGKILL et SIGCONT.

```
#include <signal.h>
int sigprocmask( int action,  const  sigset_t ens_signaux, sigset_t
*ancien);
```

Le paramètre **action** permet d'indiquer la nature de l'opération voulue.
Si action=SIG_BLOCK, l'ensemble des signaux bloqués est égal aux signaux déjà bloqués auxquels s'ajoutent les signaux placés dans l'ensemble ens_signaux ;
Si action=SIG_UNBLOCK, les signaux placés dans l'ensemble ens_signaux sont retirés de l'ensemble des signaux bloqués ;
Si action=SIG_SETMASK, l'ensemble des signaux bloqués devient égal à l'ensemble ens_signaux.

Bloquer les signaux (suite)

La fonction renvoie 0 en cas de succès et -1 sinon. La variable **errno** prend l'une des deux valeurs suivantes :

- **EFAULT** si un pointeur invalide**,**
- **EINVAL** si action est invalide

La fonction **sigpending()** permet de connaître l'ensemble des signaux pendants qui sont bloqués par le processus. Le prototype de la fonction est :

int sigpending(sigset_t *ens);

La fonction **sigsuspend()** permet de façon atomique de modifier le masque des signaux et de se bloquer en attente. Une fois un signal non bloqué délivré, la primitive se termine en restaurant le masque antérieur :

int sigsuspend(const sigset_t *ens);

Un exemple d'utilisation

Le processus bloque le signal SIGINT durant son travail qui consiste la réalisation d'un calcul numérique.
Si le caractère « CTRL C » a été frappé au moins une fois au clavier durant l'exécution du calcul numérique, le signal SIGINT est délivré suite à son déblocage et entraîne la terminaison du processus.
Les traces d'exécution sont donc :
Je vais faire mon travail tranquillement
SIGINT pendant

Si le caractère « CTRL C » n'a pas été frappé au clavier durant l'exécution du calcul numérique, les traces d'exécution sont par contre :
Je vais faire mon travail tranquillement
Signaux débloqués

Un exemple d'utilisation (suite)

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
main() {
int i;
sigset_t ens, ens1;
sigemptyset(&ens);
sigaddset(&ens, SIGINT);
sigprocmask(SIG_SETMASK, &ens, 0);
printf("je vais faire mon travail tranquillement \n");
i=0 ;
while(i<1000000000){
i=i+1 ;}
sigpending(&ens1);
if (sigismember(&ens1, SIGINT))
printf("SIGINT pendant \n");
sigemptyset(&ens);
sigprocmask(SIG_SETMASK, &ens, 0);
printf("signaux débloqués \n");
}
```

La primitive signal

- L'appel système **signal()**. L'emploi de cette primitive est simple mais son utilisation peut poser des problèmes de compatibilité avec les différents systèmes UNIX ;

#include <signal.h>
void (*signal(int num_sig, void (*func)(int)))(int);
Retourne le pointeur de fonction du signal si OK, la constante **SIG_ERR** si erreur.

• Le premier argument num_sig est le numéro du signal .
• Le second argument handler définie par *func est attaché au signal num_sig, désigné par le nom qui lui associé. Trois possibilités pour cet argument func :
func=SIG_DFL, alors l'action par défaut est attaché au signal num_sig;
func=SIG_IGN, alors le signal num_sig est ignoré;
func est un pointeur sur la fonction chargée de gérer le signal envoyé.

La primitive signal (suite)

Un exemple 1: Ce handler affiche la valeur k ayant provoqué la violation : reçu signal 11 pour k=910

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
void segv();
int k,l;
char buf[1];
int main() {
/* mise en place traitement signal "violation" */
if(signal(SIGSEGV, segv)==SIG_ERR)
{ perror("signal");
exit(1); }
/* boucle devant mener à une violation de la segmentation mémoire */
for(k=0 ;;++k)
l=buf[k] ; }
void segv(sig)
int sig ;
 { printf(“reçu signal %d pour k=%x \n”,sig,k);
exit (0); }
```

La primitive signal (suite)

Un exemple 2: Le programme suivant «prog_sig0.c » montre l'utilisation du masquage des signaux.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include<unistd.h>
void sighandler(int sig);
int main(){
char buffer[256] ;
if(signal(SIGTERM, sighandler)==SIG_ERR){
perror("Ne peut pas manipuler le signal \n");
exit(1);}
while(1){
scanf("%s",buffer);
printf("Input : %s \n",buffer);}
return 0 ;}
void sighandler(int sig){
printf("Masquage du signal SIGTERM \n");}
```

La primitive signal (suite)

- Les primitives sleep(), pause() et alarm() peuvent être utiles lorsqu'on travaille avec des signaux.

```
#include <unistd.h>
int pause(void);
int sleep(unsigned int seconds);
unsigned int alarm(unsigned int seconds);
```

- La primitive **sleep (int seconds)** permet de bloquer (état endormi) le processus courant sur la durée passée en paramètre.

- La primitive **pause()** bloque en attente le processus appelant jusqu'à l'arrivée d'un signal.

- La primitive **alarm()** permet à un processus d'armer une temporisation. A l'issue de cette temporisation le signal SIGALRM est délivré au processus. Le comportement par défaut est d'arrêter l'exécution du processus.

La primitive signal (suite)

Un exemple 3: Le programme « prog_sig1.c » montre l'utilisation du signal SIGINT avec pause() pour éviter la charge de CPU.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int nsig = 0;
void traiter_sig(){
nsig++;
printf ("\n Signal CTRL_C\n");}
int main(){
if ( signal(SIGINT, traiter_sig) == SIG_ERR){
printf ("Signal");
exit (1);}
while (nsig< 5)
pause();   /*attente NON active*/
printf ("%d signaux SIGINT sont arrivés\n", nsig);
return 0;}
```

La primitive sigaction

- L'appel système **sigaction()**.

#include <signal.h>
int **sigaction**(int signum, const struct sigaction *nouv_action, const struct sigaction *anc_action);
Retourne 0 si OK, -1 si erreur

Le premier paramètre représente le signal à dérouter.
Les deuxième et troisième paramètres sont respectivement le nouveau et l'ancien comportement à adopter à l'arrivée du signal.

Si le pointeur nouveau est différent de NULL, alors le système modifie l'action du signal signum.
Si le pointeur ancien est différent de NULL, alors le système retourne l'action précédente qui était prévue pour le signal signum.
Si les pointeurs nouveau et ancien sont NULL, l'appel système teste uniquement la validité du signal.

La primitive sigaction (suite)

- L'emploi de cette fonction **sigaction()** est plus complexe, elle présente un comportement fiable, normalisé par POSIX.

- La structure **sigaction** à la forme suivante :

struct sigaction {
void (*sa_handler()) ();
/* Fonction de déroutement, SIG_IGN ou SIG_DFL*/
sigset_t sa_mask;
/* Liste des signaux qui sont bloqués durant l'exécution de la fonction de déroutement */
unsigned long sa_flags;
/* Options définissant le comportement du processus à la réception du signal*/
};

La primitive sigaction (suite)

Exemple: Le processus effectuant la violation mémoire crée un fils, qui fait pour sa part un calcul numérique. Ce fils est tué par son père dans le gestionnaire de signal levé par la violation mémoire.

```
#include <stdio.h>
#include<stdlib.h>
#include <signal.h>
void segv();
int k,l, pid;
char buf[1];
int main()
{
struct sigaction action ;
int i,j ;
/* mise en place traitement signal "violation" */
sigemptyset(&action.sa_mask) ;
action.sa_flags=0 ;
action.sa_handler=segv ;
sigaction(SIGSEGV, &action, 0) ;
```

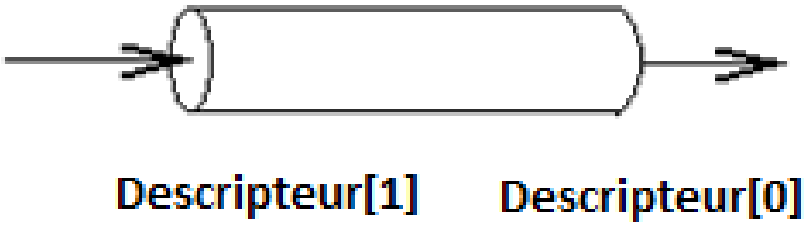
La primitive sigaction (suite)

```
pid=fork() ;
if(pid==0) {
printf(“je suis le fils \n”);
i=0 ;
while(i<1000){
i=i+1 ;}
exit(0) ;}
else
/*boucle devant mener à une violation de la segmentation mémoire */
for(k=0 ;;++k)
l=buf[k] ;
}
/* procédure de traitement de la réception du signal */
void segv(sig)
int sig ;{
printf(“reçu signal %d pour k=%x \n”,sig,k);
kill(SIGKILL,pid);
exit (0);}
```

Qu’est ce qu’un tube anonyme?

Un **tube** peut être représenté comme un tuyau dans lequel circulent des informations.
Les tubes servent à faire communiquer plusieurs processus entre eux.

Ecrit des informations dans le tube. Celui-ci est appelé **entrée** du tube ;



Lit les informations dans le tube. Il est nommé **sortie** du tube.

Création d'un tube anonyme

On crée un tube par un appel à la fonction pipe, déclarée dans unistd.h :

int pipe(int descripteur[2]);

La fonction renvoie 0 si elle réussit, et elle crée alors un nouveau tube.
La fonction **pipe** remplit le tableau descripteur passé en paramètre, avec :
descripteur[0] désigne la sortie du tube;
descripteur[1] désigne l'entrée du tube;
Le principe est qu'un processus va écrire dans descripteur[1] et qu'un autre processus va lire les mêmes données dans descripteur[0].

Ecriture et lecture dans d'un tube anonyme

Pour écrire dans un tube, on utilise la fonction **write** :

int write(int descripteur[1], char *bloc, int taille);

descripteur doit correspondre à l'entrée d'un tube
bloc est un pointeur vers la mémoire contenant ces octets
taille est le nombre d'octets qu'on souhaite écrire

Pour lire dans un tube, on utilise la fonction read :
int read(int descripteur[0], char *bloc, int taille);

descripteur doit correspondre à la sortie d'un tube
bloc pointe vers la mémoire destinée à recevoir les octets
taille donne le nombre d'octets qu'on souhaite lire
La fonction renvoie le nombre d'octets lus. Si cette valeur est inférieure à taille, c'est qu'une erreur s'est produite en cours de lecture.

Ecriture et lecture dans d'un tube anonyme (suite)

Exemple : Pour écrire le message "Bonjour" dans un tube, depuis le père.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include<sys/wait>
int main(){
int desc_p[2];
pid_t pid_fils ;
char messageEcr[7] ;
pipe(desc_p) ;
pid_fils = fork();
if(pid_fils != 0)  /* Processus père */
{
sprintf(messageEcr, "bonjour");
/* La fonction sprintf permet de remplir une chaîne de caractère avec un texte donné */
write(desc_p [1], messageEcr, 7);}
return 0;}
```

Ecriture et lecture dans d'un tube anonyme (suite)

Exemple : Pour lire un message envoyé par le père à un fils.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include<sys/wait>
int main(){
int desc_p[2];
pid_t pid_fils ;
char messageLir[7] ;
pipe(desc_p) ;
pid_fils = fork();
if(pid_fils == 0)   /* Processus fils */
{read(desc_p [0], messageLir, 7);
printf("Message réçu= %s",messageLir);}
return 0;}
```

Fermeture d'un tube anonyme

La primitive close (fd) permet de fermer le descripteur fd.
int close (int fd) ;
Exemple : Pour fermer l'entrée du processus fils et la sortie du processus père.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void){
pid_t pid_fils;
int desc_p[2];
pipe(desc_p);
pid_fils = fork();
if(pid_fils == 0)  /* Processus fils */
close(desc_p[1]);
else  /* Processus père */
close(desc_p[0]);
return 0;}
```

Exemple d'utilisation

Exemple de communication entre processus par l'intermédiaire d'un tube.
Le processus fils écrit à destination de son père la chaîne de caractères « bonjour ».
Les étapes du programme sont:

- Le processus père ouvre un tube en utilisant la fonction pipe().
- Le processus père crée un fils en utilisant la fonction fork().
- Les descripteurs en lecture et écriture du tube sont utilisables par les 2 processus. Chacun des deux processus ferme le descripteur en écriture et l'autre ferme le descripteur en lecture.
- Le fils lit le message.
- Le père envoie un message à son fils.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include<sys/wait>
int main(){
int pip[2],status ;
pid_t pid_fils ;
char chaine[7] ;
pipe(pip) ;
pid_fils =fork() ;  /*Processus fils*/
if(pid_fils==0) { printf(" Fermeture de l'entrée dans le fils. \n") ;
close(pip[1]) ;
read(pip[0], chaine,7);
```

```
printf(“Nous sommes dans le fils (pid=%d).\n a réçu
le message suivant du père : %s \
n",getpid(),chaine) ;
exit(0);}
else {
printf("\n Fermeture de la sortie dans le père. \n") ;
close(pip[0]) ;
sprintf(chaine,"bonjour”);
printf(“Nous sommes dans le père (pid=%d).\n
envoie le message suivant au fils : %s \
n",getpid(),chaine) ;
write(pip[1], chaine,7);
wait(&status);}
return 0 ;}
```

Rediriger les entrées-sorties vers des tubes

- On peut lier la sortie d'un tube à stdin ou l'entrée d'un tube à **stdout**.
- Toutes les informations qui sortent du tube arrivent dans l'entrée standard et peuvent être lues avec scanf, fgets, etc...
- Toutes les informations qui sortent par **stdout** sont écrites dans le tube. On peut utiliser printf, puts, etc...
- Il suffit d'utiliser la fonction :

int dup2(int ancienDescripteur, int nouveauDescripteur);

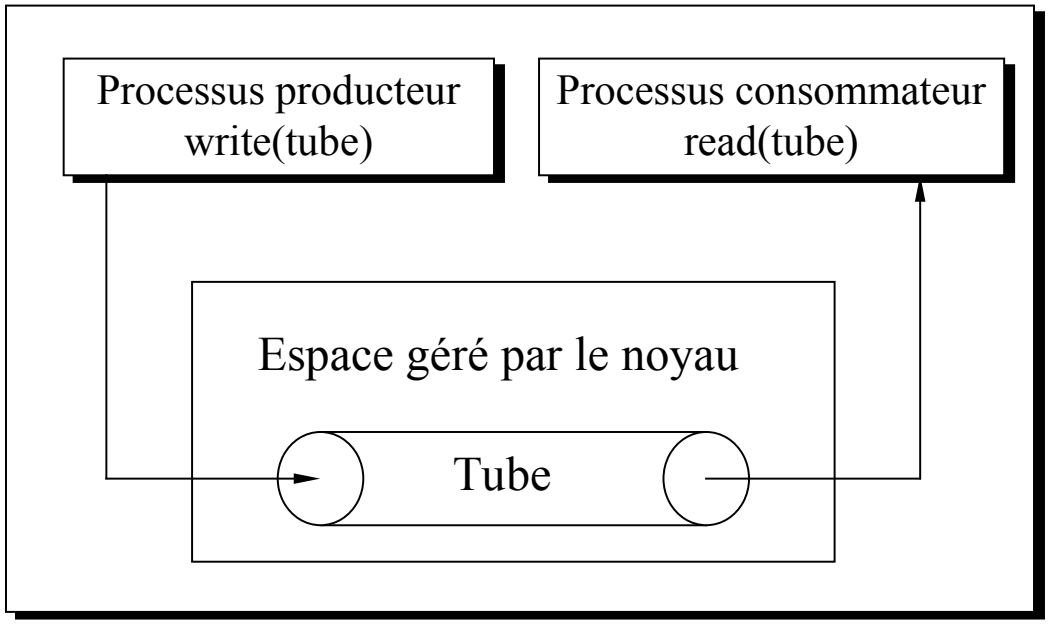
• Le premier paramètre est :
descripteur[0] si on veut lier la sortie,
descripteur[1] si on veut lier l'entrée.
• Le second paramètre correspond au nouveau descripteur que l'on veut lier au tube.
Il existe deux constantes, déclarées dans unistd.h :
STDIN_FILENO
STDOUT_FILENO
On utilise STDIN_FILENO lorsqu'on veut lier la sortie à stdin
STDOUT_FILENO lorsqu'on veut lier l'entrée à stdout

Les tube nommés

Ils correspondent à un fichier (inode) dans lequel les processus pourront lire et écrire.

Cela implique opération sur périphérique mais deux processus **indépendants** pourront se transmettre des informations.

Les opérations autorisées sur un tube sont la création, la lecture, l'écriture, et la fermeture.



Création d'un tube nommés

Créer un tube nommé par fonction mkfifo de la bibliothèque sys/stat.h :
int mkfifo (const char* nom, mode_t mode);
Renvoie 0 en cas de succès et -1 dans le cas contraire.
nom du tube nommé
mode correspond aux droits d'accès associés au tube
S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXU, activation respective des droits en lecture, écriture, exécution et lecture/écriture/exécution pour le propriétaire de fichier.
S_IRGRP, S_IWGRP, S_IXGRP, S_IRWXG, activation respective des droits en lecture, écriture, exécution et lecture/écriture/exécution pour le groupe du propriétaire de fichier.
S_IROTH, S_IWOTH, S_IXOTH, S_IRWXO, activation respective des droits en lecture, écriture, exécution et lecture/écriture/exécution pour les autres.
Ces constantes peuvent être combinées entre elles à l'aide de l'opération de OU binaire « | ».

Ouverture d'un tube nommés

❑ Pour utiliser un tube nomme crée, il est nécessaire de l'ouvrir en lecture ou en écriture avec open :

```
int open(const char *nom,int mode_ouverture) ;
```

• **nom** est nom du fichier
• Le second argument indique si c'est l'entrée ou la sortie du tube. Il existe deux constantes pour cela, déclarées dans fcntl.h :
O_WRONLY : pour l'entrée,
O_RDONLY : pour la sortie.

Exemple : Pour ouvrir l'entrée d'un tube « essai » :

```
descripteur[1] = open("essai", O_WRONLY);
```

Ensuite, nous pouvons écrire et lire avec write et read comme si c'était des tubes classiques.

Exemple d'utilisation

Le premier programme ecrivain.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include<sys/stat.h>
#include<fcntl.h>
int main(){
int fd ;
mode_t mode ;
mode=S_IRUSR | S_IWUSR ;
char message[7] ;
mkfifo("fictub",mode) ;
sprintf(message, "bonjour");
fd=open("fictub",O_WRONLY);
printf("ici l'écrivain [%d] \n",getpid());
write(fd,message,7) ;
close(fd) ;
return 0 ;}
```

Le deuxième programme lecteur.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include<sys/stat.h>
#include<fcntl.h>
int main(){
int fd ;
char message[7] ;
fd=open(“fictub”,O_RDONLY);
printf(“ici le lecteur [%d] \n”,getpid());
read(fd,message,7) ;
message[7]=0 ;
printf(“%s”,message);
return 0 ;}
```