


**Cours « système d'exploitation »
2^{ème} année
IUT de Caen, Département d'Informatique
Année 2000 – 2001
(François Bourdon)**



Chapitre 1
Système de Gestion des Fichiers :
Concepts Avancés
(partie 3)

Plan

1. Système de Gestion des Fichiers : Concepts avancés

1.1 Représentation interne du SGF

1.2 Les E/S et le SGF

1.3 E/S tamponnées : le Buffer Cache

1.4 Le système de fichiers virtuel (SFV)

1.5 Appels système et SGF

1.6 Cohérence d'un SGF

1.7 Le système de fichiers "/proc"

1.8 Monter un SGF

1.9 SGF et caractéristiques physiques d'un disque

1.10 Organisation classique d'un SGF

2. Création et Ordonnancement de Processus

3. Synchronisation de Processus

4. ...

1.5 Appels Systèmes et SGF

Introduction

Certaines structures de données du SGF ne sont pas accessibles directement ; elles nécessitent le recours aux « appels système » (*system calls*).

Pour rendre le système robuste, tout appel système par un programme en C (ou en un autre langage) n'accède pas directement aux fonctions du système d'exploitation.

Le système différencie donc les appels système des applications de ceux du traitement dans le noyau.

Les appels systèmes sont détournés au moyen d'une table dans le noyau. Lors d'un appel système, l'adresse de la fonction correspondante du système d'exploitation est cherchée dans cette table.

Traitement des erreurs

Tous les appels système importants retournent une valeur de type *int*, dont l'interprétation est commune :

- une valeur de 0 ou plus indique que l'appel s'est déroulé correctement ;
- -1 signale une erreur.

La variable *extern int errno* fournit, sous forme de numéros d'erreur, des informations complémentaires au processus appelant.

Soit l'exemple suivant :

```
# include <errno.h>  
int main (void) {  
    ...  
    fd = open ("abc.txt", 0, 0644);  
    if (fd > 0) { close (fd); }  
    else {  
        perror ("Fichtre, qu'en ai-je fait ?");  
        return 1 ;  
    }  
    return 0 ;  
}
```

Si le fichier « abc.txt » n'existe pas, cela déclenche une erreur qui s'affichera à l'écran de la façon suivante :

```
$ prog
```

```
Fichtre qu'en ai-je fait ? : No such file or directory
```

```
$
```

Eléments de base

Un fichier est une suite d'**articles** ou d'**enregistrements logiques** d'un certain type, qui ne peuvent être manipulés qu'au travers d'opérations spécifiques. Elles sont traduites par le système d'exploitation en d'autres opérations agissant sur l'espace mémoire. On trouve au minimum :

créer un fichier,
ouvrir un fichier (en lecture ou en écriture),
fermer un fichier,
détruire un fichier,
pointer au début d'un fichier,

ou encore :

éditer le contenu d'un fichier,
copier un fichier dans un autre fichier,
renommer un fichier.

Certaines de ces opérations correspondent à la manipulation de zones de la mémoire secondaire. Par exemple, la destruction d'un fichier se traduit par la libération d'une ou de plusieurs zones (blocs).

L'utilisateur dispose également d'opérations lui permettant d'accéder aux informations contenues dans un fichier :

lire un article,
écrire dans un article,
modifier un article,
insérer un article,
détruire un article,
retrouver un article.

Les entrées/sorties de fichiers sont réalisées par les routines : **open, close, read, write, lseek ...**

Sur le plan de la programmation, ces routines ressemblent beaucoup aux appels de bibliothèques. Mais au contraire des fonctions de bibliothèques qui sont définies dans le standard C ANSI, ces appels sont spécifiques à UNIX (et Linux).

Utilisation de ces appels :

1. Ouvrir un fichier par les appels **creat** et **open**.
2. Réaliser des E/S par les appels **read** et **write**, ainsi que des positionnement avec **lseek**.
3. Fermer le fichier par l'appel **close** ; pas utile en fin de programme, car les fichiers ouverts par un processus, sont automatiquement fermés en fin de processus.

Les deux appels **open** et **creat** retournent une valeur de type **int**, qui représente le descripteur de fichier. **Ce descripteur est l'indice dans de la table de fichier spécifique au processus.** Sa valeur doit être 0 ou plus.

Deux définitions/interfaces possibles pour
"open" :

```
int open (char * chemin, int indicateurs_états)
```

```
int open ( char * chemin, int indicateurs_états, int droits-d'accès)
```

La syntaxe de l'appel **open()** est la suivante :

```
valeur := open (ref, type, droits);
```

où

<i>ref</i>	→ un nom de fichier (ex. "abc.txt"),
<i>type</i>	→ le type d'ouverture (lecture, écriture ou création),
<i>droits</i>	→ les droits d'accès donnés au fichier créé,
<i>valeur</i>	→ un entier est retourné dans la variable <i>valeur</i> , c'est le descripteur de fichier qui est un indice dans la table des descripteurs ; il sera utilisé dans les accès ultérieurs au fichier.

Exemple

```
#include <stdio.h>
#include <fcntl.h>
int main (int argc, char **argv) {
    int fd ;
    if (argc < 2) {
        fprintf(stderr," ERREUR : aucun paramètre\n");
        return 1;
    }
    fd=open(argv[1],O_WRONLY|O_CREAT|O_TRUNC,0644);
    if (fd == -1) {
        perror ("Ouverture");
        return 2;
    }
}
```

Sur cet exemple le fichier est ouvert en écriture (O_WRONLY). Si le fichier existe déjà on supprime le contenu (O_TRUNC), sinon on le crée (O_CREAT) avec le droit « 644 ».

Les sémaphores O_RDONLY, O_WRONLY et O_RDWR s'excluent mutuellement.

La valeur choisie peut se combiner avec une ou plusieurs valeurs possibles (O_CREAT, O_TRUNC, O_EXCL ...), on utilise pour cela un OU logique "|".

Définitions/interfaces des fonctions "**read**" et "**write**" :

```
int read (int fd, char *tampon, unsigned int nombre)
int write (int fd, char *tampon, unsigned int nombre)
```

L'opération de lecture, **read**, s'utilise de la manière suivante :

```
nombre := read (valeur, tampon, compteur);
```

où

valeur → la variable définie plus haut (int fd),
tampon → l'adresse (char *) de la structure de données qui recevra les informations lues et
compteur → le nombre (int) d'octets que l'utilisateur voudrait lire ; *nombre* reçoit le nombre d'octets réellement lus.

L'opération d'écriture **write** s'utilise de la même façon.

Exemple

```
#include <stdio.h>
#include <fcntl.h>
#define BLKSIZE 1024

int main (int argc, char **argv) {

    int fd_in, fd_out ;
    char buffer [BLKSIZE];
    int n;

    if (argc != 3) {
        fprintf(stderr, " ERREUR : arguments %s
                                incomplets\n, argv[0]");
        return 1;
    }
    fd_in=open(argv[1],O_RDONLY);
    if (fd == -1) {
        perror ("Ouverture du fichier d'entrée");
        return 2;
    }
    fd_out=open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,
                                0644);

    if (fd_out == -1) {
        perror ("Ouverture du fichier de sortie");
        return 2;
    }
    while ((n=read(fd_in, buffer,BLKSIZE)) > 0)
        write(fd_out, buffer, n);

    close (fd_in);
    close (fd_out);

}
```

La fonction **read** détecte la fin de fichier et renvoie -1 .

Dans l'algorithme *open*, on met-à-jour dans la table des fichiers, le déplacement octet courant à partir duquel le noyau est en attente du premier *read()* ou *write()*. Ce déplacement est nul pour un *open* ordinaire et est égal à la taille du fichier pour une ouverture en mode *append* (concaténation $\langle = \rangle$ « \rangle » dans le shell).

algorithme open

entrée: (1) nom de fichier

(2) type d'ouverture

sortie: descripteur de fichier

```
{  
    convertir le nom du fichier en un i-noeud (namei);  
    if(fichier inexistant ou accès non permis)  
        return (erreur);  
    attribuer à l'i-noeud un élément de la table des fichiers,  
    initialiser le compte et le déplacement octet;  
    attribuer un descripteur de fichier utilisateur,  
    positionner un pointeur vers l'élément associé  
    de la table des fichiers;  
    if(type d'ouverture spécifie une troncature du fichier)  
        libérer tous les blocs du fichier (algo. free);  
    déverrouiller l'i-noeud; /* verrouiller dans namei */  
    return (descripteur de fichier);  
}
```

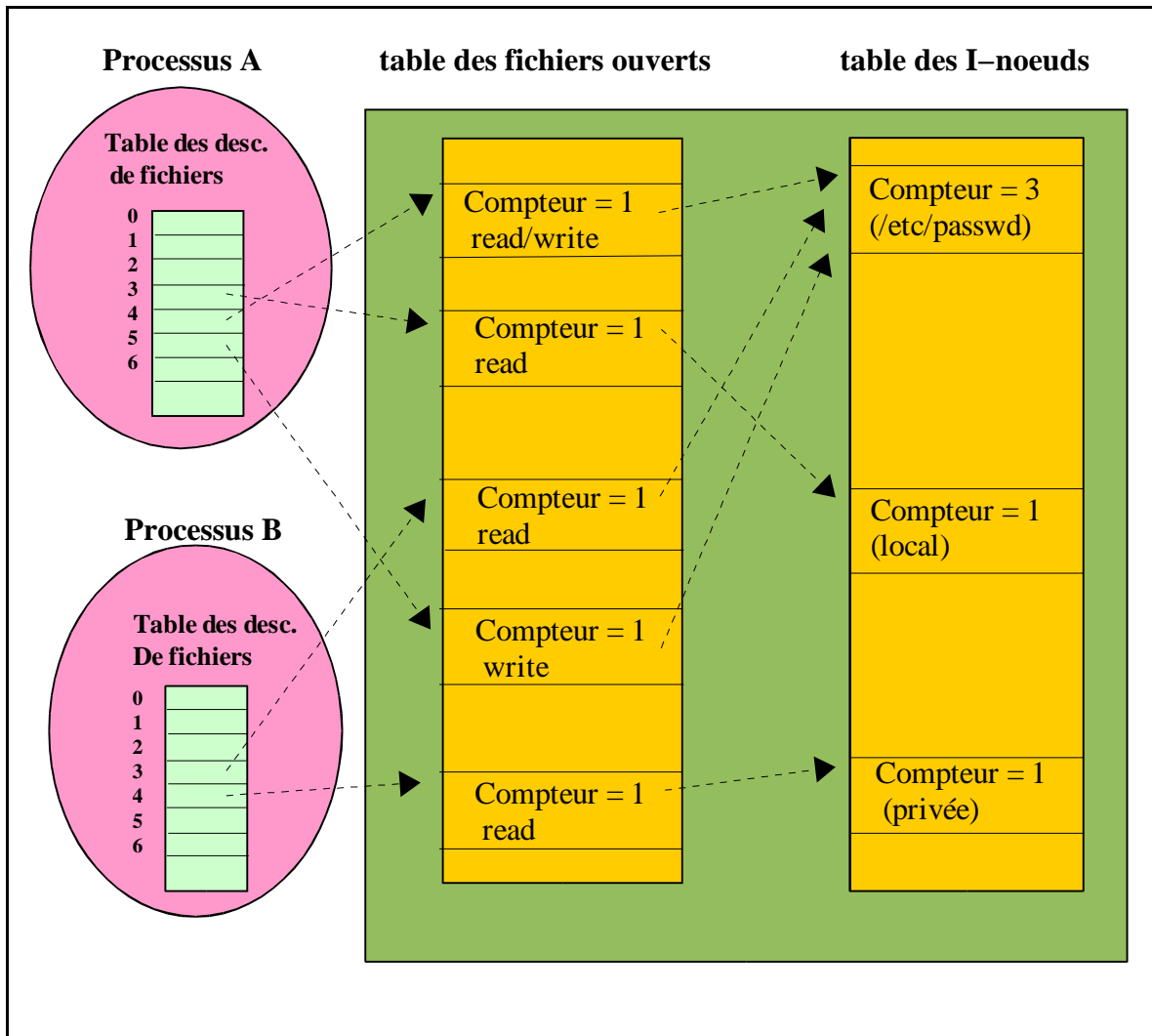
Les trois premières entrées de la table des descripteurs de fichier dans chaque processus sont automatiquement allouées et réservées dès la création, pour les fichiers suivants :

entrée 0 : entrée standard (stdin)

entrée 1 : sortie standard (stdout)

entrée 2 : erreur standard (stderr)

Lors de la création d'un processus (fork), celui-ci hérite de la table des descripteurs de fichiers du père.



Processus A :

```
fd1 = open ("local", O_RDONLY);
fd2 = open ("/etc/passwd", O_WRONLY);
fd2 = open ("/etc/passwd", O_RDONLY);
```

Processus B :

```
fd1 = open ("/etc/passwd", O_RDONLY);
fd2 = open ("privée", O_RDONLY);
```

Utilisation des Répertoires

Pour plus de souplesse il est préférable d'utiliser les appels "**opendir**", "**readdir**" et "**closedir**" lors de manipulation de fichiers répertoires.

La structure "**struct dirent**" (contenue dans "**dirent.h**") contient les deux composants suivants :

```
ino_t d_ino
char d_name [ ]
```

Définitions/interfaces des fonctions "**opendir**"
"**readdir**" et "**closedir**" :

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(char *répertoire)
struct dirent *readdir(DIR *pointeur_de_répertoire)
int closedir(DIR pointeur_de_répertoire)
```

Il n'existe pas de fonction "**writedir**" qui serait trop dangereuse pour la cohérence du SGF. Pour modifier le contenu d'un fichier répertoire il faut procéder indirectement par l'utilisation des fonctions "**open**", "**creat**", "**mkdir**" ou encore "**mknod**".

Il existe aussi les fonctions "**chdir**", et "**rmdir**".

Gestion des Fichiers Périphériques

Les entrées/sorties vers les fichiers périphériques utilisent les mêmes appels système que les fichiers classiques.

Il y a quelques différences pour les périphériques ayant une interface série ou les lignes de terminaux.

Procédure habituelle :

1. Ouvrir la ligne (appel "**open**").
2. Lire et écrire au travers de la ligne (appels "**read**" et "**write**"). Un placement par l'appel "**lseek**" n'as aucun effet. L'appel "**ioctl**" peut modifier les paramètres de la ligne.
3. Fermer la ligne (appel "**close**").

Appels de fichier étendus

Il existe plusieurs appels système pour connaître et modifier les droits d'accès aux fichiers : **"access"**, **"stat"** et **"fstat"**.

"access"

L'appel **"access"** permet de vérifier si le processus en cours peut accéder au fichier visé.

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

Avec pour modes de fonctionnement (2ème paramètre) :

R_OK	Vérification du droit de lecture
W_OK	Vérification du droit d'écriture
X_OK	Vérification du droit d'exécution
F_OK	Vérification de l'existence

Lorsque l'accès vérifié est autorisé, l'appel renvoie 0, sinon -1.

"stat / fstat"

Les appels "stat" et "fstat" ne diffèrent que par leurs paramètres. Ils rassemblent les informations les plus importantes de l'i-noeud d'un fichier, dans une structure ("struct stat").

```
#include <unistd.h>
#include <sys/stat.h>
int stat (char *chemin, struct stat *tampon_stat)
int fstat (int fd, struct stat *tampon_stat)
```

La structure "stat" retournée est la suivante :

dev_t st_dev	N° du périphérique contenant le fichier.
ino_t st_ino	N° d'i-noeud.
mode_t st_mode	Type du fichier et droits d'accès.
nlink_t st_nlink	Nombre de liens (links).
uid_t st_uid	Identificateur de l'utilisateur du fichier.
gid_t st_gid	Identificateur de groupe du fichier.
off_t st_size	Taille en octets du fichier.
time_t st_atime	Heure du dernier accès.
time_t st_mtime	Heure de la dernière modification.
time_t st_ctime	Heure de la dernière modification de l'état, changé par les appels chmod, chown, creat, link, mknod, pipe, unlink, utime et write.

1.6 Cohérence du SGF

Le noyau ordonne ses écritures sur disque de façon à minimiser la corruption des systèmes de fichiers lors d'une panne du système. La commande *fsck* vérifie de telles incohérences et répare le système de fichiers si cela est nécessaire.

fsck se compose de 5 phases :

1. test sur les i-noeuds,
2. test par les répertoires,
3. test de connexion inter-répertoires,
4. test du nombre des références,
5. test des blocs libres (S-V) et des groupes de cylindre (BSD).

Détail des 5 phases de "fsck"

1. Test sur les i-noeuds : contrôle des blocs et des tailles

Technique -> A partir de chaque i-noeud on parcourt tous les blocs correspondants.

Objectif -> contrôler la taille de chaque fichier et marquage des blocs parcourus (bons).

Type d'erreur -> taille incorrecte (bloc de données utilisé par deux fichiers, déjà marqué "bon" à la 2nde rencontre).

2. Test par les répertoires : contrôle des chemins

Technique -> On parcourt les répertoires (arborescence).

Objectif -> Vérifier que les doublons (nom, i-noeud) sont corrects.

Type d'erreur -> Il existe un i-noeud vide dans un doublon, ou le type du i-noeud est mauvais.

3. Test de connexion inter-répertoire : contrôle de la connectivité

Technique -> Vérifier que chaque i-noeud est référencé quelque-part.

Objectif -> Il doit exister un répertoire qui référence chaque i-noeud.

Type d'erreur -> Un fichier est perdu.

4. Test du nombre des références : compte des liens

5. Test des blocs libres

Technique -> Addition entre la somme des blocs libres et la somme des blocs occupés.

Objectif -> On doit obtenir la taille du disque.

Type d'erreur -> La taille du disque est mauvaise (pb grave !) ou des blocs sont perdus ou en trop (correction possible).

Exemples d'incohérences

- ➔ Un bloc disque peut appartenir à plus d'un i-noeud ou à la liste des i-noeuds libres et à un i-noeud. Lorsque le noyau libère un bloc disque d'un fichier, il retourne le numéro de bloc à la copie mémoire du super bloc et alloue le bloc disque à un nouveau fichier. Si le noyau a écrit l'i-noeud et les blocs du nouveau fichier sur le disque mais est tombé en panne avant de mettre à jour l'i-noeud de l'ancien fichier sur disque, les deux i-noeuds adresseront le même numéro de bloc disque. De même, si le noyau a écrit le super bloc et sa liste des blocs libres sur le disque et est tombé en panne avant que l'ancien i-noeud ne soit sauvegardé, le bloc disque apparaîtra dans la liste des blocs libres et dans l'ancien i-noeud.
- ➔ Si un numéro de bloc n'est pas dans la liste des blocs libres ni utilisé par un fichier, le système de fichier est incohérent car tous les blocs doivent apparaître quelque part.
- ➔ Un i-noeud peut avoir un compte lien différent de 0, mais son numéro d'i-noeud peut ne pas exister dans aucun des répertoires du système de fichiers.
- ➔ Si le format d'un i-noeud est incorrect (ex. type du fichier indéfini). Ceci peut arriver si l'administrateur a monté un système de fichiers mal formaté. Le noyau accède aux blocs disques en pensant qu'ils contiennent des i-noeuds alors qu'en réalité ils contiennent des données.
- ➔ Si un numéro de i-noeud apparaît dans un élément d'un répertoire alors que l'i-noeud est libre.
- ➔ Si le nombre de blocs libres ou d'i-noeuds libres enregistrés dans le super bloc n'est pas conforme au nombre de blocs existants sur le disque.

```
$ fsck [options] sys_fic
```

On dispose également de la fonction "**e2fsck**", qui correspond à la fonction spécifique appelée par "**fsck**" pour le système de fichier natif "**ext2**" de LINUX.

Sans argument "**fsck**" lit le fichier "**/etc/fstab**" où sont décrits l'ensemble des "File Systems" (partitions) à vérifier.

Un rang de passage permet de lancer les vérifications en parallèle sur les FS qui sont sur des disques différents.

"**fsck**" est appelée systématiquement au **boot**.

Certaines incohérences sont corrigées automatiquement :
ex. les fichiers vides non référencés sont détruits ;
pour le reste un dialogue s'établit avec l'administrateur.

Après "**fsck**" le FS est cohérent, mais il peut y avoir eu perte d'information.

Une analyse manuelle de "**/lost+found**" peut permettre de récupérer certains fichiers perdus et stockés dans ce répertoire.

1.7 Le Système de Fichiers `"/proc"`

Ce système de fichiers ne correspond pas à des emplacements sur le disque dur, il constitue un lien logique entre de nombreuses informations du noyau du système d'exploitation.

Les fichiers "virtuels" représentent l'accès aux structures de données internes du noyau.

```
[$HOME/]$ cd /proc/
[/proc] $ ls
1      353   5    587 660 720 cmdline  kcore  mtrr    swaps
1043 364   512 594 661 946 cpuinfo  kmsg   net     sys
1044 380   529 595 665 967 devices ksyms  parport tty
1045 396   546 6   667 968 dma     loadavg partitions uptime
1046 4     581 603 677 969 fb      locks  pci     version
127  412   582 604 679 970 filesystems mdstat rtc
151  428   583 619 681 971 fs      meminfo scsi
2    435   584 638 687 972 ide     misc   self
3    447   585 644 688 apm interrupts modules slabinfo
300  481   586 646 689 bus ioports mounts  stat
[/proc] $ more filesystems
ext2
nodev   proc
        iso9660
nodev   devpts
[/proc] $
```

Il existe un [répertoire](#) par processus en cours du nom du PID du processus correspondant.

Le répertoire [net](#) contient des informations liées au réseau, [scsi](#) aux périphériques SCSI, et [sys](#) aux variables du noyau et [self](#) est un lien vers le répertoire correspondant au processus courant.

Contenu du répertoire associé au processus 1104 (bash courant).

```
[/proc] $ echo $$
1104
[/proc]$ cd 1104
[/proc/1104]$ ls -l
total 0
-r--r--r--  1 francois users    0  Sep 28 14:54 cmdline
lrwx-----  1 francois users    0  Sep 28 14:54 cwd -> /proc/1104
-r-----  1 francois users    0  Sep 28 14:54 environ
lrwx-----  1 francois users    0  Sep 28 14:54 exe -> /bin/bash
dr-x-----  2 francois users    0  Sep 28 14:54 fd
pr--r--r--  1 francois users    0  Sep 28 14:54 maps
-rw-----  1 francois users    0  Sep 28 14:54 mem
lrwx-----  1 francois users    0  Sep 28 14:54 root -> /
-r--r--r--  1 francois users    0  Sep 28 14:54 stat
-r--r--r--  1 francois users    0  Sep 28 14:54 statm
-r--r--r--  1 francois users    0  Sep 28 14:54 status
[francois@er145 1104]$
```

où,

"cmdline"	-> liste des arguments du processus 1104
"environ"	-> liste des variables d'environnement de 1104
"fd"	-> répertoire contenant des liens sur les fichiers ouverts par le processus 1104
"maps"	-> liste des zones mémoires contenues dans l'espace d'adressage de 1104.
"mem"	-> contenu de l'espace d'adressage du processus
"stat", "statm" et "status"	-> état du processus.

Contenu du fichier "status" décrivant l'état du processus 1104 :

```
[/proc/1104]$ more status  
Name: exe  
State: R (running)  
Pid: 1104  
PPid: 1071  
Uid: 101 101 101 101  
Gid: 100 100 100 100  
Groups: 100  
VmSize: 1768 kB  
VmLck: 0 kB  
VmRSS: 1008 kB  
VmData: 156 kB  
VmStk: 12 kB  
VmExe: 340 kB  
VmLib: 1196 kB  
SigPnd: 0000000000000000  
SigBlk: 000000000010002  
SigIgn: 000000000384004  
SigCgt: 000000007813efb  
CapInh: 0000000ffffeff  
CapPrm: 0000000000000000  
CapEff: 0000000000000000  
[/proc/1104]$
```

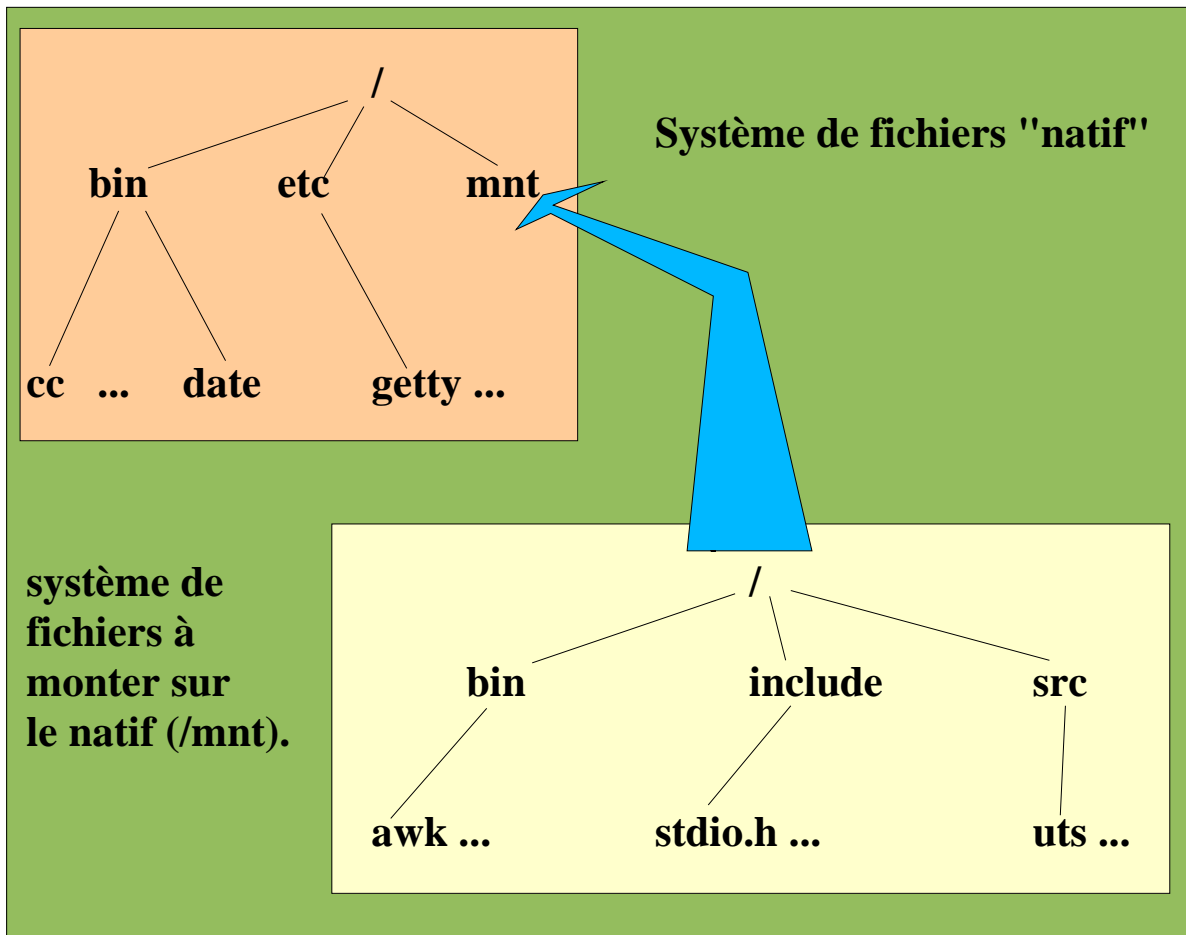
1.8 Monter et démonter un SGF

On appelle système de fichiers (file-system) l'espace disque initialisé au moyen de la commande *mkfs*. On désigne également par ce nom le logiciel du noyau UNIX qui assure la gestion de cet espace disque.

L'espace disque en question est soit le disque en entier, soit une partition du disque.

Une fois initialisé (création du super bloc et des parties dédiées aux blocs de données et aux i-noeuds), un système de fichiers est attaché à un répertoire du système de fichiers principal (désigné par /) au moyen de la commande *mount*. Il peut en être détaché au moyen de la commande *umount*.

Les utilisateurs accèdent donc aux données d'une partition disque à travers un système de fichiers au lieu de la considérer comme une suite de blocs disque.



mount (chemin-d'accès-au-FS-à-monter, point-de-montage-dans-le-répertoire-courant, options);

options indique si le répertoire doit être monté en « lecture-seulement ».

On peut réaliser des montages à travers le réseau. Seul l'administrateur ("**root**") peut utiliser cette commande.

Une table des volumes montés garde la trace des différents périphériques ; elle permet d'associer les disques à l'arborescence du système de fichiers. Ainsi l'utilisateur peut parcourir les répertoires sans en supposer leur organisation physique. Cette table contient :

Un numéro de périphérique monté.
Un pointeur sur un tampon qui contient le super-bloc du système de fichiers.
Un pointeur sur l'i-noeud racine du système de fichiers monté (« / » du système de fichiers « /dev/dsk1 » sur le schéma).
Un pointeur sur l'i-noeud du répertoire qui est le point de montage (« usr » du système de fichiers racine sur le schéma).

Le numéro de périphérique identifie le système de fichiers monté ; c'est le numéro du système de fichiers logique mentionné précédemment.

Des connections peuvent être réalisées automatiquement au démarrage du système. Pour cela il suffit de compléter le fichier **"/etc/fstab"** :

```
[$HOME/]$ more /etc/fstab
/dev/hda5      /          ext2      defaults  1 1
/dev/hda6      /home     ext2      defaults  1 2
/dev/hda7      /usr      ext2      defaults  1 2
/dev/hda8      swap      swap      defaults  0 0
/dev/fd0       /mnt/floppy ext2      owner,noauto 0 0
/dev/cdrom     /mnt/cdrom iso9660   owner,noauto,ro 0 0
/dev/sdb1     /mnt/zip  vfat     noauto    0 0
none          /proc     proc     defaults  0 0
none          /dev/pts  devpts   gid=5,mode=620 0 0
er175:/tmp    /mnt      nfs      noauto    0 0
[$HOME/]$
```

La **colonne 1** désigne les périphériques en cas de montage à travers le réseau (la dernière ligne de l'exemple présente ce genre de syntaxe).

La **colonne 2** désigne le répertoire local où va être effectué le montage.

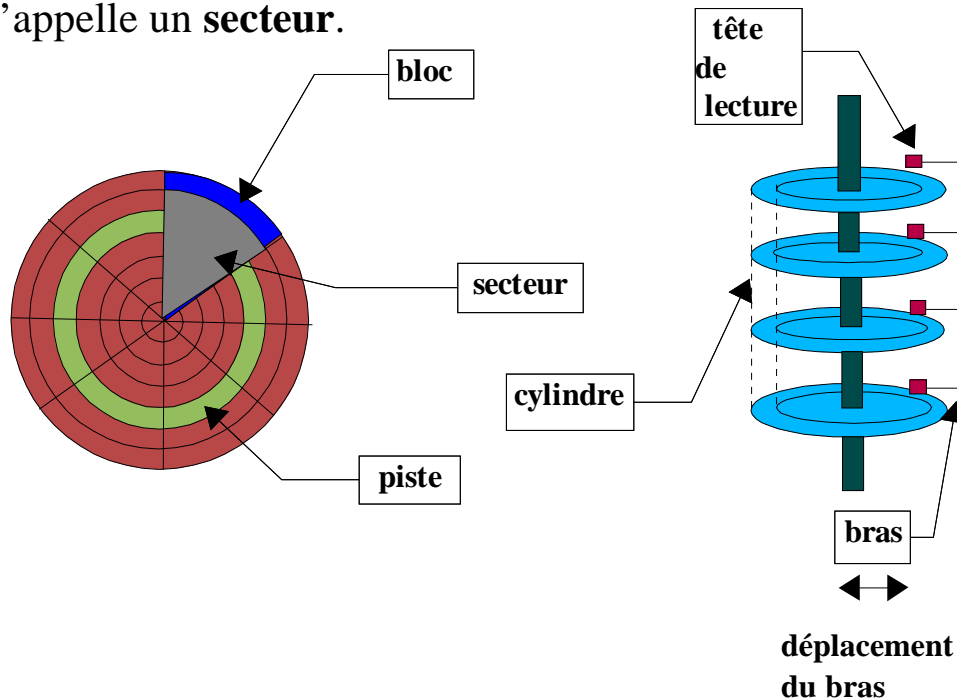
La **colonne 3** indique le système de fichiers à connecter.

La **colonne 4** contient les options de connexion. **"auto"** signifie un montage automatique au démarrage du système.

La **colonne 5** sert à la commande **"dump"**, alors que la **6** sert à la commande **"fsck"** (0 indique que le FS ne doit pas être contrôlé).

1.9 Caractéristiques physiques d'un disque

Un fichier est découpé en blocs logiques auxquels sont attribués des blocs physiques du disque. Un disque est constitué de **plateaux** ; chaque plateau est divisé en **pistes** (cercles concentriques) et en **quartiers** d'angle fixe. Chaque intersection entre une piste et un quartier s'appelle un **secteur**.



Le formatage permet de définir la division des disques en **pistes** et **secteurs**. Généralement l'intersection d'un secteur avec une piste correspond au **bloc**, c'est-à-dire l'unité de transfert entre le périphérique et la mémoire centrale.

L'ensemble des pistes qui peuvent être atteintes sans mouvement du bras forme ce que l'on appelle un **cylindre**.

Il existe plusieurs algorithmes d'ordonnement des requêtes de déplacement sur le disque, afin d'optimiser ces déplacements :

- ordonnancement dans l'ordre d'arrivée,
- ordonnancement suivant le plus court temps de recherche,
- ordonnancement par balayage,
- ordonnancement réduisant le temps de latence.

1.10 Organisation classique (Unix) d'un SGF

L'organisation des fichiers est arborescente, une organisation récente est la suivante :

/etc :	les fichiers de configuration,
/bin :	les binaires nécessaires en mode mono-utilisateur,
/sbin :	les binaires pour l'utilisateur root en mode mono-utilisateur,
/tmp :	le répertoire temporaire en mode mono-utilisateur,
/dev :	le répertoire des fichiers spéciaux,
/var :	le répertoire des fichiers « variables »,
/var/mail :	remplace <i>/usr/spool/mail</i> les fichiers contenant le courrier des utilisateurs,
/var/spool :	remplace <i>/usr/spool</i> ,
/var/tmp :	remplace <i>/usr/tmp</i> ,
/var/acct :	remplace <i>/usr/adm</i> ,
/var/log :	les fichiers de « log »,
/var/crash :	les fichiers image mémoire en cas de crash du système,
/usr/bin :	tous les autres fichiers binaires,
/usr/lib :	les bibliothèques système,
/usr/libdata :	les fichiers de données du système,
/usr/libexec :	les programmes exécutés par d'autres programmes,
/usr/sbin :	les autres binaires pour l'utilisateur root,
/usr/share :	les fichiers indépendants de l'architecture comme les manuels, les sources, les fichiers de définition des terminaux,
/home :	le nom standard pour le répertoire des utilisateurs.