Gestion des fichiers

1 – Généralités

•Définitions

• Organisation logique, organisation physique

2 – Organisation physique

• UNIX : i-list et i-node

• rappels sur le fonctionnement d'un disque

3 – Organisation logique

• L'arborescence UNIX,

• Différents types de fichiers

• Modes et droits d'accès

4 – Organisation logique/physique : illustration par quelques commandes : `cp, mv, rm, ln, ln -s`

5 – Processus et fichiers,

Annexe: la bibliothèque d'entrées-sorties du langage C

Qu'est-ce qu'un fichier ?

• Définition : ensemble d'informations regroupées en vue de leur conservation et de leur utilisation dans un système informatique.

• Type des informations :

- Programmes : fichier binaire exécutable , « script », *byte code*, …,
- Données : fichiers ASCII, binaires, images, son, fichiers d'administration (`/etc/passwd`),…,

• Caractéristiques d'un fichier **:**

- il possède un nom qui permet de **l'identifier** de manière unique (unicité),
- il est rangé sur un support **permanent** (disques),

Sécurité et intégrité :

La sécurité concerne la résistance aux attaques (gestion correcte des droits d'accès aux informations), l'intégrité concerne la résistance aux pannes (duplication des données, par exemple).

Stockage des données et leur représentation :

Le format de représentation des données pose le problème de la portabilité. Par exemple, quel format choisir pour un fichier contenant des données du type "flottants", c'est à dire des réels, si on veut échanger ces données entre deux machines d'architectures différentes (un PC et un MacIntosh)?

Si on range les données sous forme de suite de caractères ASCII, pas de problème, les données seront lues correctement par toutes les machines, mais le volume de stockage pourra être très important, en particulier si les nombres ont une grande dynamique et une précision de plusieurs décimales.
Inversement, le rangement en binaire est moins consommateur de surface disque, mais pose le problème de la compatibilité des représentations internes (IEEE ou non).

Pour les applications qui utilisent plusieurs machines simultanément (client/serveur, par exemple) on adopte des représentations normalisées (format XDR des RPC, CDR de CORBA, ou primitives hton() et ntoh() de la bibliothèque des sockets).

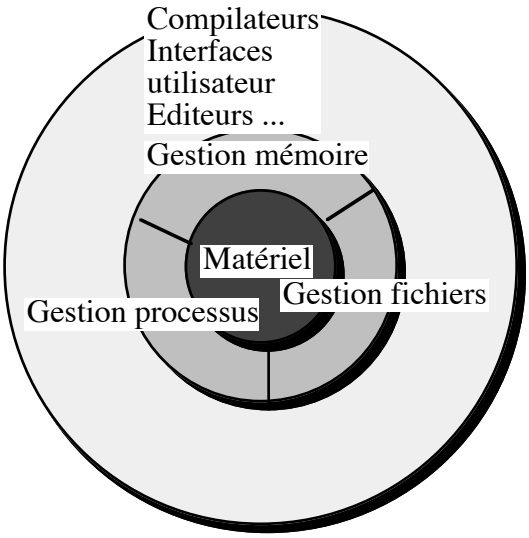
Services assurés par le SGF

- Le *Système de Gestion de Fichiers (SGF)* ou *File System* doit fournir les services suivants :
  - **intégrité** (non altération des informations rangées dans les fichiers),
  - **sécurité** (contrôle d'accès aux fichiers),
  - **permanence** (pérennité des informations : rôle de conservation des informations),
  - **datation** (mémorisation des dates des actions effectuées : mise à jour,…),

- Lorsque les fichiers sont utilisés comme support de **communication** (par transfert ou partage), se pose la question de la **représentation** des informations (données, programmes) :
  - représentation suivant un format « local », propre à une architecture (cf. : exécutable, entiers de 8 à 64 bits, *little endians* et *big endians*…) qui permet seulement l'échange d'informations entre machines ayant la même architecture,
  - représentation adoptant un format normalisé (*ASCII*, *bytecode* java, gif, doc, mp3, XDR, CDR…) qui autorise l'échange d'informations entre différents types d'architectures,

Place du SGF

• Place du SGF (*file system*) dans Unix (qui comporte seulement deux couches `K` et `U`) :



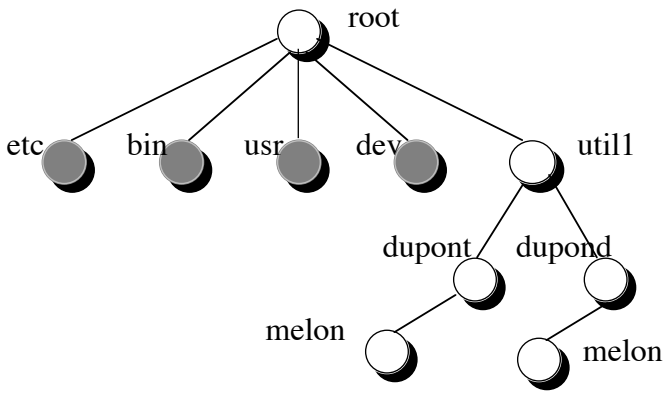
• Le SGF (comme d'autres gestionnaires de ressources) peut être déporté à l'extérieur du noyau, ou ne pas exister (systèmes embarqués)

Le répertoire permet de passer du stockage " à plat" des fichiers sur le disque à l'organisation en arbre vue par les utilisateurs. Les fichiers **répertoires** servent uniquement à l'organisation de l'arbre et aux déplacements dans celui-ci.

Rôle du SGF (aspect système)

- Assurer la correspondance entre l'**organisation logique (hiérarchie)** des fichiers (celle que voit l'utilisateur) et l'**organisation physique (organisation "à plat")** des fichiers (leur implantation physique sur les mémoires de masse)

- Organisation logique : toujours une arborescence. Par exemple, sous UNIX :



- Les fichiers qui ont des successeurs dans l'arbre s'appellent les répertoires (*directories*), un répertoire est donc un fichier qui contient la liste de ses fils.

- Les répertoires sont des annuaires qui **assurent la correspondance et la séparation entre organisation logique et physique,** au même titre qu'un annuaire réseau, qui associe nom de machine et adresse IP

Plan

1 – Généralités

- Définitions
- Organisation logique, organisation physique

2 – Organisation physique

- UNIX : i-list et i-node
- rappels sur le fonctionnement d'un disque

3 – Organisation logique

- L'arborescence UNIX,
- Différents types de fichiers
- Modes et droits d'accès

4 – Organisation logique/physique : illustration par quelques commandes : `cp, mv, rm, ln, ln -s`

5 – Processus et fichiers,

Annexe: la bibliothèque d'entrées-sorties du langage C

La i-list d'un *file system* UNIX, comme ses équivalents sur d'autres systèmes, est dupliquée sur le disque : en effet la destruction d'une partie de la i-list signifie que les fichiers qui y sont décrits sont inaccessibles, mêmes s'ils sont parfaitement intègres sur le disque. De plus, chacun des exemplaires de la i-list est réparti aléatoirement sur le disque pour limiter les pertes en cas de destruction partielle du support physique.

Organisation physique des fichiers

• Sur chaque disque, un **fichier spécial** décrit l'ensemble des fichiers implantés sur ce disque. Citons :

- la MFT (*Master File Table*) de NTFS (*NT File System*) sous Windows,
- la i-list de UFS (*Unix File System*) sous UNIX,
- la FAT (*File Allocation Table*) de MicroSoft DOS,

• Il existe différentes techniques pour gérer l'omplantation des fichiers sur un disque :

- chaînage de blocs à partir de la FAT pour MS-DOS,
- le HFS (*Hierarchical File System*) de Mac OS est basé sur le concept de *b-tree* (*balanced tree*, b-arbre),
- UFS (*Unix File System*) des familles Unix utilise un mélange d'accès direct et de chaînage,
- le NTFS (*NT File System*) de Windows conjugue b-arbre, compression et journalisation,

Organisation physique, exemple : UNIX

• La i-list est une table qui décrit l'ensemble des fichiers implantés sur un disque. Sa taille, qui détermine le nombre de ses entrées, est fixée à l'initialisation du disque. Elle doit être proportionnelle au nombre maximum de fichiers que l'in autorise sur ce disque.

• Chaque entrée de la *i-list* s'appelle un *i-node* et occupe 64 octets.

• Comment calculer T, la taille de la i-list ? On peut choisir :

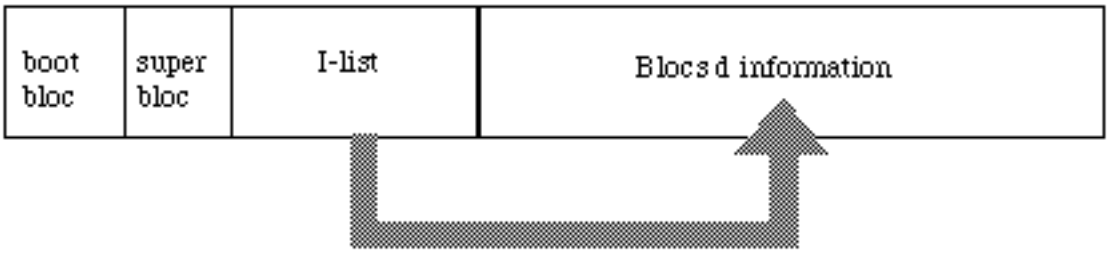
- T grand, pour permettre la création d'un grand nombre de fichiers : une grande surface disque est alors confisquée par la i-list, peut-être de façon inutile,
- T petit, donc gain de place, mais risque de ne pas pouvoir créer un fichier bien qu'il reste de la place sur le disque.

-

On donne ici la structure fonctionnelle du disque. Dans la réalité des implantations, la i-list est dupliquée et dispersée sur le disque, pour des raisons de sécurité.

Un disque peut être divisé en plusieurs *file systems*, c'est à dire être structuré en plusieurs disques logiques.

Organisation générale des `file systems` sous UNIX



- Si ce file system permet le démarrage du système, alors le *boot-bloc* contient les informations de démarrage (*bootstrap*)
- Le *super-bloc* contient, entre autres, les informations suivantes :
  - La taille en blocs de la *i-list,* celle du volume,
  - La liste des blocs libres,
  - Le nom du système de fichiers.

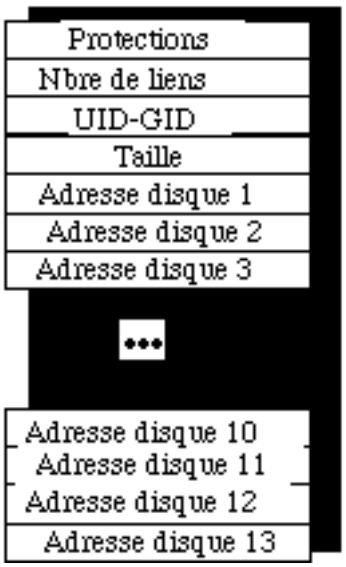
- La commande "`df .`" donne l'occupation de la surface disque :

```
Filesystem      512-blocks     Used      Avail     Capacity
Mounted
/dev/disk0s9    41932936   7673352 33840256     18%     /
```

Le i-node contient TOUTES les informations concernant un fichier : droits d'accès, adresses d'implantation sur le disque, taille, date de création, etc.

La i-list de Unix : Un i-node

• Format d'un i-node :



• Commentaire sur les champs du i-node :

- « Protections » : droits d'accès sur le fichier (cf. `umask` plus loin)
- « `uid`, `gid` » : (*user identifier, group identifier*), paire qui identifie le créateur (cf. `/etc/passwd`)
- « taille » : taille, en octets, du fichier. (Remplacée par les numéros de majeur et de mineur pour les fichiers de `/dev`: périphériques).
- « Adresse disque» : chacune de ces entrées peut contenir un numéro de bloc sur le disque. (**Détails page suivante**),
- « Nombre de liens » : initialisé à 1 lors de la création du fichier, c'est le nombre d'arcs qui arrivent sur le fichier dans l'arborescence,

Les 13 adresses de blocs sur le disque contenues dans un i-node sont séparées en deux catégories :

- 10 adresses de blocs dits « d'information » (sur lesquelles sont rangés les 10 premiers blocs du fichier),
- 3 adresses de blocs dits « d'index » (qui contiennent des adresses de blocs d'information).

Les accès à un fichier qui se font en utilisant les 10 premières adresses sont plus rapides (directs !) et plus sûrs (la perte d'un de ces blocs signifie perte des seules informations qu'il contient) que les accès utilisant les trois dernières adresses.

Les accès qui se font par l'intermédiaire de ces 3 dernières sont moins rapides (une ou plusieurs indirections, ces blocs contiennent des pointeurs) et moins sûrs (la perte d'un de ces blocs signifie perte des informations contenues dans tous les blocs dont il contient les adresses).

Mais cette organisation, qui permet d'avoir une taille fixe pour les i-nodes, convient dans la plupart des cas : un bloc UNIX peut faire jusqu'à 8Ko, ce qui permet des accès directs aux blocs d'informations pour tous les fichiers dont la taille est inférieure à 80 Ko.

Concevoir des applications modulaires organisées en petits fichiers permet de respecter le dicton : *small is beautiful* et la consigne *KISS* (*keep it simple and stupid*), où *stupid* veut dire que le programme inclus dans chaque module doit fait des opérations élémentaires…

Cette inégalité entre les accès aux premiers et derniers blocs d'un fichier est résolue par l'utilisation d'un **cache** par le système de gestion de fichiers d'UNIX.

Remarque :
la taille du bloc était fixée à 512 octets sur les premiers systèmes UNIX, elle est maintenant souvent beaucoup plus grande (4 K octets). Nous avons choisi cette taille de 512 octets, taille "historique", pour faciliter l'exposé.

Du i-node aux blocs alloués sur disque

• Dans chaque i-node on trouve13 adresses de blocs (à l'origine, 1 bloc = 512 octets) :

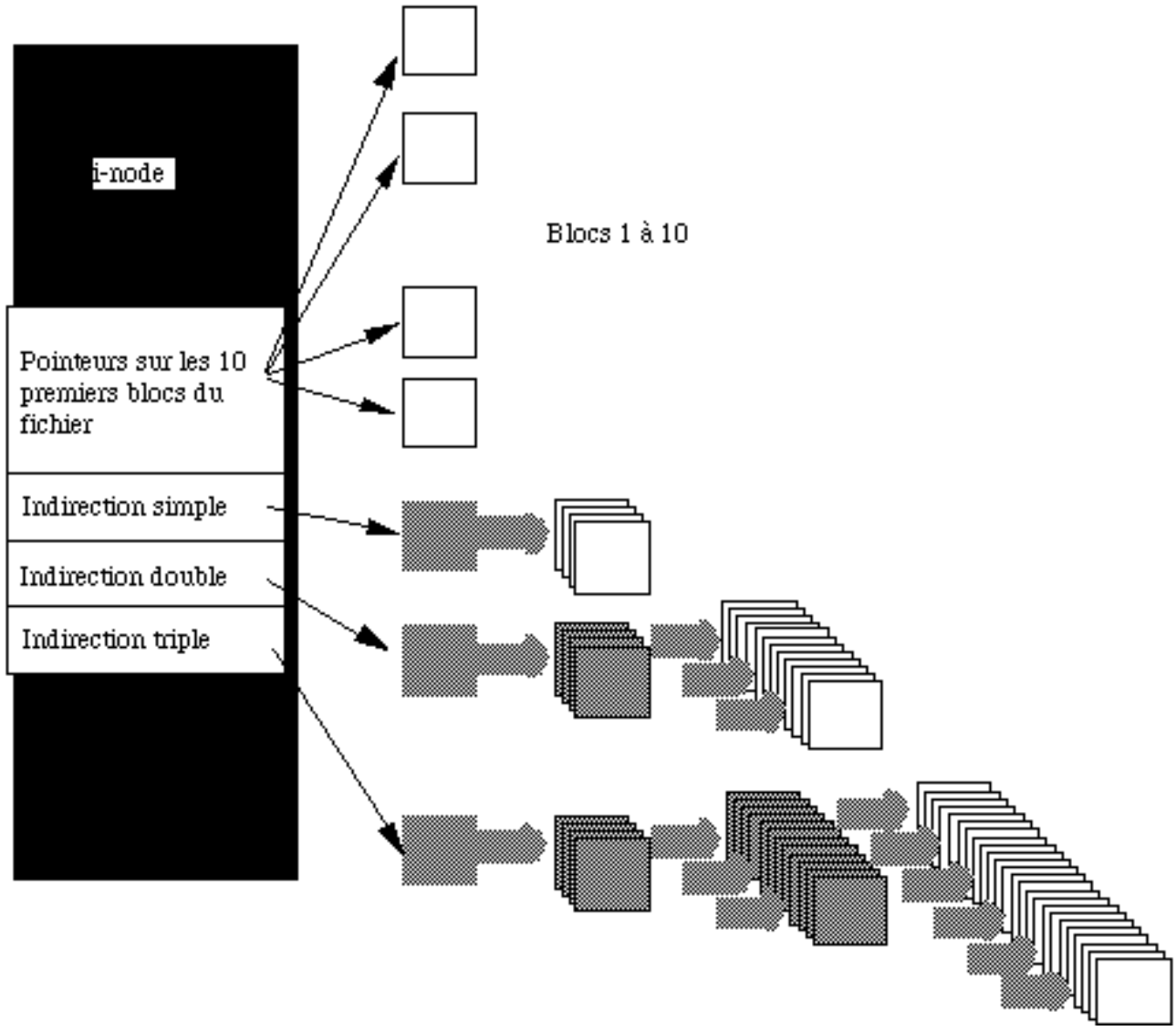
- les 10 premières pointent sur les 10 premiers blocs du fichier (B_0, ..., B_9),
- la 11ème pointe sur un bloc de 128 pointeurs qui adressent les 128 blocs de données suivants (B_{10} à B_{137}),
- la 12ème pointe sur un bloc de 128 pointeurs donnant chacun l'adresse d'un autre bloc de 128 pointeurs vers les blocs de données suivants. Ces blocs sont donc au nombre de 128^2 : 16384 (B_{138} à B_{16522}),
- la 13ème pointe sur un bloc de 128 pointeurs donnant chacun l'adresse d'un bloc de 128 pointeurs vers des blocs de 128 pointeurs vers des blocs de données. Ces blocs sont donc au nombre de 128^3 : 2097152 (B_{16523} à $B_{2113674}$).

Les pointeurs contenant les adresses de blocs sur le disque sont rangés sur 4 octets.
Les 10 pointeurs sur les 10 premiers blocs (accès direct) et les trois suivant (de plus en plus d'indirections) occupent donc (13*4) octets quelle que soit la taille du fichier.
Le jeu des indirections permet d'avoir des i-node de taille fixe, au prix d'une légère perte de place dans le cas des petits fichiers.

Ce style de gestion favorise les petits fichiers : le temps d'accès à une information n'est pas le même suivant qu'ell e se trouve dans les 10 premiers blocs ou dans les suivants.
Ce travers est compensé par l'utilisation d'un *cache disque* en mémoire (cf. chapitre sur la hiérarchie de mémoire).

Du i-node aux blocs alloués sur disque : illustration

• Sur UNIX originel : 1 bloc = 512 octets



Du i-node aux blocs alloués sur disque :

• Exemple avec des blocs de 4 Ko :

Nombre maximum de fichiers que l'on peut ranger sur un disque de 800 Mo (on ne tient pas compte de la place pour ranger le super bloc et la i-list) en supposant que la taille de **chacun** des fichiers est de 600 Ko :

- Pour un fichier, il faut 600/4 (=150) blocs d'information,
- Les 10 premières adresses d'un i-node sont donc utilisées (elles donnent un accès **direct** aux premiers 40 Ko des fichiers),
- La 11ème pointe sur un bloc d'**index** qui contient 4K/4 (= 1K) adresses de blocs de données.
- Il faut donc 10 +1 + 140 (=151) blocs par fichiers. Or y a 800M/4K (=200K) blocs sur le disque, donc on range 200K/151 fichiers, environ 1356.

• Cette organisation est un **compromis** entre accès direct pur (plus rapide, plus sûr, mais qui demanderait des i-nodes de taille variable) et accès indexé,

• Cette organisation **favorise les petits fichiers** : le temps d'accès à une information est différent selon que cette information est en fin ou en début de fichier

• Le **cache** rend **équitable** (en terme de latence) l'accès aux informations

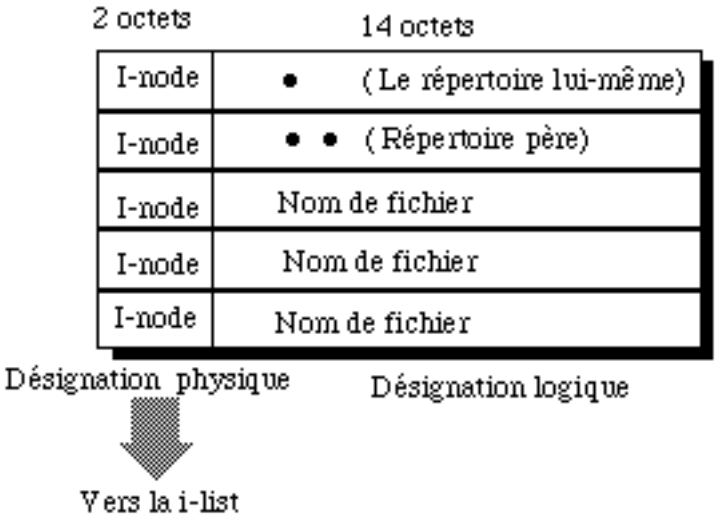
On remarquera que le répertoire UNIX reflète parfaitement la séparation organisation logique / organisation physique.
La première organisation concerne l'utilisateur et la seconde le système.
Chacune des entrées du répertoire comporte deux champs : l'un contient le i-node qui décrit complètement le fichier, l'autre contient le nom du fichier, information fondamentale pour l'utilisateur mais qui intéresse peu UNIX.

Le répertoire est un annuaire qui assure la correspondance entre adresse physique et nom logique de l'information. On pourra faire le rapprochement avec le fichier /etc/hosts qui associe nom de machine et adresse IP, la table de pages qui associe adresse virtuelle et adresse physique, etc.

L'implantation historique décrite ci-contre a été améliorée. Dans les versions actuelles d'UNIX, la taille du champ « nom de fichier » est variable, ce qui élimine la contrainte sur la taille des noms de fichiers.

Répertoire Unix

• Structure "historique" du répertoire UNIX :



• le répertoire fait correspondre organisation logique et physique qui sont **complètement** séparées

• exemples de commandes concernant les répertoires :

- `cd` (simple déplacement dans l'arbre)
- `ls` (simple affichage du contenu du répertoire courant)
- `ls-l` (pour chaque fichier du répertoire courant, on ouvre le i-node correspondant, le nombre d'accès disque est **beaucoup** plus grand que dans le cas de `ls`)

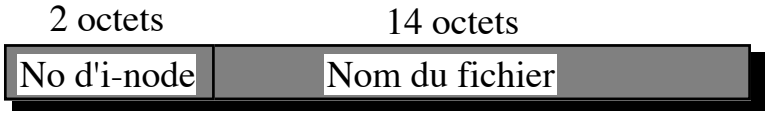
L'exemple ci-contre montre comment le SGF navigue dans les répertoires pour retrouver l'information cherchée.

On suppose que tous les droits d'accès sont suffisants.

La commande donnée peut échouer si celui qui l'a émise n'a pas le droit de traverser un répertoire menant vers le fichier, s'il n'a pas les droits de lecture sur le fichier, ou s'il n'a pas le droit d'utiliser la commande, ici cat...

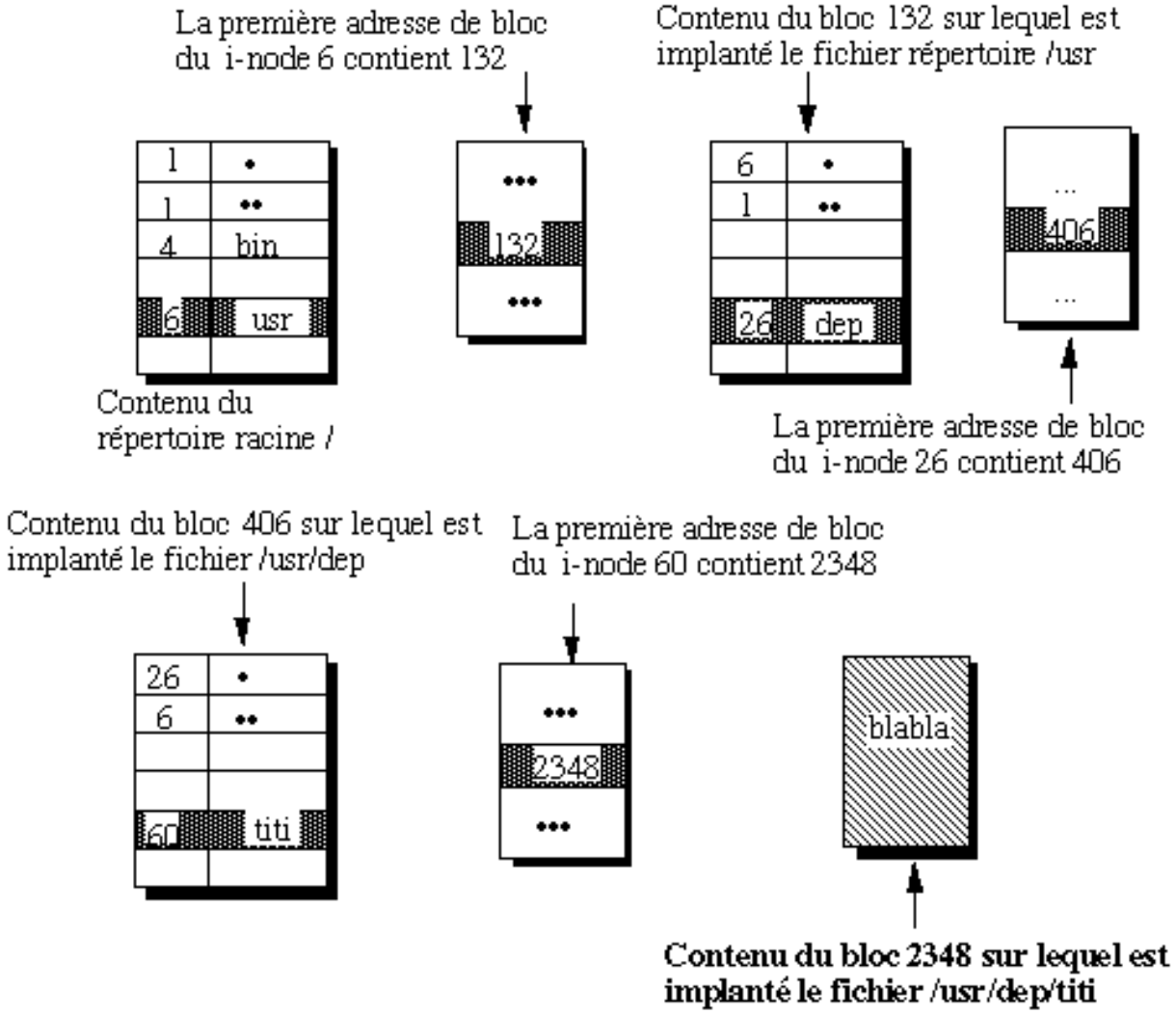
Exemple d'accès à un fichier

• Rappel : format d'une entrée d'un répertoire :



Entrée d'un répertoire UNIX

• Effet de la commande `cat /usr/dep/titi` :



Un *disque* est divisé en *pistes* (tracks) ; 75 à 500 pistes par surface de disque.
Une *piste* est divisée en *secteurs* ; 32 à 4096 octets par secteur, 4 à 32 secteurs par piste.

Les *secteurs* sont l'unité de base. Le SGF manipule des blocs qui correspondent aux secteurs.

Un disque peut comprendre plusieurs *plateaux* chacun ayant deux surfaces.

Un *cylindre* est un ensemble des pistes qui ont la même position sur les disques.
Les accès se font en précisant un numéro de piste (ou de cylindre), un numéro de surface, un numéro de secteur.

Soit un disque qui possède les caractéristiques suivantes :

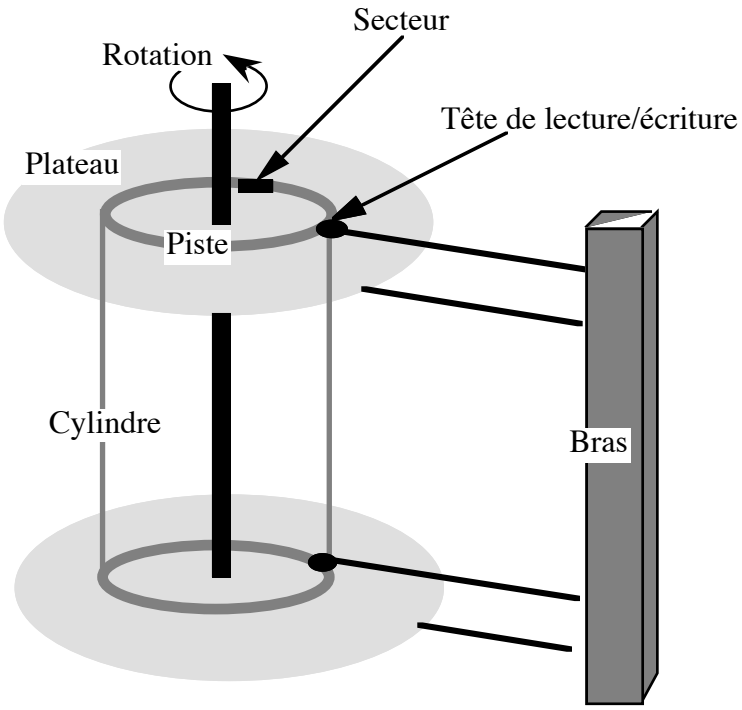
- longueur d'une piste : 32768 octets,
- temps d'une rotation : 16.67 ms,
- temps moyen de déplacement du bras : 30 ms.

Le temps moyen de lecture d'un bloc quelconque de k octets est :

30 + 8,3 + (k / 32768) x 16,67 ms

Rappel : fonctionnement d'un disque

- Les têtes de lecture/écriture sont déplacées jusqu'à la piste, positionnées sur la surface, puis attendent que le secteur voulu arrive jusqu'à elles par rotation :



- Remarque : les blocs dits "contigus" sont ceux qui ont les mêmes adresses de secteur sur des pistes différentes (lecture simultanée);

L'unité d'échange d'un disque se comporte comme le fait le système : elle doit ordonnancer les accès aux pistes, comme le système ordonnance l'accès au processeur.

Les algorithmes utilisés pour gérer les disques présentent les mêmes propriétés que les algorithmes d'ordonnancement.

Dans l'exemple ci-contre, l'unité d'échange a mémorisé cet ensemble de pistes à lire : 110, 190, 30, 120, 10, 122, 60, 65.

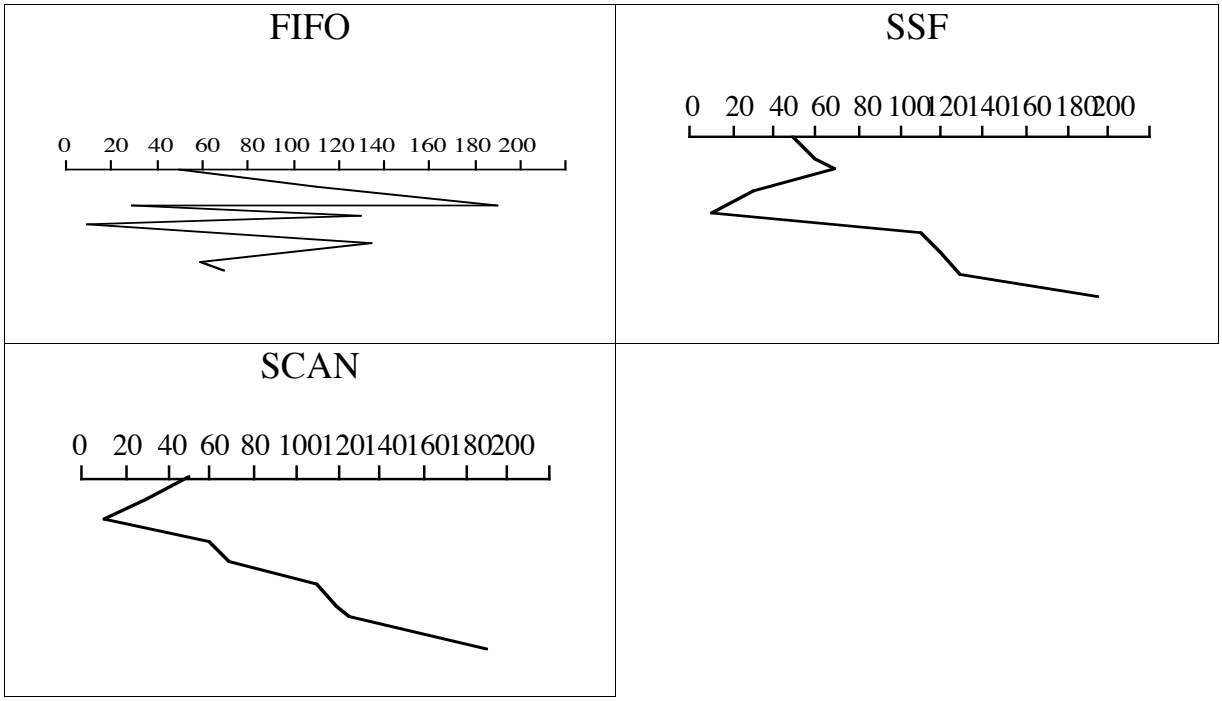
Si on utilise SCAN à partir de 50, on classe les pistes à lire en deux suites :
l'une décroissante à partir de 50 vers la piste 0 : (30,10),
l'autre croissante vers la fin du disque : (60, 65, 110, 120, 122, 190).

Il peut y avoir famine si la liste est modifiée dynamiquement et si de nombreuses demandes d'accès sont faites vers des pistes voisines de la piste courante et quelques unes vers des pistes "lointaines".

Gestion des déplacements de la tête

- Algorithmes disponibles (à rapprocher de ceux qui décident de l'ordonnancement des processus) :
  - FIFO (*First In First Out*), accès dans l'ordre des demandes,
  - SSF (*Shortest Seek First*), accès à la piste la plus proche, optimal, mais possibilité de famine,
  - SCAN (ou ascenceur, ou chasse neige)le plus utilisé en pratique.
- Exemple : la tête est sur la piste 50, l'unité d'échange a mémorisé les demandes d'accès sur les pistes suivantes :

(110, 190, 30, 120, 10, 122, 60, 65)



Plan

1 – Généralités

- Définitions
- Organisation logique, organisation physique

2 – Organisation physique

- UNIX : i-list et i-node
- rappels sur le fonctionnement d'un disque

3 – Organisation logique

- L'arborescence UNIX,
- Différents types de fichiers
- Modes et droits d'accès

4 – Organisation logique/physique : illustration par quelques commandes : `cp, mv, rm, ln, ln -s`

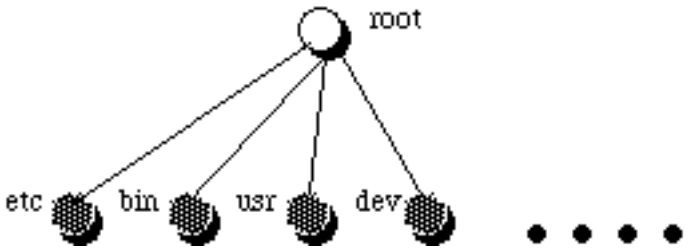
5 – Processus et fichiers,

Annexe: la bibliothèque d'entrées-sorties du langage C

Organisation logique : l'arborescence

• On prendra celle d'UNIX comme exemple d'organisation logique.

• Commentaires sur quelques répertoires d'administration :



• `/dev` pour les fichiers **spéciaux** (périphériques) : les périphériques sont BANALISES et vus comme des fichiers

• `/etc` pour les fichiers d'administration comme `passwd, group, hosts`,...

•`/bin et /usr/bin` pour les commandes du shell, ...,

•`/usr/include` pour les fichiers `.h`

•`/var/spool/mail` pour la messagerie

• Un répertoire bien pratique :

- `/tmp` dans lequel tout le monde peut lire et écrire

Les différents types de fichiers

• Plusieurs catégories :

- fichiers ordinaires (exécutables, données, …) : ce sont des sommets pendants dans l'arbre),
- fichiers répertoires (les nœuds qui ont un successeur dans l'arbre),
- fichiers spéciaux (sommets pendants dans l'arbre) : périphériques, liens symboliques, tubes, ...

• Exemple. On obtient le type d'un fichier grâce à la commande `ls–l` :

```
$ls –l
drwxr-xr-x 56 dupont        3072 May 14 09:04 ..
-rw-r-----  1 dupont           5 May 14 09:05 fic1
lrwxrwxrwx  1 dupont           4 May 14 09:05 fic2 ->
fic1
```

• les fichiers de `/dev` sont en mode bloc (`b`) ou caractère (`c`) :

```
crw-------  1 dupont   28, 128 Feb  3  1995 audio
brw-rw-rw-  1 root     36,   0 Feb  2  1995 fd0
crw-rw-rw-  1 root     11,   0 Jul  4  1995 tcp
```

• En mode bloc, on utilise le cache, pas en mode caractère.

Fonction lseek :

```
lseek(int file,int Offset,int From);
```

`From` : début, fin ou position courante

`Offset` : nombre d'octets à partir de Depuis

Redirection des entrées/sorties :

- `ls > fichier`
- `ls > /dev/ptty2`
- `cat bark.au > /dev/audio`
- `ls -lR > /dev/audio`
- `mail dupont -s "fichier.c" < fichier.c`

Structure des fichiers et modes d'accès sous UNIX

• Contrairement à d'autres systèmes, Unix ne propose pas de format de fichier : un fichier Unix est une simple suite d'octets.

- avantages :
  - portabilité et faible encombrement du SGF,
  - **banalisation** : les périphériques étant traités comme des fichiers, ceci permet la **redirection** des entrées/sorties, exemples :
    - `ls > fichier`
    - `ls > /dev/ptty2`
    - `cat bark.au > /dev/audio`
    - `mail le_prof -s "TP 1" < fichier.c`
- inconvénients : nombreuses fonctions à réaliser par l'utilisateur (exemple : développer une base de données !).

• Accès aux fichiers :

- l'accès par défaut est **séquentiel** : une fonction d'accès (`read`, `write`, etc) à un fichier reprend le curseur dans le fichier là où l'avait laissé le traitement précédent.
- Pour faire un accès direct, il faut utiliser la fonction `lseek`

Pour lire un fichier, il faut avoir le droit de lecture (`r`) sur ce fichier et le droit de traverser (`x`) le, ou les, répertoires qui y conduisent.

Exemple :

• Pour créer un répertoire privé, c'est à dire qui peut être lu seulement si on en donne le nom :

```
$mkdir   private
$chmod   711     private
$cd      private
$mkdir   dir1
$chmod   755     dir1
```

• Personne ne peut lire dans `private`, (par exemple faire : `ls private`) c'est à dire voir qu'il contient `dir1`, mais on peut accéder aux informations contenues dans `dir1 (cd dir1)`.

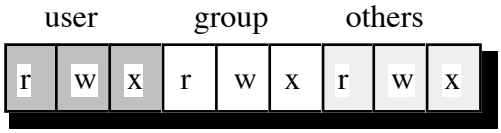
• En effet :
– pour faire `cd` il faut `x` sur le répertoire visé
– pour faire `ls` il faut `r` sur le répertoire visé

Les droits d'accès aux fichiers

• A la création du fichier, le SGF :

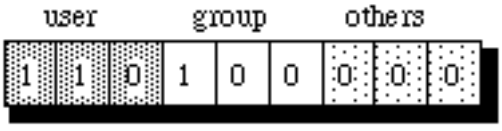
- copie les `uid` et `gid` (trouvés dans `/etc/passwd`) du créateur dans le i-node,
- initialise les droits d'accès, aussi dans le i-node, en utilisant le `umask` (trouvé dans le fichier qui définit l'environnement),

• Ces droits d'accès sont codés sur 10 bits :



• Exemple : droits d'accès pour le fichier fic.c :

```
-rw-r-----  1 dupont      8229 Mar 20 10:08 fic.c
```



• `umask` renvoie (ou initialise) les accès interdits par défaut sur les fichiers créés. Par exemple, un `umask 022` signifie écriture interdite pour group et others.

La commande `chmod` permet de positionner le bit `s` :

Commandes	Résultats affichés à l'écran
`ls -l ma_commande`	
	`-rwxr--r-- 1 dupont ··· ma_commande`
`chmod u+s ma_commande`	
`ls -l ma_commande`	
	`-rwsr--r-- 1 dupont ··· ma_commande`

Lorsqu'un utilisateur quelconque exécutera le fichier `ma_commande` il prendra l'identité "`dupont`" et aura donc accès à tous les fichiers appartenant à `dupont` comme s'il était `dupont`.

Dans `/etc/passwd`, un utilisateur peut modifier son mot de passe (et son shell) par défaut, pourtant, il n'a pas l'accès en écriture sur ce fichier, qui appartient à `root` ! Mais il utilise la commande `passwd` qui appartient à `root`, sur laquelle le bit s est positionné à 1, il prend ainsi l'identité "`root`" et peut écrire sur les fichiers appartenant à `root`, en particulier il peut modifier `/etc/passwd.`

Droits d'accès : le bit s

• Si le bit `s` est positionné sur un fichier :

l'utilisateur U du fichier prend l'identité du PROPRIETAIRE P du fichier :

- on parle de `real uid` et `gid` pour ceux de U,

- on parle de `effective uid` et `gid` pour ceux de P,

• Exemple, les droits sur `/bin/passwd` sont positionnés comme suit par `root` pour que les utilisateurs puissent accéder à `/etc/passwd` :

```
-r-sr-sr-x  1 root     7728 Jul 16  1994 /bin/passwd
-rw-r--r--  1 root      559 Jul 25  1995 /etc/passwd
```

ici le bit s est positionné sur l'exécutable `/bin/passwd`, celui qui utilise cette commande prend donc l'identité de `root` pendant cette utilisation. Il pourra donc mettre à jour le fichier `/etc/passwd`, bien qu'il n'ait pas le droit d'y écrire...

Plan

1 – Généralités

- Définitions
- Organisation logique, organisation physique

2 – Organisation physique

- UNIX : i-list et i-node
- rappels sur le fonctionnement d'un disque

3 – Organisation logique

- L'arborescence UNIX,
- Différents types de fichiers
- Modes et droits d'accès

4 – Organisation logique/physique : illustration par quelques commandes : `cp, mv, rm, ln, ln -s`

5 – Processus et fichiers,

Annexe: la bibliothèque d'entrées-sorties du langage C

Commande **cp**

Cette commande copie un fichier dans un autre, le SGF enchaîne les actions suivantes :

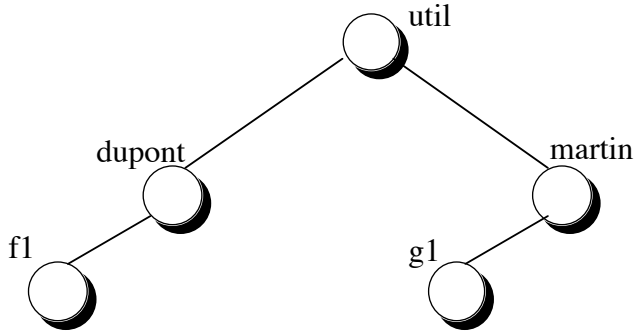
– chercher un i-node libre dans la i-liste,
– s'il en existe un, chercher de la place libre sur le disque,
– si il y en a assez, copier les blocs, mettre à jour le i-node trouvé,

Un fichier peut ne pas être créé pour plusieurs raisons :

– pas le droit d'écrire dans le répertoire visé, de lire dans le rép. source,
– pas le droit de lire le fichier source,
– pas de i-node libre, bien qu'il y ait de la place sur disque,
– i-node libre, plus de place sur disque,

Commande `cp`

- `dupont` fait : `cp f1 f2` dans son *home directory*
- Etats de l'arborescence et du répertoire `dupont` **avant** l'exécution de cette commande:



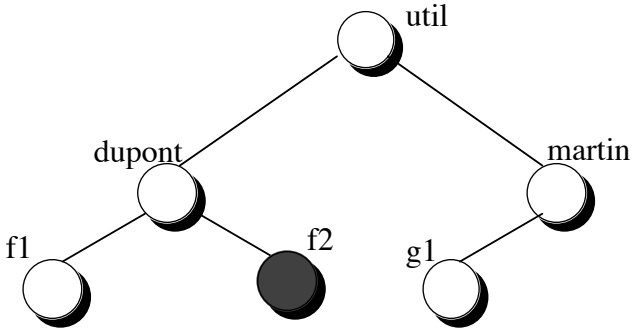
Catalogue dupont

29	.
12	..
50	f1

Catalogue martin

38	.
12	..
47	g1

- Etats de l'arborescence et du répertoire `dupont` **après** l'exécution de cette commande:



Catalogue dupont

29	.
12	..
50	f1
62	f2

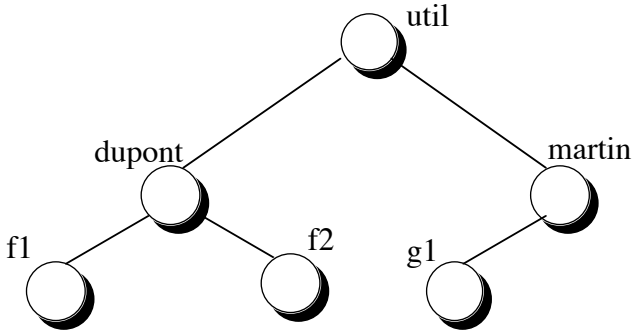
Catalogue martin

38	.
12	..
47	g1

Note : Si on passe un fichier d'un *file system* à un autre par `mv` , il y copie puis destruction (équivalent de `cp` suivi de `rm`).

Commande `mv`

- `dupont` fait maintenant : `mv f2 f50`
- Etats de l'arborescence et du répertoire `dupont` **avant** l'exécution de cette commande:



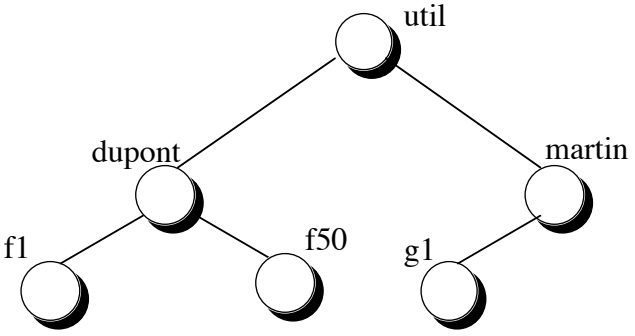
Catalogue dupont

29	.
12	..
50	f1
62	f2

Catalogue martin

38	.
12	..
47	g1

- Etats de l'arborescence et du répertoire `dupont` **après** l'exécution de cette commande:



Catalogue dupont

29	.
12	..
50	f1
62	f50

Catalogue martin

38	.
12	..
47	g1

- Très peu de travail sur le disque pour le SGF : seul le nom du fichier dans le répertoire change, pas de gestion d'espace disque.

Commande ***ln*** (lien)

Appels système :
- `link (Origine, Destination);`

Pas de `ln` entre file system différents

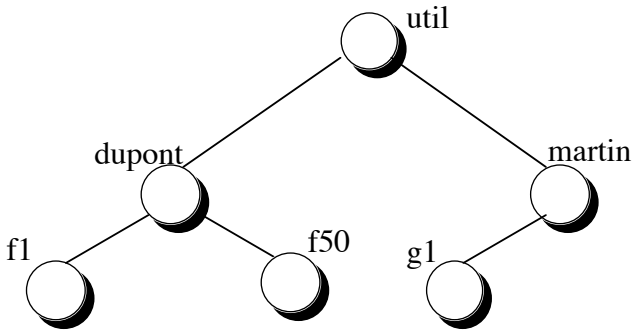
Commande `ln`

• Cette commande s'appelle *hard link* dans le jargon UNIX :

- Le fichier n'est **pas** dupliqué
- `ln` augmente de 1 le nombre de liens sur le fichier

• Exemple, `martin` fait : `ln ../dupont/f50 g2`

• Etats de l'arborescence et du répertoire `dupont` **avant** l'exécution de cette commande :



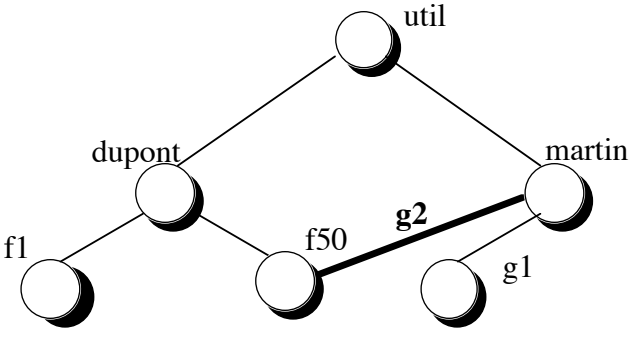
Catalogue dupont

29	.
12	..
50	f1
62	f50

Catalogue martin

38	.
12	..
47	g1

• Etats de l'arborescence et du répertoire `dupont` **après** l'exécution de cette commande :



Catalogue dupont

29	.
12	..
50	f1
62	f50

Catalogue martin

38	.
12	..
47	g1
62	g2

Commande `ln -s`

• Cette commande (*symbolic link*) **crée** un fichier, donc alloue un i-node qui contient la chaîne de caractère entrée et marque ce fichier comme étant de type `l`,

• Exemple, la commande :

```
ln -s ../dupont/f50  g3
```

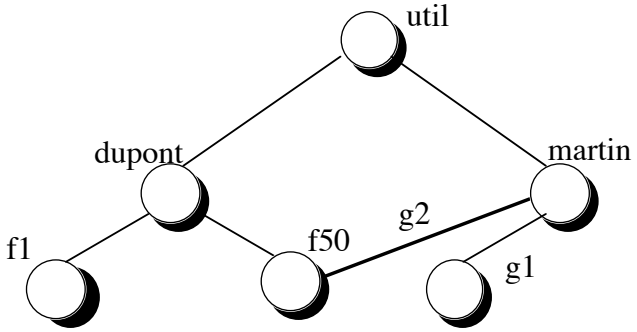
va donc créer dans `dupont` un fichier `g3` dont le contenu est `../dupont/f50` et dont le type est `l`.

• un lien symbolique est donc un **pointeur,**

Commande `ln -s`

• `martin` fait : `ln -s ../dupont/f50 g3`

• Etats de l'arborescence et du répertoire **avant** l'exécution :



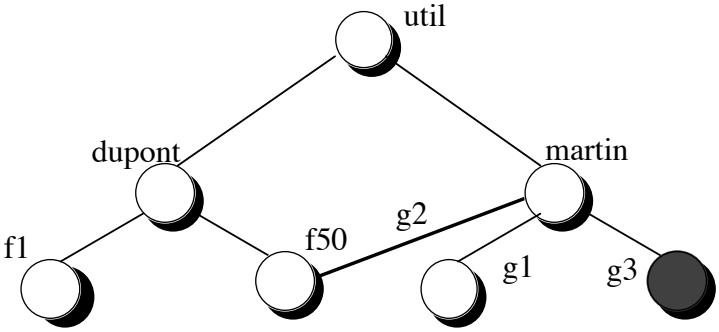
Catalogue dupont

29	.
12	..
50	f1
62	f50

Catalogue martin

38	.
12	..
47	g1
62	g2

• Etats de l'arborescence et du répertoire **après** l'exécution :



Catalogue dupont

29	.
12	..
50	f1
62	f50

Catalogue martin

38	.
12	..
47	g1
62	g2
101	g3

• Contenu du fichier g3 : `../dupont/f50`

• **attention** si dupont fait : `mv f50 f60,`

alors `cat g3` donnera `file not found` !

Commande `ln` *et* `ln –s` *(différence)*

• Depuis le répertoire courant, faire :

Commande	Commentaire
`mkdir dir1 dir2`	créer deux répertoires
`cd dir1`	aller dans le répertoire `dir1`
`echo abc > fic1`	créer `fic1`
`cd ../dir2`	aller dans `dir2`
`ln ../dir1/fic1 fic2`	`fic2` est un **synonyme** de `fic1`
`ln -s /dir1/fic1 fic3`	`fic3` est un **pointeur** vers `fic1`

• Contenu du répertoire `dir2`, obtenu par `ls –il` :

```
23117  -rw-r--r--   2 dupont    fic2
28865 lrwxrwxrwx    1 dupont    fic3 -> ../dir1/fic1
```

• Contenu du répertoire `dir1`, obtenu par `ls –il` :

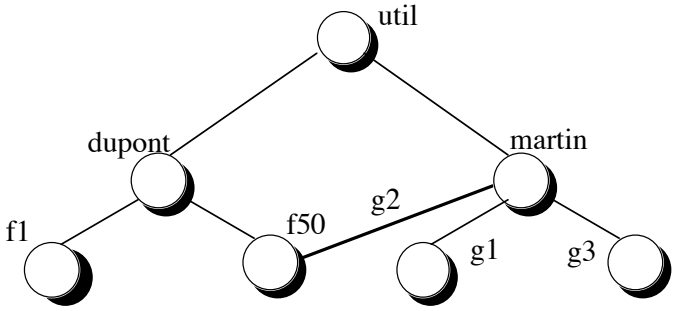
```
23117 -rw-r--r--  2 dupont      fic1
```

- Les fichiers `fic2` dans `dir2` et `fic1` dans `dir1` ont le **même** numéro de i-node. **: 23117,**
- `fic3` est de **type `l`,** c'est à dire un lien symbolique, un pointeur, c'est un autre fichier : il a un numéro de i-node différent : **28865**.

C'est ainsi que l'on peut récupérer les fichiers détruits. En fait, ils sont détruits dans l'organisation logique, mais pas forcément dans l'organisation physique.

Commande `rm`

• Situation initiale :



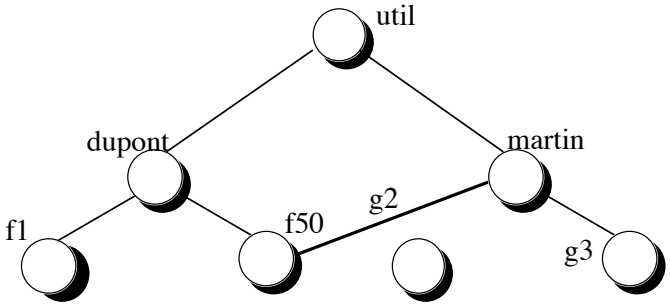
Catalogue dupont

29	.
12	..
50	f1
62	f50

Catalogue martin

38	.
12	..
47	g1
62	g2
101	g3

• Effet de `rm /util/martin/g1`:



Catalogue dupont

29	.
12	..
50	f1
62	f50

Catalogue martin

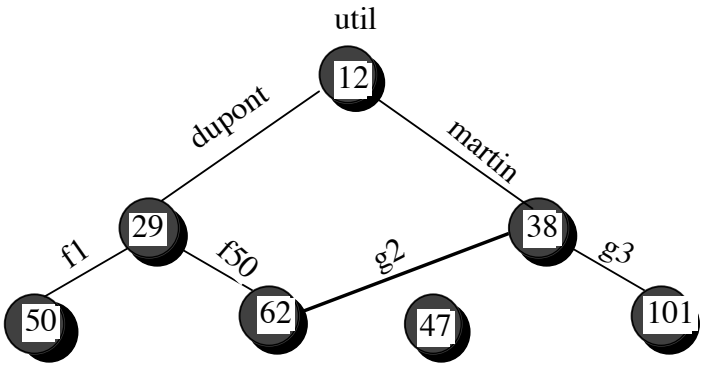
38	.
12	..
0	g1
62	g2
101	g3

• La commande `rm` (`remove`) porte bien son nom (contrairement à de nombreuses commandes Unix), elle ne veut pas dire `delete` : seul l'arc conduisant au fichier est détruit : **nbre de liens =nbre de liens - 1**

• Mais, il peut y avoir plusieurs arcs (links) conduisant à ce fichier ! Le fichier n'est détruit, c'est à dire le i-node considére comme **libre**, seulement quand le dernier lien sur le fichier est détruit ;

Nom de fichier et i-node : récapitulatif

- l'utilisateur s'intéresse aux **noms des fichiers, les PATHNAMES** ou nom des arcs, Unix **gère les i-nodes, noms des nœuds** :



Catalogue dupont

29	.
12	..
50	f1
62	f50

Catalogue martin

38	.
12	..
0	g1
62	g2
101	g3

- Le répertoire assure la **correspondance** pathname/i-node ; sous Unix il peut y avoir plusieurs chemins (pathnames) pour accèder au même fichier (i–node)
- Le répertoire assure l'**indépendance** des organisation logiques et physiques : Unix gère des i-nodes, les véritables fichiers, l'utilsateur construit l'arbre des **pathnames**
- Un fichier n'est détruit que si le dernier arc (`link`) qui y conduit disparaît

Plan

1 – Généralités

- Définitions
- Organisation logique, organisation physique

2 – Organisation physique

- UNIX : i-list et i-node
- rappels sur le fonctionnement d'un disque

3 – Organisation logique

- L'arborescence UNIX,
- Différents types de fichiers
- Modes et droits d'accès

4 – Organisation logique/physique : illustration par quelques commandes : `cp, mv, rm, ln, ln -s`

5 – Processus et fichiers,

Annexe: la bibliothèque d'entrées-sorties du langage C

Processus et fichiers : Table des fichiers

• Descripteur/ nom de fichier :

- Pour accéder à un fichier, un processus ne le désigne pas directement par son nom, mais il passe par un intermédiaire, appelé descripteur, qui peut être un simple numéro,
- Cette association se fait en utilisant les fonctions du type « open ».

• Table des fichiers ouverts (*User Open File Table, UOFT*)

- Sous UNIX, le mécanisme repose sur l'utilisation d'un table appelée *User Open File Table*. Elle contient une entrée par fichier ouvert **pour** le processus. (Pas forcément ouvert **par** lui). Il y a une telle table par processus.
- Chaque entrée de cette UOFT contient un pointeur sur une table appelée *System Open File Table*, commune à tous les processus. Chaque **ouverture** de fichier ajoute une entrée dans cette table. Les entrées de la SOFT contiennent le nom du fichier et la valeur courante du pointeur dans celui-ci.

Processus et fichiers : Premier exemple

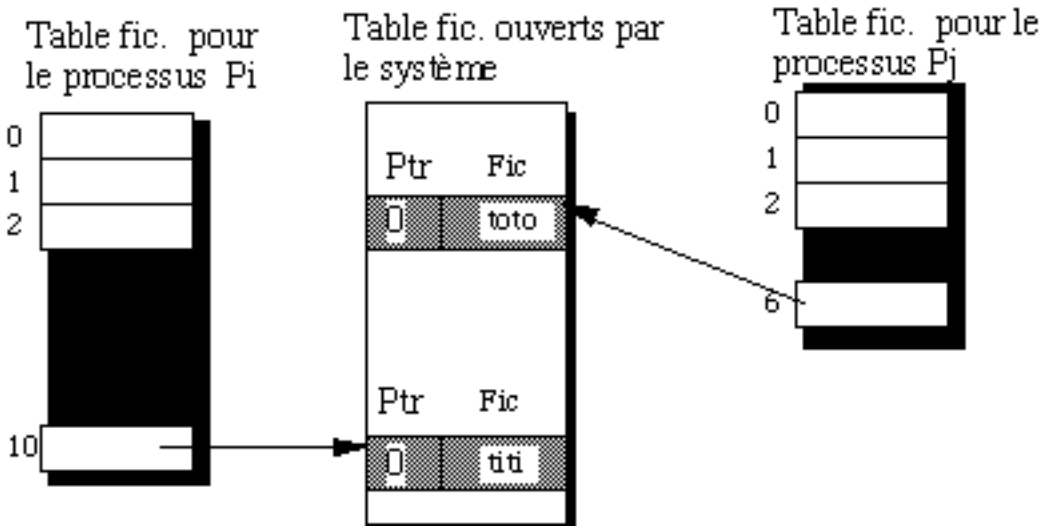
• Dans l'exemple suivant, on suppose que d'autres appels à `open` ont précédé ceux faits par Pi sur le fichier `titi` et Pj sur le fichier `toto`:

Programme exécuté par Pi	Programme exécuté par Pj

`int main(void){`
`int Ret;`
`...`
`Ret= open ("titi", O_RDONLY);`
`printf ("retour de open = %d\n", Ret);`
`...`
	`int main(void){`
	`int Ret;`
	`...`
	`Ret= open ("toto", O_RDONLY);`
	`printf ("retour de open = %d\n", Ret);`
	`...`

Résultat : `retour de open = 10`	Résultat : `retour de open = 6`

• Effet de ces deux appels à `open` sur les tables :



• Une entrée de la SOFT contient deux champs :

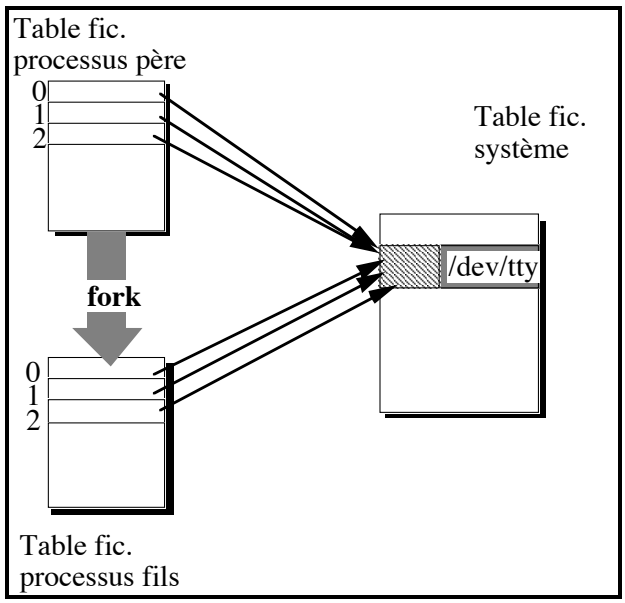
- `Ptr`: la valeur du curseur dans la fichier
- `Fic`: le nom du fichier (*pathname* complet)

On obtient le nom du terminal courant en utilisant la commande `tty`.

Processus et fichiers : Création de la table `UOFT`

• Initialisation de la table des fichiers ouverts pour un processus :

- un processus hérite (par duplication) de la table des fichiers ouverts de son père,
- Ainsi, lors de la création d'un processus à partir du shell, le système lui associe une copie de la *user open file table* du shell dans laquelle les trois premières entrées : 0 (`stdin`), 1 (`stdout`), et 2 (`sdterr`) pointent vers le terminal, ou pseudo terminal, courant :



• On note que ces trois entrées contiennent un pointeur vers la même entrée de la *system open file table* qui donne le nom de la ressource.

Redirection des Entrées/Sorties (1)

• La redirection des E/S montre l'intérêt de passer par un descripteur logique plutôt que de donner directement le nom de l'objet sur lequel on travaille,

• Rappelons que l'on peut rediriger les E/S, en particulier en utilisant les symboles > et < :

- > redirige 1, la sortie standard,
- < redirige 0, l'entrée standard

• Exemple de redirection :

- `ls –l > titi` demande au shell de créer un processus dont les sorties ne se feront pas sur le terminal courant mais dans le fichier `titi` ,
- le schéma de la page suivante indique comment se fait une redirection dans la UOFT

Redirection des Entrées/Sorties (2)

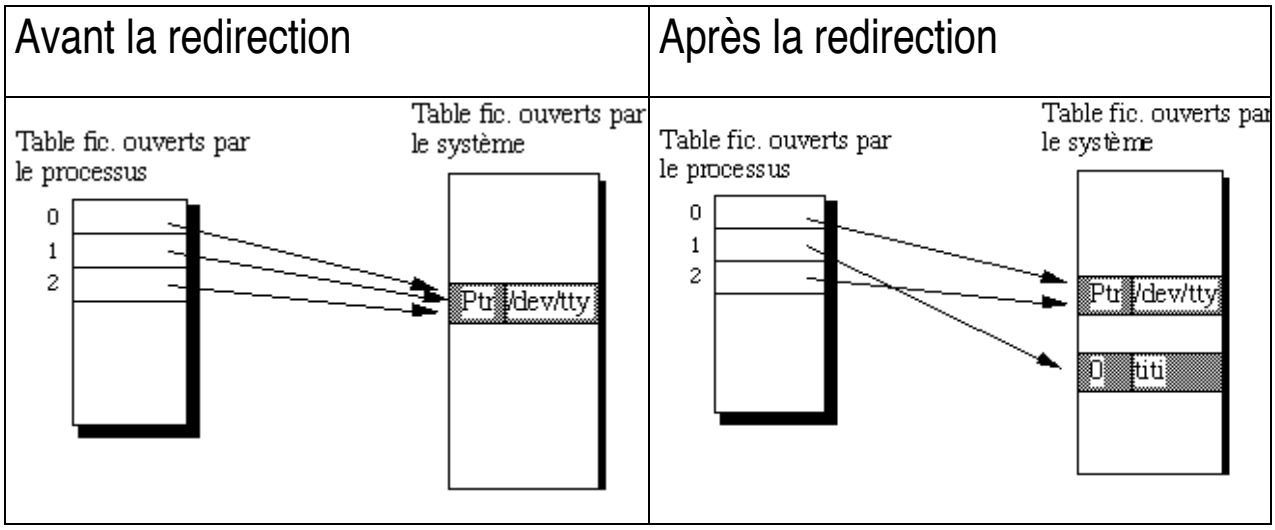
• Effet de la commande `ls –l > titi` au niveau des tables:

1- création et ouverture d'un fichier `titi,`

2- attribution à ce fichier d'une entrée dans la table commune (SOFT)

3- modification du contenu de l'entrée 1 de la table propre au processus (UOFT) qui pointe maintenant vers l'entrée attribuée à `titi` dans la table commune

• Etat de la table *UOFT* avant et après la redirection :



Par défaut 0, 1 et 2 sont associés au terminal courant (/dev/tty).
Chaque open sur un fichier renvoie le numéro de l'entrée qui a été associée à ce fichier dans la table des fichiers ouverts par le processus.
L'allocation se fait ainsi : la table est scrutée à partir de l'entrée zéro, et **la première entrée libre** est associée au fichier à ouvrir.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main(void) {
   int Ret_Op, i;

   for (i=0; i<3; i++){
      Ret_Op = open ("/etc/hosts", O_RDONLY);
      printf ("retour de open = %d\n", Ret_Op);
   }
   close (0);

   for (i=0; i<3; i++){
      Ret_Op = open ("/etc/hosts", O_RDONLY);
      printf ("retour de open = %d\n", Ret_Op);
   }
return 1;
}
```

Résultat de l'exécution du programme précédent :

```
retour de open = 3
retour de open = 4
retour de open = 5
retour de open = 0
retour de open = 6
retour de open = 7
```

Commentaire:
L'appel à `close` a libéré l'entré d'index 3, qui devient ainsi la première entrée libre.

Exemple d'exécution de la commande `limit` :

```
cputime         unlimited
filesize        unlimited
datasize        6MB
stacksize       8MB
coredumpsize    0kB
memoryuse       unlimited
memorylocked    unlimited
maxproc         100
descriptors   256     (256 fichiers maximum ouverts simultanémént)
```

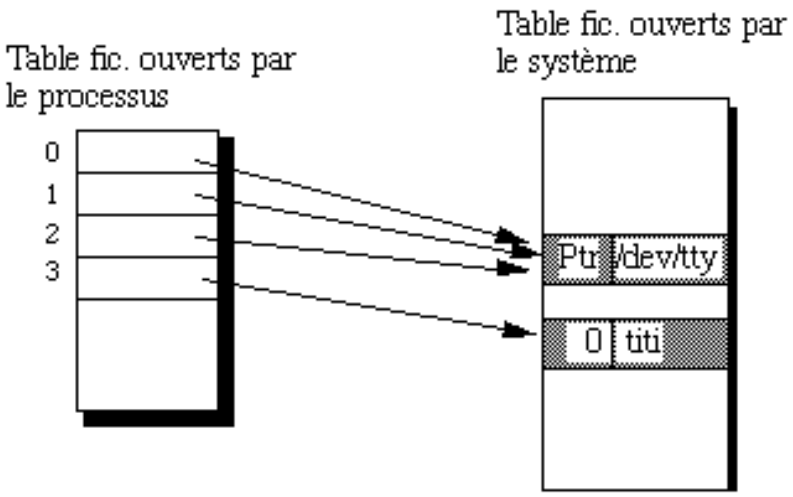
Processus et fichiers : Gestion de la table

• Comment sont attribuées les entrées de la *user open file table :*

- Lors de l'ouverture d'un fichier, le système lui associe la première entrée **libre** dans la *user open file table :*

programme :	résulat:

`int Ret_Op;`
`Ret_Op = open("titi", O_RDONLY);`
`printf("retour de open = %d\n", Ret_Op);`
	`retour de open = 3`



• Remarques*:*

- dans le cas d'un environnement multi fenêtres, à la création du processus, la première entrée libre peut être largement supérieure à 3,

- le nombre maximum de fichiers ouverts simultanément est un paramètre dont on peut connaître la valeur en utilisant `limit`

Soit le programme :

```
int main (void){
  int Fic1, Etat, Ret, Ret_f;

  Fic1  =  open  ("/etc/hosts",0);  /* Ouverture d un fichier  */
 /* Le pere positionne le curseur en 3 dans le fichier */
 lseek(Fic1, 3, SEEK_CUR);

  /*    Creation d un processus  fils         */
  Ret_f = fork ();

  if (Ret_f == 0) {
   /* Le fils decale le curseur de 7 positions dans le fichier */
   Ret = lseek(Fic1, 7, SEEK_CUR);
   printf("Pid : %d : Valeur de Fic1= %d et Ret=  %d\n",
          (int)getpid(), Fic1, Ret);
   exit (1);
   }

   wait(&Etat);

  /* verifier la position courante  dans le fichier
   pour chaque processus */
  Ret = lseek(Fic1, 0, SEEK_CUR);
  printf("Pid : %d : Valeur de Fic1= %d et Ret=  %d\n",
         (int)getpid(), Fic1, Ret);
  return 1;
 }
```

Résultat de l'exécution du programme précédent :

```
Pid : 869 : Valeur de Fic1= 13 et Ret=  10
Pid : 868 : Valeur de Fic1= 13 et Ret=  10
```

On constate que les deux processus trouvent le curseur en 10 :

- Le fils a trouvé le curseur en 3 et l'a avancé de 7, il est donc en 10
- Le père le voit aussi en 10.

Processus et fichiers : Partage de pointeur

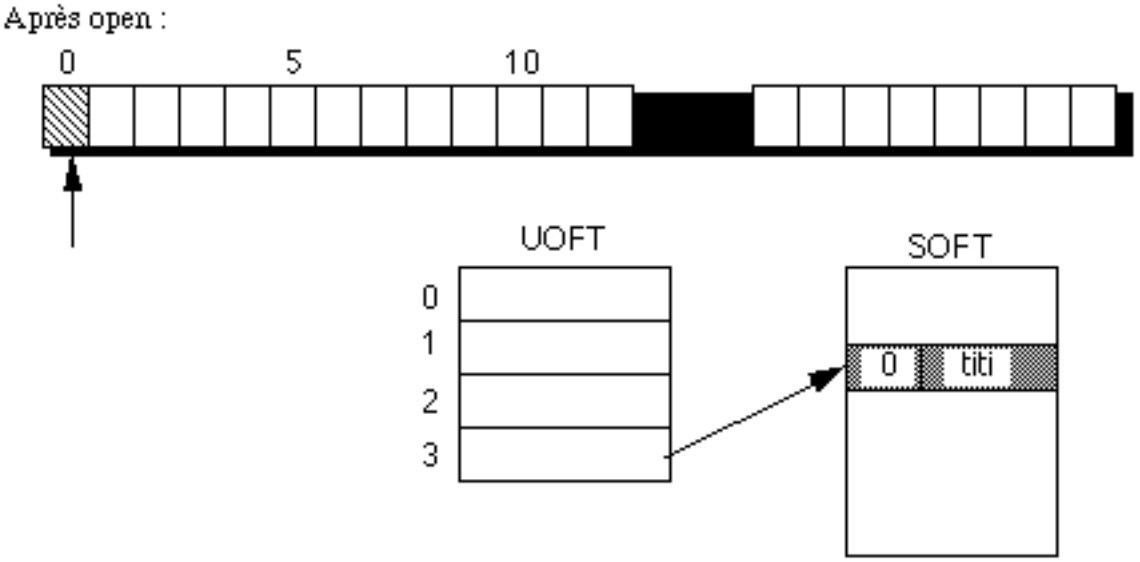
• Soit cet extrait de programme :

```
Fic1  =  open  ("titi", O_RDWR) ;

/* Positionner le curseur en 3 */
lseek(Fic1, 3, SEEK_CUR);

/* Creation d un processus  fils */
Ret_f = fork ();
...
if (Ret_f == 0) {
       /* Decaler le curseur de 7 positions */
        lseek(Fic1, 7, SEEK_CUR);
        ...
    }
...
```

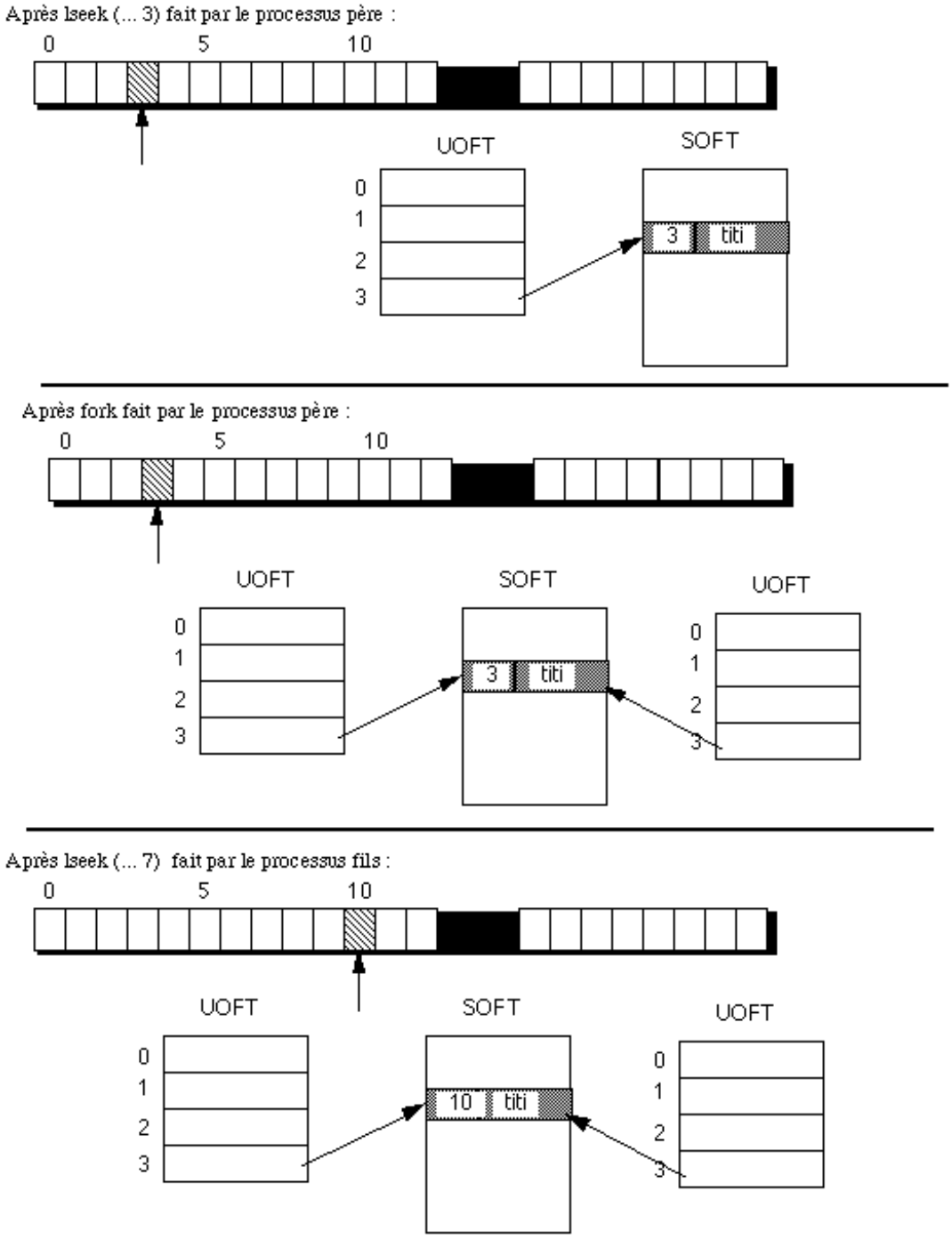
• Illustration du déplacement du curseur et de l'évolution des tables :



Voir suite…

Partage de pointeur

• …suite :



Les fichiers du type « `pipe` » (tubes)

• Les accès à un fichier déclaré comme « tube » (*pipe*) sont gérés par Unix suivant le schéma producteur/consommateur :

- Les **read** et **write** sur ce fichier seront en fait respectivement des « **consommer** » et « **déposer** »

• Conséquences :

- Les lectures sont destructrices : un caractère « lu » est retiré du tube,
- Un read est bloquant si le tube est vide
- Un write est bloquant si le tube est plein.

Les fichiers du type « `pipe` » (tubes)

• Il existe deux types de « tubes » :

– Le tube standard

  - créé par la fonction `pipe` : seuls y accèdent les processus qui ont le créateur pour ancêtre
  - il est détruit une fois que tous les processus qui l'utilisent sont terminés (*process persistent*),

– Le tube nommé (*named pipe*)

  - créé par la fonction `mkfifo` ou `mknod` : tous les processus qui connaissent son nom, et ont les droits d'accès adéquats, peuvent y accéder (il a une entrée dans la i-list),
  - *system persistent* : il n'est pas détruit quand le processus qui l'a créé se termine,

Exemple d'utilisation de la primitive pipe correspondant au schéma ci-contre.

Le père dépose dans le tube les caractères saisis au clavier par un getchar. Le fils les affiche.
Les **read** et **write** sur ce fichier seront en fait des **consommer** et **déposer**, c'est à dire que le read sera bloquant si le tube est vide et le write bloquant s'il est plein.

```
...
int  main (void){
  int Ret_Fork, Tube[2], Etat;
  char c;

  pipe(Tube);
  Ret_Fork = fork();

  if (Ret_Fork != 0 ){
...
    close(Tube[0]);
    printf("Envoyer les caracteres!\n");
    while( (c = getchar()) != EOF )
      write (Tube[1], &c, 1);
    wait (&Etat);
    }

  if (Ret_Fork == 0 ){
      printf ("Fils pret en lecture.\n");
      close (Tube[1]);
      while ( read (Tube[0], &c, 1) )
      {
       printf ("caract recu = %c (0x%x)\n", c, c);
      }
     exit (0);
    }
...
}
```

Les fichiers du type « `pipe` » (tubes)

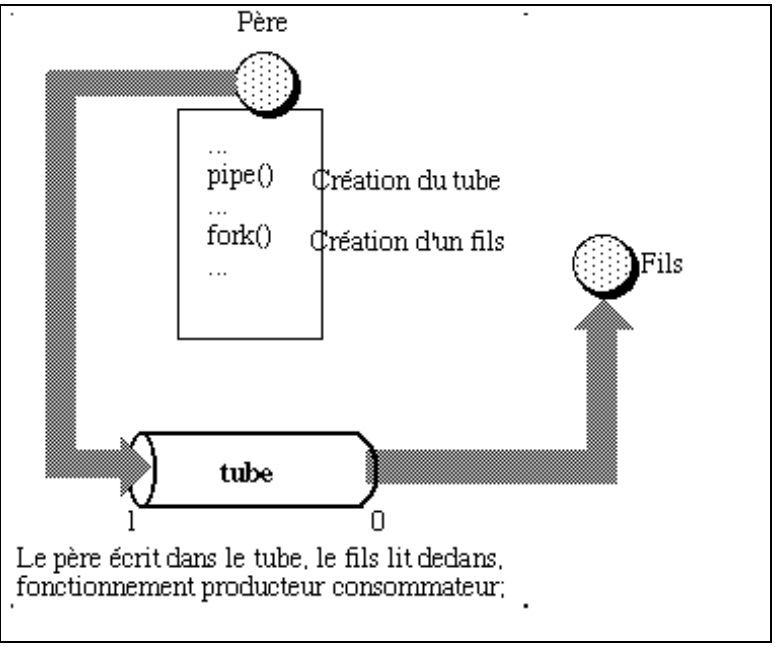
• Le programme suivant montre l'utilisation des tubes standards en C:

```
int main (void){
  int, Tube[2], Ret_Fork;
  char c;

  pipe(Tube);
  Ret_Fork = fork();

  if (Ret_Fork != 0){
   ...
   write (Tube[1], &c, 1);
   ...
   }

  if (Ret_Fork == 0){
   ...
   read (Tube[0], &c, 1);
   ...
   }
...
}
```



Exemple de création d'un tube nommé, il sera accessible à tous les utilisateurs puisque les droits sont read-write pour toutes les catégories d'utilisateurs (0666).
Les **read** et **write** sur ce fichier seront en fait des **consommer** et **déposer**, c'est à dire que le read sera bloquant si le tube est vide et le write bloquant s'il est plein.

```
/* Creation d'un tube nomme visible de partout */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

int main (int nb_arg, char ** argv){
  int Ret_mknod;
  if (nb_arg != 2){
    printf (" Utilisation : %s Nom_du_tube a creer\n",
              argv[0]);
    exit (1);
    }

  Ret_mknod = mknod(argv[1], S_IFIFO | 0666, 0);
  if ( ret_mknod != 0){
    perror (" mknod ");
    exit(1);
    }
  printf (" On a cree le tube  %s \n", argv[1]);

 ...
}
```

Utilisation des "tubes"

• Tubes standards :

ne sont vus que par les descendants d'un ancêtre commun, plusieurs producteurs/consommateurs possibles sur le tube. Ils sont créés par la commande `pipe.`

• Tubes nommés :

accessibles pour peu qu'on connaisse leur nom et qu'on ait les droits d'accès adéquts. Ils sont créés par `mknod ou mkfifo`.

• Les tubes en shell. Le symbole **|** est un tube sur lequel arrivent les sorties du processus situé à sa gauche et où le processus situé à sa droite prend ses entrées :

```
ps -aux | grep dupont
ypcat passwd | grep -i dupont
```

```
/*  ------------------------------------------------
   Lister les noms des fichiers sources C
   du repertoire courant.(On pourrait les ranger
   dans un fichier en faisant le fprint dans un fichier
   et non pas sur stdout).
  ------------------------------------------------ */
#include <stdio.h>
#include <stdlib.h>
int  main(void){
  char *cmd = "/usr/bin/ls *.c";
  char buf[BUFSIZ];
  FILE *Ptr_Fichier;
  long nbfic = 0;
  Ptr_Fichier = popen(cmd, "r");
  if (Ptr_Fichier != NULL)
    while (fgets(buf, BUFSIZ, ptr) != NULL) {
      nbfic = nbfic + 1;
      fprintf(stdout, "Fichier %ld -> %s", nbfic, buf);
    }
...
}

/*-------------------------------------------------
   Envoi de mail.
  ------------------------------------------------ */
#include <stdio.h>
int  main (int argc, char *argv[]){
  FILE *Tube;
  char Com[128];
  char Sujet [] = "essai";
   if (argc != 3) {
    printf (" Utilisation : %s destinataire message\n", argv[0]);
    exit(1);
  }
  ...
  sprintf(Com, "/usr/bin/mail -s %s %s", Sujet, argv [1]);
  Tube = popen(Com, "w");
  fprintf(Tube, argv [2]);
  fflush(Tube);
  pclose(Tube);
}
```

La primitive popen

• Description de l'appel système :

```
FILE * popen(const char *com, const char *mode_acces)
```

• Cette fonction crée un processus qui exécute la commande contenue dans `com` et qui se synchronise avec le processus appelant via un tube. L'accès au tube se fait suivant le mode d'accès désigné par `mode_acces`.

• Exemple :

```
int Nb_Util;
char *Commande = "who | wc -l";
FILE *Le_Tube;
...
Le_Tube = popen (Commande, "r");
fscanf (Le_Tube , "%d", & Nb_Util ) ;
fclose (Le_Tube);
printf ("Il y a %d utilisateurs\n", Nb_Util);
...
```

```
 /*-----------------------------------------------------
    utilisation d'un fichier-verrou
    ATTENTION, utiliser un fichier visible
    PAR TOUS LES UTILISATEURS du verrou
    cet exemple suppose deux processus
    dans le meme repertoire  !!!!!!!!!
    UTILISER LOCKF (cf. la suite)
    ----------------------------------------------------- */

#define VERROU "verrou"

int main (void){
 int f2, i, ret;

 while (1){
  printf ("Pid %d : avant entree dans SC\n", (int)getpid() );

  do{
    f2=open (VERROU, O_EXCL | O_CREAT | O_TRUNC | O_EXCL , 0666);
    sleep (2);  /* pour limiter l'attente active */
    }
   while (f2  == -1) ;

   printf ("Pid %d : entree dans SC\n", (int)getpid() );
    /*-----------------------------------------------
     Debut de section critique
     -----------------------------------------------*/
    for (i=0;  i<10000000; i++) ;
    /*-----------------------------------------------
     Fin de section critique
     -----------------------------------------------*/
  ret = remove (VERROU);
  sleep(2);
  printf ("Pid %d : sortie SC, ret = %d \n", (int)getpid(), ret );
  }
return 1;
}
```

Synchronisation des accès fichiers

• Utilisation de "fichiers-verrous" :

```
ret = open  ("Verrou", O_CREAT | O_EXCL,  0666)
```

on vérifie la valeur de `ret` pour savoir si le verrou existe ou non :

Processus 1

```
do
  {
   i = open ("Verrou", O_CREAT | O_EXCL...)
   sleep (n); /* pour limiter l'attente active */
  }
while ( i == -1);

        Section
        critique

  remove ( "Verrou" )
```

Processus 2

```
do
  {
   i = open ("Verrou", O_CREAT | O_EXCL...)
   sleep (n); /* pour limiter l'attente active */
  }
while ( i == -1);

        Section
        critique

  remove ( "Verrou" )
```

• Conditions de bon fonctionnement :

- mettre le fichier-verrou dans `/tmp` ou tout autre répertoire accessible par **tous** les utilisateurs concernés,

- donner à ce fichier-verrou un nom bien particulier

• Utiliser plutôt un **vrai** verrou (à l'aide de la fonction `lockf`) !

```
/*********************************************************
          synchro par lockf
 remarque : les processus  qui n'utilisent pas lockf
 accedent sans controle au fichier
*********************************************************/
...
int  main (int argc, char *argv[]){
...
Sortie = open (argv[1], O_RDWR)) == -1) ;
...
/* verrouillage de tout le fichier */
lockf (Sortie, F_LOCK , 0);
printf ("Pid %d : entree dans SC\n", (int)getpid() );

/*-----------------------------------------------
  Debut de section critique
  -----------------------------------------------*/
   ...
   write (Sortie, Tab, strlen(Tab);
   ...
/*-----------------------------------------------
  Fin de section critique
  -----------------------------------------------*/
...
/* se repositionner au debut du fichier , sinon on ne
  deverrouille que ce qui suit la position courante */
lseek(Sortie, 0, SEEK_SET);
lockf (Sortie,  F_ULOCK , 0);
      printf ("Pid %d :  sortie SC, =  %d\n", (int)getpid());
...
}
```

Synchronisation des accès fichiers

- Pour verrouiller les accès à un fichier : `lockf.` Cette fonction gère des « advisory locks » c'est à dire que le passage par le verrou n'est pas obligatoire pour accéder au fichier.
- Le verrouillage et le déverrouillage se font sur `nb_octets` en partant de la **position courante**. Ceci évite le problème du **faux partage** : verrouiller toute la ressource, alors que seule un sous-ensemble est utilisé.

verrouillage : `lockf(fichier, F_LOCK, nb_octets)`

déverrouillage : `lockf(fichier,F_ULOCK, nb_octets)`

où `fichier` est la valeur renvoyée par un `open, creat, dup` ou `pipe.`

si `nb_octets = 0,` tout le fichier est verrouillé.

- Interblocage possible,

• Remarque : `lockf` agit à partir de la position courante, utiliser `lseek` pourgérer cette position.

Récapitulatif : fichiers Unix

• Fichiers standards, ouverts par `open`, **(lecteurs/écrivains)** pas de synchronisation implicite :

- les processus accèdent aux fichiers en fonction de leurs seuls droits d'accès, **pas de contrôle de concurrence**,
- synchronisation possible via le "verrou à la Unix" : `creat` et test d'existence **,**
- synchronisation en utilisant `lockf` ou `flock.`

• Fichiers tubes, **synchronisés producteur/consommateur :**

- tube standard (ouvert par `pipe`) : il faut un ancêtre créateur
- tube nommé (ouvert par `mkfifo` ou `mknod`) : il suffit d'en connaître le nom
- `popen` : un tube est créé entre le processus courant et celui qui exécute la commande passée en paramètre à `popen`

Plan

1 – Généralités

- Définitions
- Organisation logique, organisation physique

2 – Organisation physique

- UNIX : i-list et i-node
- rappels sur le fonctionnement d'un disque

3 – Organisation logique

- L'arborescence UNIX,
- Différents types de fichiers
- Modes et droits d'accès

4 – Organisation logique/physique : illustration par quelques commandes : `cp, mv, rm, ln, ln -s`

5 – Processus et fichiers,

Annexe: la bibliothèque d'entrées-sorties du langage C

Fichiers standards (associés au terminal courant)

	Haut niveau	Bas niveau	Défaut
Entrée standard	`stdin`	`0`	clavier
Sortie standard	`stdout`	`1`	écran
Sortie erreur	`stderr`	`2`	écran

Ces fichiers standards sont toujours ouverts, sauf si l'utilisateur les ferme explicitement.

E/S en C sous UNIX

• On peut utiliser deux bibliothèques :

- celle dite de bas niveau (famille d'appels liés à `open` : `read, write`, etc)
- celle dite de haut niveau (famille d'appels liés à `fopen` : `fread, fwrite`, etc)

• Relation entre processus et fichier : on va associer une variable locale au fichier dont on connait le nom global :

- **Haut niveau :** pointeur sur une structure décrivant le fichier (variable de type `FILE*` renvoyée par `fopen`)
- **Bas niveau :** descripteur de fichier :c'est à dire un index dans la table des fichiers ouverts par le processus (variable de type `int,` renvoyée par `open`)

• Par la suite on présente la bibliothèque de bas niveau (`open, read, write`, etc), les fonctions de haut niveau seront données dans les pages de commentaires

Ces fonctions de bas niveau sont spécifiques à UNIX.
`open` renvoie une entrée dans la table des fichiers ouverts par le processus. Elle associe un numéro local à un nom global.

Ouverture et fermeture des fichiers avec la **biblio. de haut niveau** :

```
FILE * fopen (char * nom_externe, char * type)
```

"r"	lecture seule
"w"	écriture seule; s'il existe, il est détruit
"a"	concaténation; création s'il n'existe pas
"r+"	lecture et écriture (mise à jour)
"w+"	écriture et lecture
"a+"	lecture et concaténation

Exemple d'utilisation de `fopen`, on ouvre le fichier dont le nom est passé sur la ligne de commande :

```
#include <stdio.h>
int main  (int argc, char *argv[] ){

  FILE *Ptr_Fichier;
  ...
  Ptr_Fichier = fopen (argv[1],"r");
  if (Ptr_Fichier == (FILE *) 0) {
    printf ("Pas pu ouvrir %s !\n", argv[1]);
    exit (1);
    }
  printf ("Fichier %s ouvert !\n", argv[1]);
  ...
}
```

Fermeture : `fclose (FILE *fp)`
Par exemple pour fermer l'entrée standard :

```
fclose (stdin);
```

E/S bas niveau

• Avant d'accéder à un fichier il faut l'ouvrir par un **open :**

```
int Fichier;
...
Fichier = open ("toto", O_RDWR);
  if (Fichier == -1) {
      perror ("open toto");
      exit (1);
    }
```

• Création de fichier

```
int creat (char *nom, int mode/* droits d'acces
  */);
```

```
/*******************************************************
     exemple d'open, creat, read et write
*******************************************************/
...
int main (int argc, char *argv[]) {
  int i, f1, f2;
...
  f1  =  open  (argv[1],0);
  if (f1 == -1){
    perror ("open");
    exit (1);
    }
  f2  =  creat (argv[2],0666);
  if (f2 == -1){
    perror ("creat");
    exit (1);
    }

  while(1){
     i = read (f1,&c,1);
     if (i != 1) break;
     write (f2, &c, 1);
    }
}
...
```

Bibliothèque de haut niveau.

on lit ou écrit des enregistrements dont on donne la taille et le nombre :

Lecture :

```
int fread (char * buf, int size, int nb_rec, FILE * fp)
```

- Valeur retournée : nombre d'enregistrements de taille `size` lus.
- En cas d'erreur ou fin de fichier renvoie 0

Ecriture :

```
int fwrite (char * buf, int size, int nb_rec, FILE * pf)
```

- Valeur retournée : nombre d'enregistrements de taille `size` écrits.

E/S bas niveau

- Lecture / Ecriture :

```
int write (int fildes, char * buffer, unsigned
  nbyte)
int read (int fildes, char * buffer, unsigned
  nbyte)
```

- Fermeture d'un fichier :

```
int close (int fildes) ;
```

Attention aux valeurs de retour (utiliser `man` pour les vérifier).

- Exemple :

```
int main ( void ){
int Mon_Fic, Ret_Read, Ret_Write;
char Tableau [512] ;
/* ouverture en lecture*/
Mon_Fic= open ("toto", O_RDONLY) ;
if (Mon_Fic== -1) {
   perror ("open");
   ...
   }
while((Ret_Read=read(Mon_Fic, Tableau, 512)) > 0){
   ...
   Ret_Write = write(1, Tableau, Ret_Read);
   if (Ret_Write == -1)  { / * Traiter l'erreur */       };
 }
close (Mon_Fic) ;
}
```

Informations sur un fichier :

```
void stat (char * path, struct stat *buffer) ;
void fstat (int fildes, struct stat *buffer) ;
int access (char * path, int access_mode) ;
```

Bibliothèque de haut niveau (équivalent de lseek)

```
long fseek (FILE *pf, long offset, int from)
```

valeur de from	0	1	2
déplacement depuis	début	position courante	fin

`offset` est un déplacement en octets. il peut être posifif ou négatif.
`fseek` ne s'applique pas aux fichiers "`pipes`".

- Retourne 0 si ok, sinon une valeur ≠0 (!!!)

Pour obtenir la P OSITION COURANTE :
`long ftell (FILE *pf)`
- retourne -1 si erreur.

E/S bas niveau

- Positionnement dans un fichier :

```
long lseek (int fildes,long offset,int from) ;
        from :
        0       depuis le début
        1       depuis la position courante
        2       à partir de la fin
```

Retourne :
- la position du pointeur à partir du début du fichier,
- -1 en cas d'erreur.

- Changer les droits d'accès :

```
int chmod (char *nom, int mode) ;
```

Exemple d'utilisation de fgets sur stdin :

```
while ( fgets(Tab, sizeof(Tab), stdin) )
   {
     if (!strncmp (Tab, "fin", 3)) break;
     if (write (Sortie, Tab, strlen(Tab)) < 0)
       {
         printf ("Erreur d'ecriture\n");
         break;
       }
```

E/S par caractère et par ligne

• Par caractère :

Lecture	Ecriture

`int getc(FILE *fp)`
`int fgetc(FILE *fp)`
`int getchar()` *idem getc(stdin)*
`int getw(FILE *fp)`
	`int putc(char c, FILE *pf)`
	`int fputc(c, pf)`
	`int putchar(c)` *idem putc(c,stdout);*
	`int putchar(c)`

• Par ligne, par convention, une ligne de fichier texte se termine par `'\n'`.

Lecture	Ecriture

`char * gets(char *buffer)` *: depuis stdin*
 	
`char * fgets(char *buffer,`
`int longmax, fp)`
	`intputs(char *chaine)` *: vers stdout*
	
	`int fputs (char *chaine, fp)`

Format pour les appels de la famille printf :

Spécificateur	Correspondance
`d`	entier
`u`	entier non signé
`f`	flottant
`x`	hexadecimal
`c`	caractère
`s`	chaîne de caractères
`g`	flottant double

Options :

`l` : complète le spécificateur pour les entiers longs- : justifier à gauche

\+ : visualiser le signe (+ ou -)

0 : compléter par des 0

Formats variables :

printf ("%*.*s", 9, 5, chaine) ;

etc...(cf. man)

Formats pour les appels de la famille scanf :

`%d`	décimal
`%x`	hexadécimal
`%s`	lit une chaine jusqu'à blanc, TAB ou LF ajoute un '/n' à la chaine
`%c`	lit un caratère
`%[..]`	Si le caractère lu fait partie de l'ensemble, on continue la scrutation et l'affectation. Les autres caractères sont considérés comme des séparateurs. Dans le cas où le caractère '^' apparait en 1ère position (`%[^..]`), il s'agit du complément de cet ensemble.
`%*d`	Pas d'affectation à une variable pour ce format d'entrée
`...`	.cf le man ..

E/S standards formatées :

	LECTURE	ECRITURE
clavier/écran	`scanf`	`printf`
Fichier	`fscanf`	`fprintf`
Mémoire	`sscanf`	`sprintf`

- Syntaxe :

```
printf  (format, liste de valeurs)
fprintf (fichier, format, liste de valeurs)
sprintf (buffer, format, liste de valeurs)

scanf   (format, liste d'adresses)
fscanf  (fichier, format, liste d'adresses)
sscanf  (buffer, format, liste d'adresses)
```

- Exemples :

```
fprintf (stderr, "usage: %s [-d] [conf]", progname);
fprintf (stderr, "httpd: could not get socket\n");
sprintf (identite, "Client: %s@%s", user, machine);
```

E/S standards formatées : fscanf , exemple

• scanf retourne le nombre d'éléments affectés. Il s'arrête lors du premier conflit.

```
/*  lecture dans un fichier ou sur /dev/tty :
 on lit des mots et on s arrete sur EOF        */

#include <stdio.h>
#include <stdlib.h>

#define MAX_MOTS 80

int main(int argc, char *argv[]){
  int ret_sc;
  FILE *Fichier;
  char Mots_lu[MAX_MOTS];

    Fichier = fopen(argv[1], "r");
  if(Fichier == NULL){
      printf("pb ouverture %s\n", argv[1]);
      return 1;
  }

  ret_sc=0;
  while ( (ret_sc != EOF) ){
      ret_sc=fscanf(Fichier,"%s", Mots_lu);
      if (ret_sc == 1) printf("mot lu : %s\n", Mots_lu);
  }

  printf("\n main : fin\n");
  return 0;
}
```