Gestion des fichiers

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,
- Fonctionnement de commandes :
  `cp,mv,rm,ln,ln -s`

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).

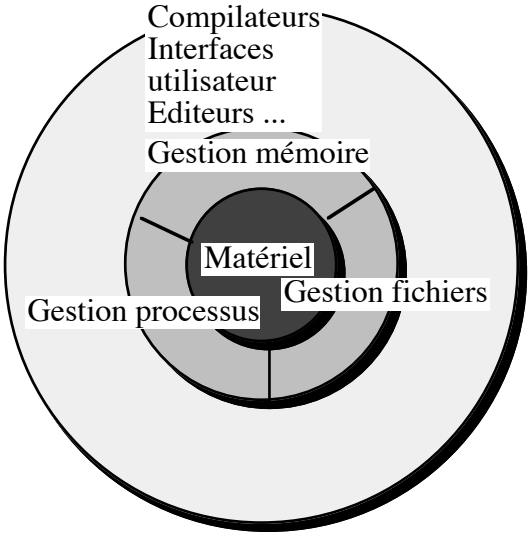
Pour les applications qui utilisent plusieurs machines simultanément (client/serveur, par exemple) on adopte des représentations normalisées (format XDR des RPC, CDR de CORBA, ou primitives hton() et ntoh() de la bibliothèque des sockets).

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`) :

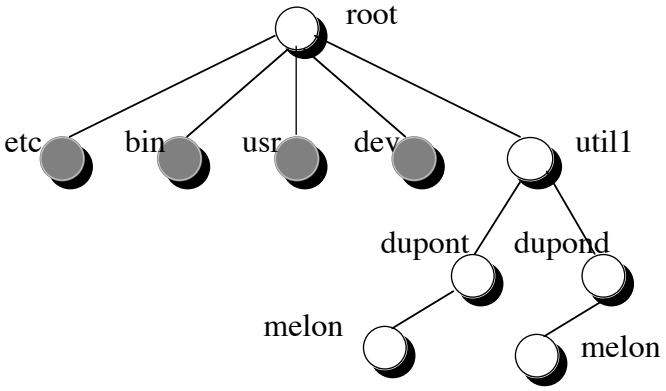


- Le SGF (comme d'autres gestionnaires de ressources) peut être déporté à l'extérieur du noyau, ou ne pas exister (systèmes embarqués)

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,
   - Fonctionnement de commandes :
     `cp,mv,rm,ln,ln -s`

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'implantation 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 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 :

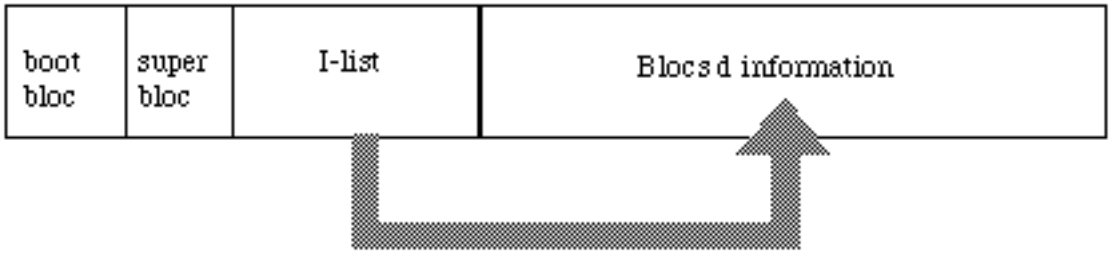
  - T grand, pour permettre la création d'un grand nombre de fichiers : une grande surface disque est alors confisquée par la i-list, peut-être de façon inutile,

  - 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 du `file system` sous UNIX



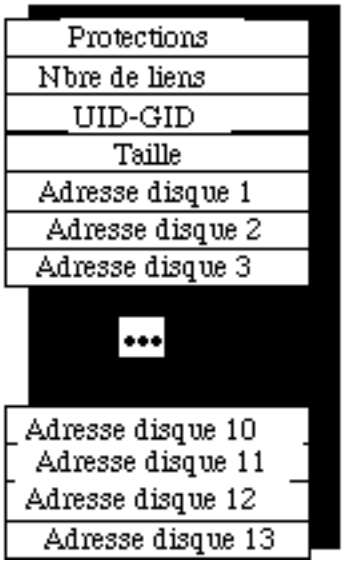
- Si ce file system permet le démarrage du système, alors le *boot-bloc* contient les informations de démarrage (*bootstrap*),
- Le *super-bloc* contient, entre autres, les informations suivantes :
  - La taille en blocs de la *i-list,* celle du volume,
  - La liste des blocs libres,
  - Le nom du système 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 « historique » 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.
Taille	Taille, en octets, du fichier (Remplacée par les numéros de majeur et de mineur pour les fichiers de `/dev`).
Adresse disque	chacune de ces entrées peut contenir un numéro de bloc sur le disque (détails page suivante).
Nbre 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 (1/4)

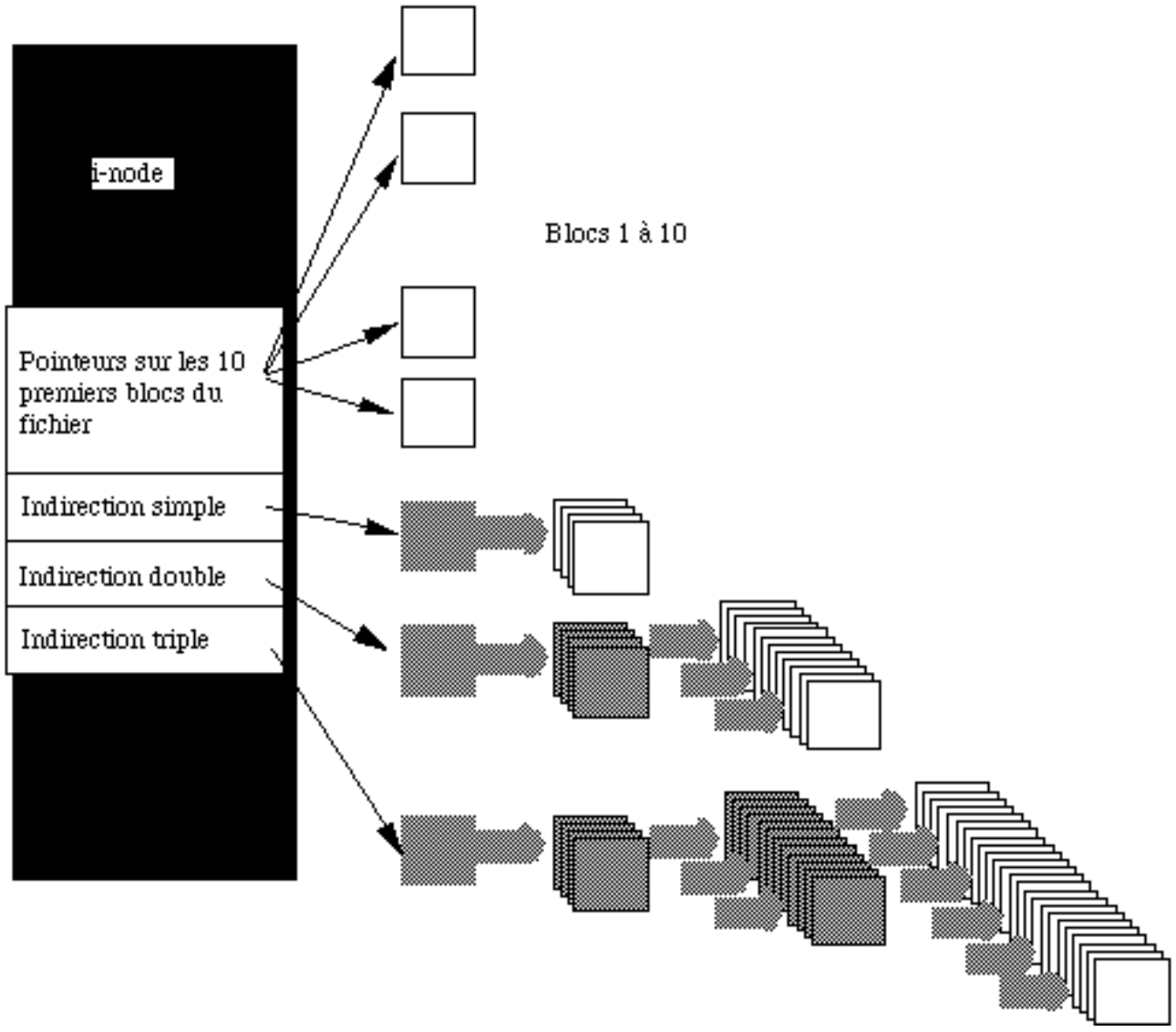
- Dans chaque i-node on trouve13 adresses de blocs (à l'origine, 1 bloc = 512 octets, 1 adresse= 4 octets) :
  - les 10 premières pointent sur les 10 premiers blocs du fichier (B_0, ..., B_9),
  - la 11ème pointe sur un bloc de 128 pointeurs qui adressent les 128 blocs de données suivants (B_{10} à B_{137}),
  - 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 (2/4) : illustration

• Sur UNIX originel : 1 bloc = 512 octets



Du i-node aux blocs alloués sur disque (3/4)

- Exemple avec des blocs de 4 Ko, une adresse de bloc=4octets :
  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 1324.

Du i-node aux blocs alloués sur disque (4/4)

- 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

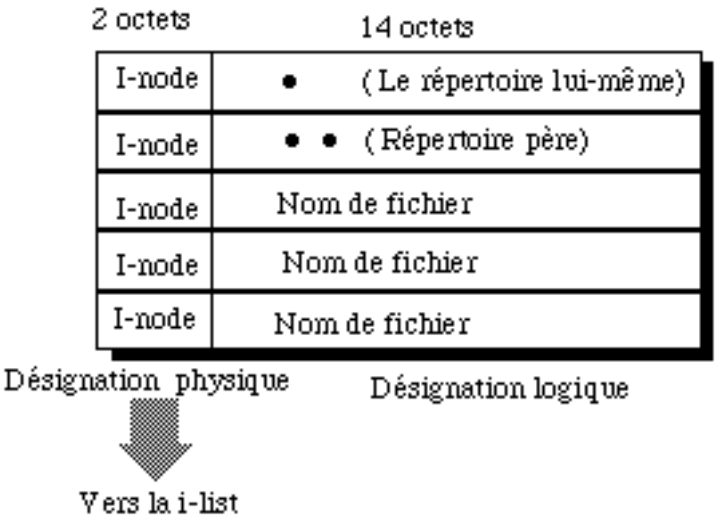
On remarquera que le répertoire UNIX reflète parfaitement la séparation organisation logique / organisation physique.
La première organisation concerne l'utilisateur et la seconde le système.
Chacune des entrées du répertoire comporte deux champs : l'un contient le i-node qui décrit complètement le fichier, 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 (cf. virtualisation : l'utilisateur voit les **noms** des ressources, pas leur adresse),
• exemples de commandes concernant les répertoires :
  – `cd` (déplacement dans l'arbre),
  – `ls` (affichage des seuls **noms** des fichiers contenus dans le 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.

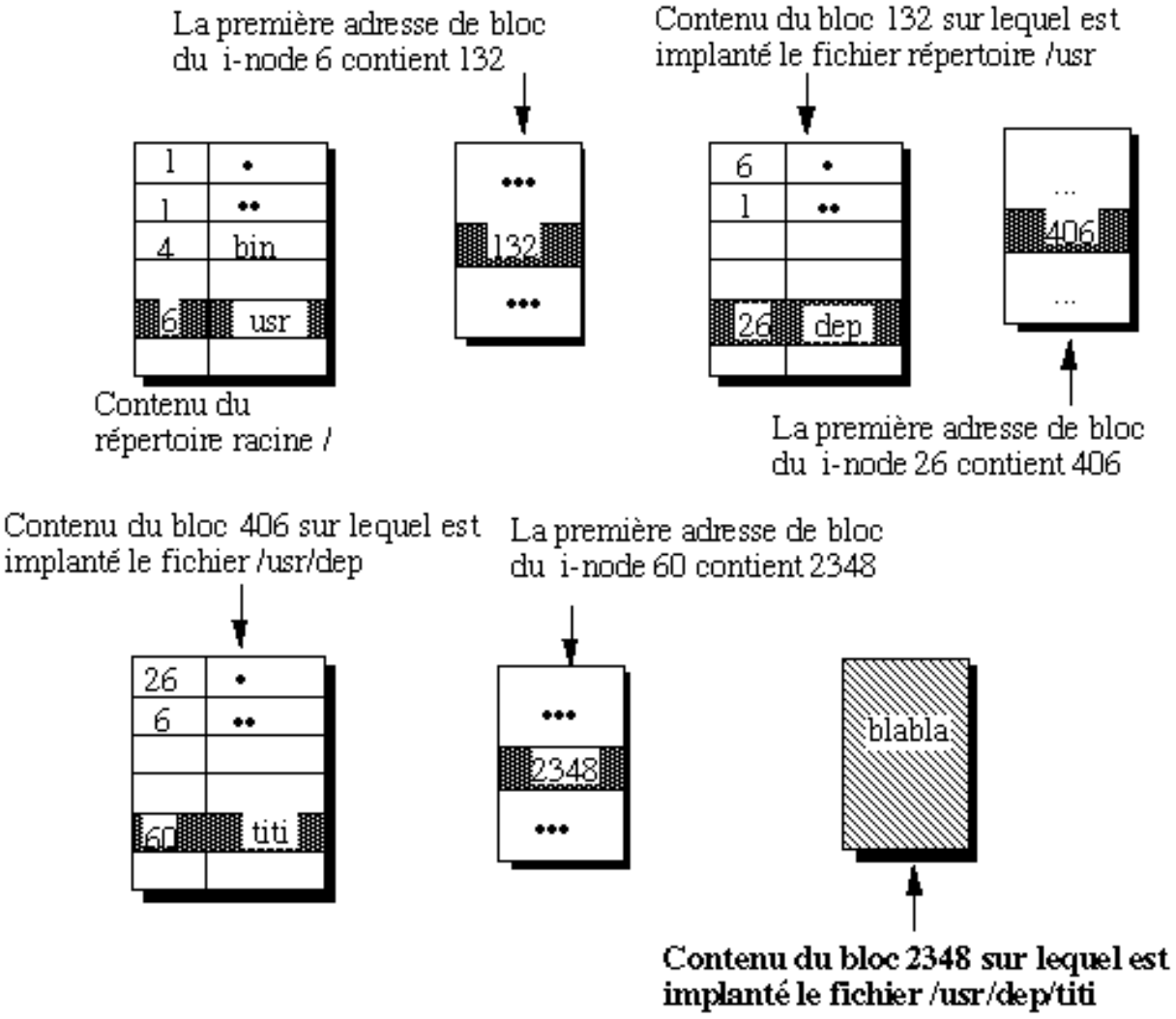
La commande donnée peut échouer si celui qui l'a émise n'a pas le droit de traverser un répertoire menant vers le fichier, s'il n'a pas les droits de lecture sur le fichier, ou s'il n'a pas le droit d'utiliser la commande, ici cat...

Exemple d'accès à un fichier

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

2 octets	14 octets
No d'inode	Nom du fichier

- 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 :

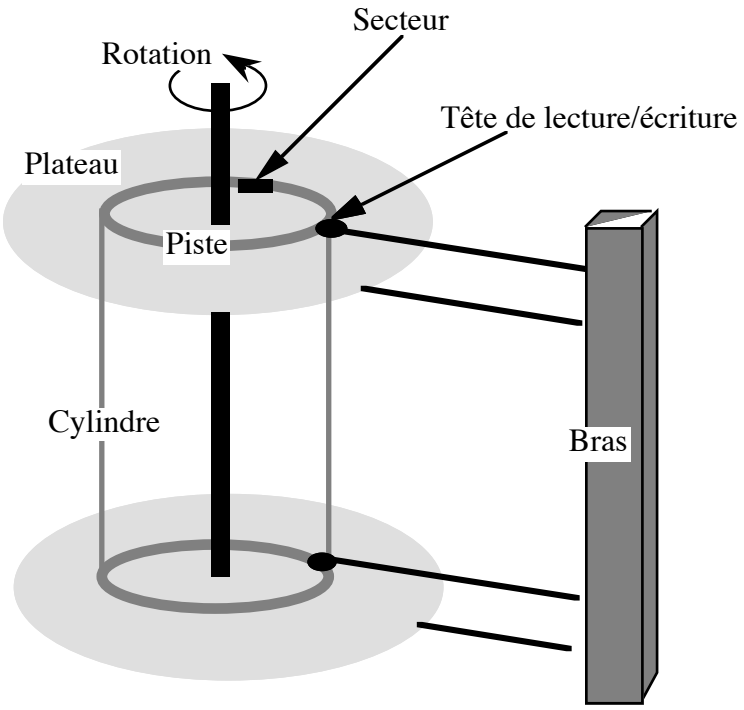
- longueur d'une piste : 32768 octets,
- temps d'une rotation : 16.67 ms,
- temps moyen de déplacement du bras : 30 ms.

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.

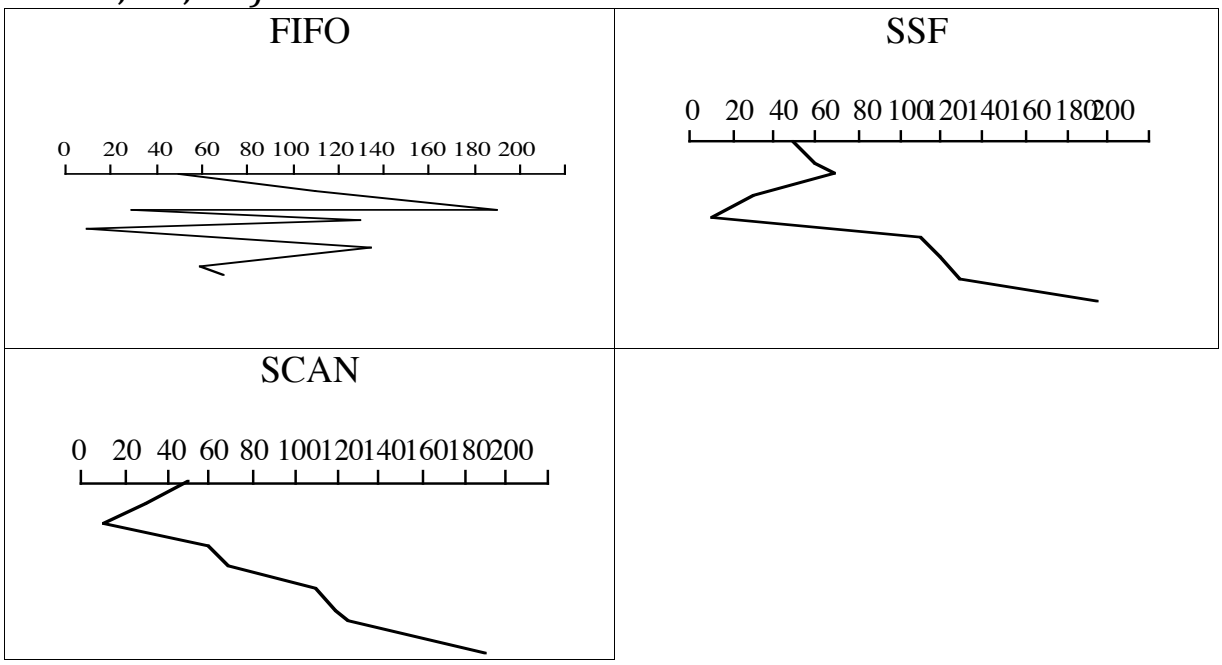
Si on utilise SCAN à partir de 50, on classe les pistes à lire en deux suites :
l'une décroissante à partir de 50 vers la piste 0 : (30,10),
l'autre croissante vers la fin du disque : (60, 65, 110, 120, 122, 190).

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

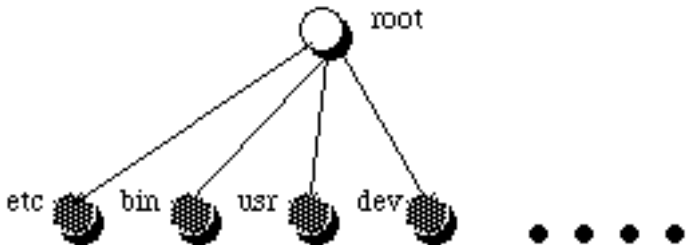
1. Généralités
   - Définitions
   - Organisation logique, organisation physique
2. Organisation physique
   - UNIX : i-list et i-node
   - Rappels sur le fonctionnement d'un disque
3. **Organisation logique**
   - L'arborescence UNIX,
   - Fonctionnement de commandes :
     `cp,mv,rm,ln,ln -s`

Annexes :

- La bibliothèque d'entrées-sorties du langage C
- Fichiers Unix : types, droits d'accès

Organisation logique : Une arborescence

- L 'exemple choisi est l'organisation logique du système Unix
- Commentaires sur quelques répertoires d'administration



  - `/etc` contient les fichiers d'administration tels que `passwd, group, hosts`,...
  - `/bin et /usr/bin` abritent les commandes du shell, ...,
  - `/usr/include` renferment les fichiers de type « `.h` »,
  - `/dev` est destiné aux fichiers spéciaux (les périphériques) : les périphériques sont ainsi banalisés et vus comme des fichiers,
- Un répertoire bien pratique :
  - `/tmp` dans lequel tout le monde peut lire et écrire.

Commande cp

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

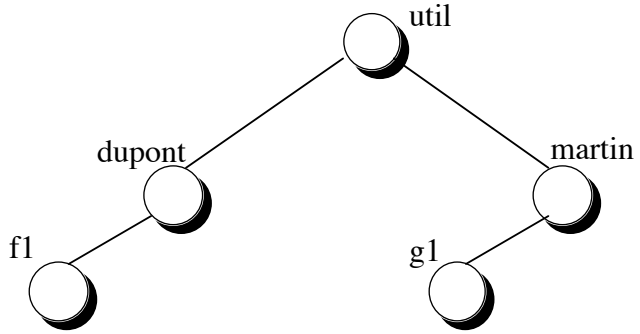
– chercher un i-node libre dans la i-liste,
– s'il en existe un, chercher de la place libre sur le disque,
– si il y en a assez, copier les blocs, mettre à jour le i-node trouvé,

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:



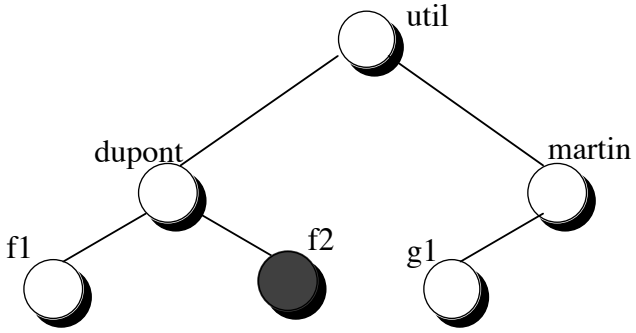
Catalogue dupont

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

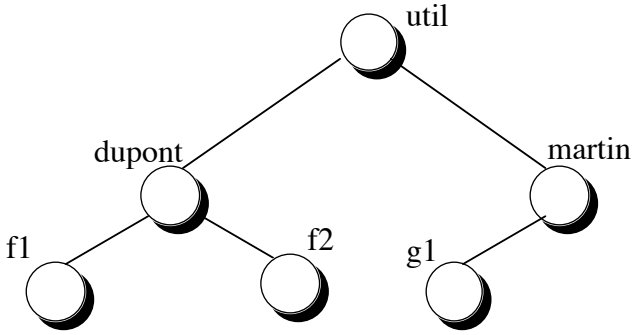
Catalogue martin

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:



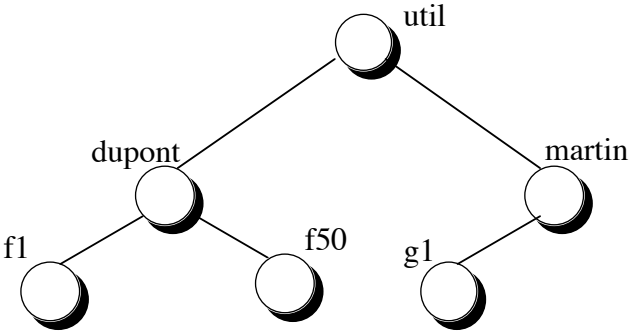
Catalogue dupont

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.

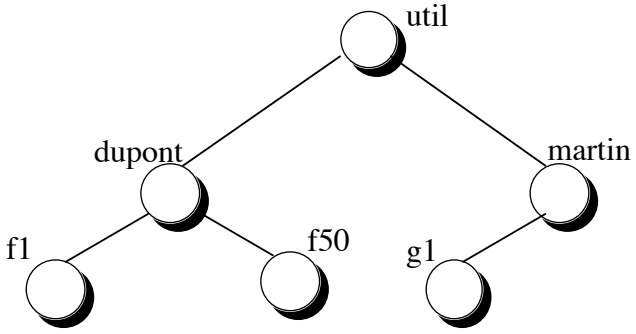
Commande *ln* (lien)

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 :



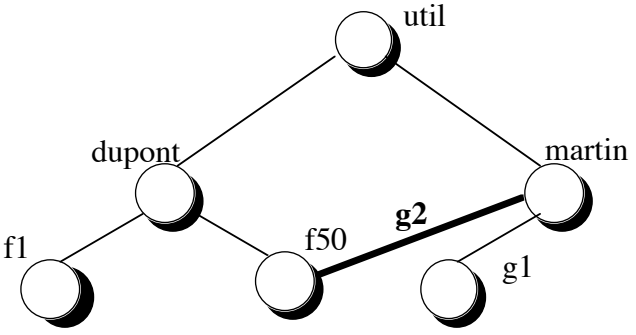
Catalogue dupont

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` *(1/2)*

- 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 :

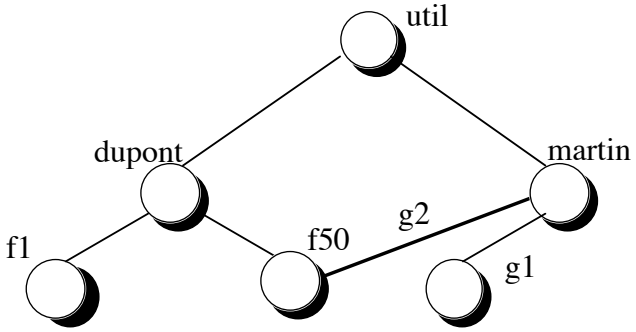
```
ln -s ../dupont/f50  g3
```

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` *(2/2)*

• `martin` fait : `ln -s ../dupont/f50 g3`
• Etats de l'arborescence et du répertoire **avant** l'exécution :



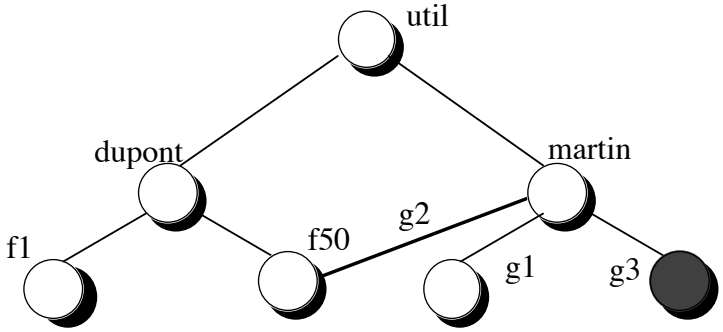
Catalogue dupont

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
```

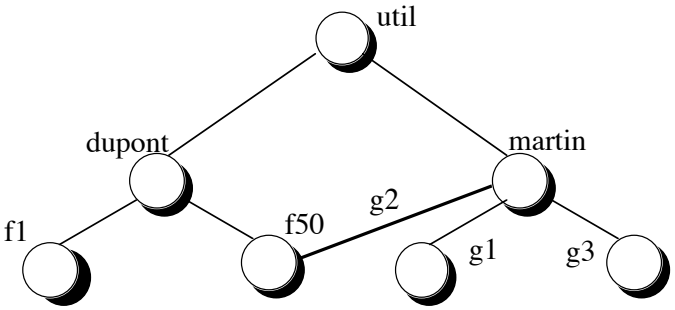
• Contenu du répertoire `dir1`, obtenu par `ls -il` :

```
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**.

Commande `rm`

• Situation initiale :



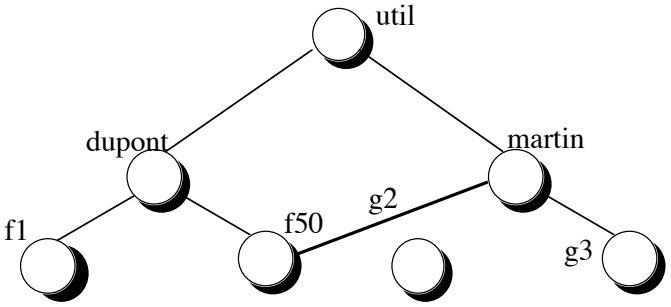
Catalogue dupont

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

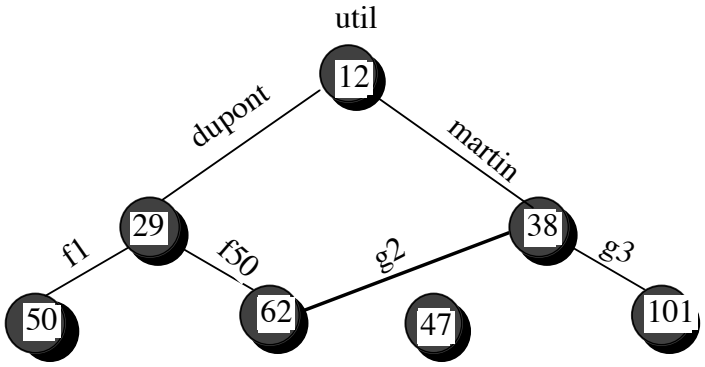
La commande `rm` (`remove`) porte bien son nom (contrairement à de nombreuses commandes Unix), elle ne veut pas dire `delete` : seul l'arc conduisant au fichier est détruit : **nbre de liens =nbre de liens - 1**

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.

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.

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.

Unix : différents types de fichiers

- Plusieurs catégories :
  - les répertoires (ce sont les nœuds qui ont un successeur dans l'arbre),
  - les fichiers ordinaires (exécutables, données…) : ce sont des sommets pendants dans l'arbre),
    - parmi eux se trouvent des fichiers spéciaux : les périphériques (dans /dev), les liens symboliques.

- Exemple. On obtient le type d'un fichier grâce à la commande `ls` et à ses diverses options **:**

```
$ ln -s fic.txt lien
$ cp fic.txt copie.txt
$ ls -il
79005292 -rw-------  1 dupont grp1  6 May 24 15:12 copie.txt
79005289 -rw-------  1 dupont grp1  6 May 24 15:11 fic.txt
79005291 lrwxrwxrwx  1 dupont grp1  7 May 24 15:12 lien -> fic.txt
```

- les fichiers de `/dev` sont en mode bloc (`b`) ou caractère (`c`). En mode bloc, on utilise le cache disque en mémoire, pas en mode caractère :

```
7611 brw-rw----  1 root disk      8,1 May 22 21:47
1055 crw--w----  1 root tty       4,0 May 22 21:47 tty0
```

Unix : structure des fichiers et modes d'accès sous

- 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 :

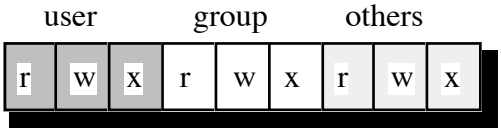
```
$mkdir   private
$chmod   711     private
$cd      private
$mkdir   dir1
$chmod   755     dir1
```

• 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

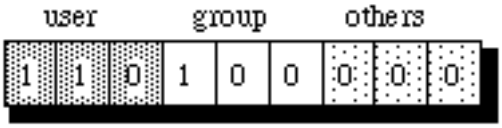
Unix : les droits d'accès aux fichiers

- 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.

Unix : 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...