Système de Gestion de Fichiers

V. Risch

IUT, Aix-Marseille Univ., 2018

Remerciements à A. Dragut

Plan de cette partie

1. Notions générales, fichier logique, fichier physique
2. Systèmes de fichiers UNIX
3. Primitives d'accès et de manipulation des fichiers

Notions générales, fichier logique, fichier physique

Système de Gestion de Fichiers

Le *Système de Gestion de Fichiers* d'un S.E. assure la conservation des données sur un support de masse non-volatile (disque dur, clé usb, CD-ROM...)

Système de Gestion de Fichiers

Le *Système de Gestion de Fichiers* d'un S.E. assure la conservation des données sur un support de masse non-volatile (disque dur, clé usb, CD-ROM...)

Un *fichier* représente une unité de stockage *indépendante* des propriétés physiques des supports utilisés

Système de Gestion de Fichiers

Le *Système de Gestion de Fichiers* d'un S.E. assure la conservation des données sur un support de masse non-volatile (disque dur, clé usb, CD-ROM...)

Un *fichier* représente une unité de stockage *indépendante* des propriétés physiques des supports utilisés

On distingue deux niveaux :

- le *fichier logique* : ensemble des données contenues dans le fichier telles que *vues par l'utilisateur*
- le *fichier physique* : espace de stockage tel qu'*alloué physiquement* sur le support de masse

Fichier logique

Un *fichier logique* est

- Un type de données standard défini par les langages de programmation et sur lequel s'appliquent des opérations de *création* et *ouverture*, *fermeture* et *destruction*. En particulier,

Fichier logique

Un *fichier logique* est

- Un type de données standard défini par les langages de programmation et sur lequel s'appliquent des opérations de *création* et *ouverture*, *fermeture* et *destruction*. En particulier,
  - création et ouverture permettent d'établir la *liaison* du fichier logique avec le fichier physique correspondant

Fichier logique

Un *fichier logique* est

- Un type de données standard défini par les langages de programmation et sur lequel s'appliquent des opérations de *création* et *ouverture*, *fermeture* et *destruction*. En particulier,
  - création et ouverture permettent d'établir la *liaison* du fichier logique avec le fichier physique correspondant
  - fermeture et destruction rompent cette liaison

Fichier logique

Un *fichier logique* est

- Un type de données standard défini par les langages de programmation et sur lequel s'appliquent des opérations de *création* et *ouverture*, *fermeture* et *destruction*. En particulier,
  - création et ouverture permettent d'établir la *liaison* du fichier logique avec le fichier physique correspondant
  - fermeture et destruction rompent cette liaison
- Un ensemble d'enregistrements, vus chacun comme une unité logique de traitement liés par la sémantique du programme qui les manipule, et accessibles par des opérations de *lecture* ou *écriture*

Modes d'accés logiques

L'accés à un fichier peut être :

- *séquentiel* : les enregistrements sont traités dans l'ordre de leur placement dans le fichier
- *direct* (ou *relatif*) : un enregistrement est atteint en spécifiant sa position relative par rapport au début du fichier
- *indexé* (ou *aléatoire*) : un enregistrement peut être atteint directement, indépendamment de son placement, grâce à une clé d'accés

Fichier physique

Il s'agit de l'entité allouée sur le support permanent et qui contient physiquement les enregistrements définis dans le fichier logique : en général un disque dur.

Fichier physique

Il s'agit de l'entité allouée sur le support permanent et qui contient physiquement les enregistrements définis dans le fichier logique : en général un disque dur.

Disque : ensemble de plateaux empilés verticalement sur un axe, et dont les faces sont divisées en *pistes* concentriques, elles-mêmes divisées en *secteurs*

Fichier physique

Il s'agit de l'entité allouée sur le support permanent et qui contient physiquement les enregistrements définis dans le fichier logique : en général un disque dur.

Disque : ensemble de plateaux empilés verticalement sur un axe, et dont les faces sont divisées en *pistes* concentriques, elles-mêmes divisées en *secteurs*
Bloc : ensemble de secteurs représentant la plus petite unité d'échange entre le disque et la mémoire centrale

Fichier physique

Il s'agit de l'entité allouée sur le support permanent et qui contient physiquement les enregistrements définis dans le fichier logique : en général un disque dur.

Disque : ensemble de plateaux empilés verticalement sur un axe, et dont les faces sont divisées en *pistes* concentriques, elles-mêmes divisées en *secteurs*
Bloc : ensemble de secteurs représentant la plus petite unité d'échange entre le disque et la mémoire centrale

Cylindre : ensemble des pistes de même rayon sur tous les plateaux

Fichier physique

Il s'agit de l'entité allouée sur le support permanent et qui contient physiquement les enregistrements définis dans le fichier logique : en général un disque dur.

Disque : ensemble de plateaux empilés verticalement sur un axe, et dont les faces sont divisées en *pistes* concentriques, elles-mêmes divisées en *secteurs*
Bloc : ensemble de secteurs représentant la plus petite unité d'échange entre le disque et la mémoire centrale

Cylindre : ensemble des pistes de même rayon sur tous les plateaux

adresse disque = (face, cylindre, secteur)

Fichier physique

Il s'agit de l'entité allouée sur le support permanent et qui contient physiquement les enregistrements définis dans le fichier logique : en général un disque dur.

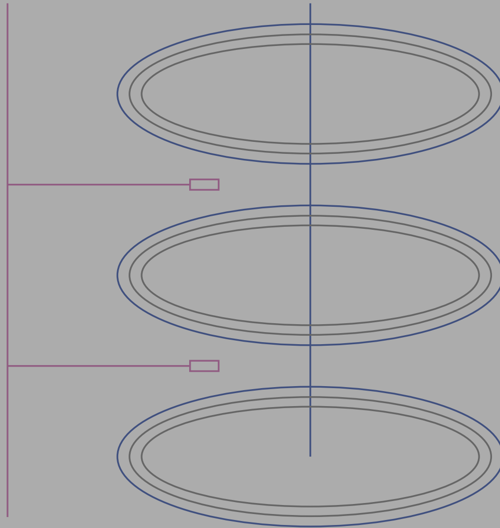
Disque : ensemble de plateaux empilés verticalement sur un axe, et dont les faces sont divisées en *pistes* concentriques, elles-mêmes divisées en *secteurs*
Bloc : ensemble de secteurs représentant la plus petite unité d'échange entre le disque et la mémoire centrale

Cylindre : ensemble des pistes de même rayon sur tous les plateaux

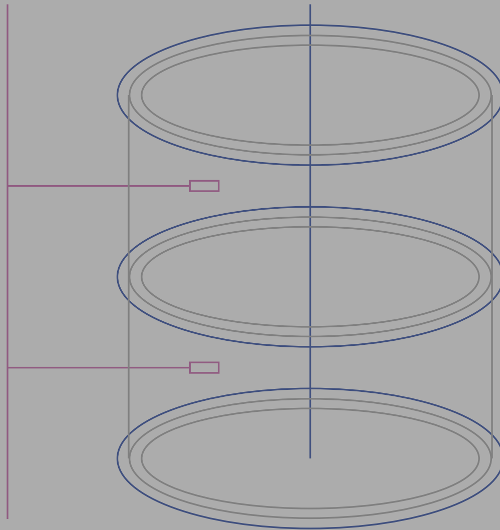
adresse disque = (face, cylindre, secteur)

→ *Un fichier physique est donc constitué d'un ensemble de blocs alloués au fichier logique*

Représentation d'un disque dur



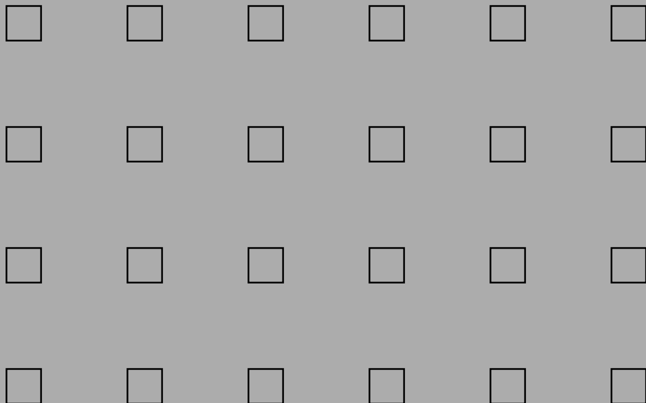
Représentation d'un disque dur



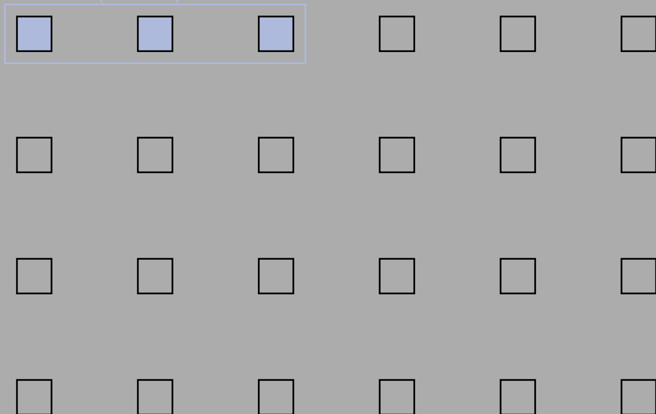
Méthodes d'allocation

- *contigue* : un même fichier occupe un ensemble de blocs contigus
  - *First Fit* : choix du premier espace libre de taille au moins égale à la taille du fichier
  - *Best Fit* : choix de l'espace dont la taille est au moins égale à la taille du fichier et qui génère le plus petit résidu
- *par zones* : variante de l'allocation contigue
- *par blocs chaînés* : liste chaînée de blocs répartis n'importe où sur le disque
- *indexée* : les adresse des blocs constituant un fichier sont rangées dans une table d'index, contenue dans un bloc particulier

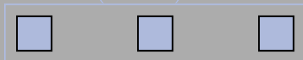
Allocation contigue

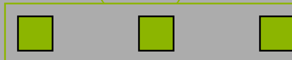


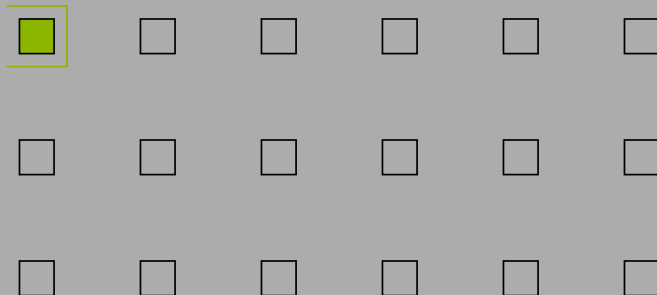
Allocation contigue



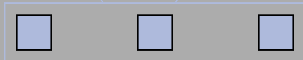
Allocation contigue

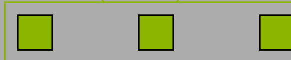


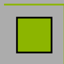


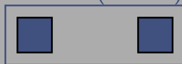


Allocation contigue



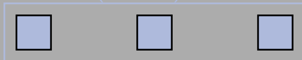




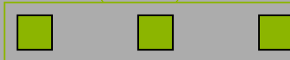


Allocation contigue

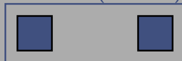
Fichier 1 (3 blocs)



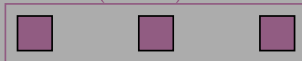
Fichier 2 (4 blocs)



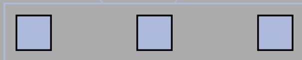
Fichier 3 (2 blocs)

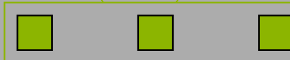


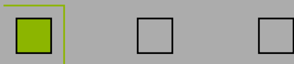
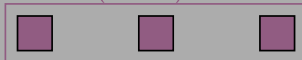
Fichier 4 (2 blocs)



Allocation contigue

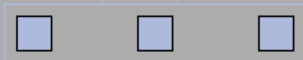




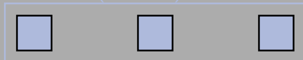
Allocation contigue

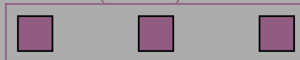
Fichier 1 (3 blocs)



Fichier 4 (2 blocs)

Allocation contigue

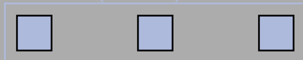




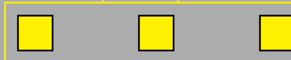
→ *Fragmentation de l'espace disque*

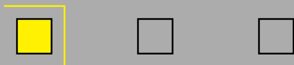
Allocation contigue

Fichier 1 (3 blocs)

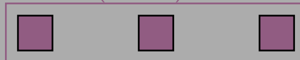


Fichier 5 (4 blocs), *BEST FIT*



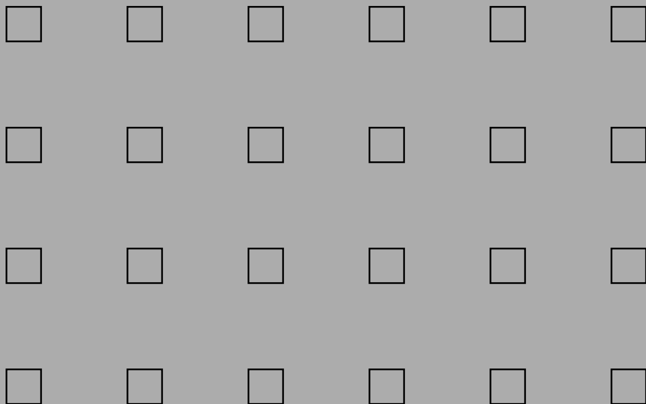


Fichier 4 (2 blocs)

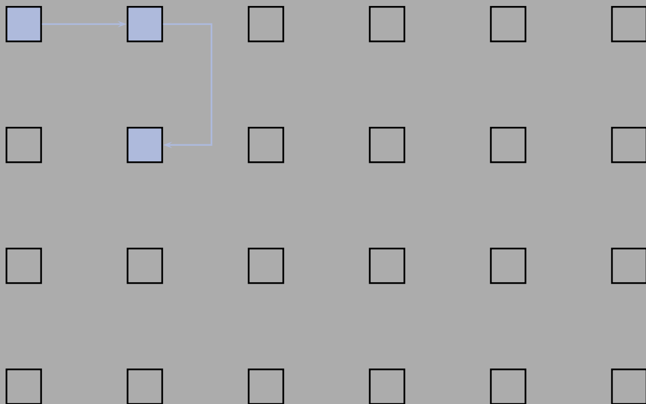


→ *Fragmentation résiduelle (mais fragmentation quand même...)*

Allocation par blocs chaînés



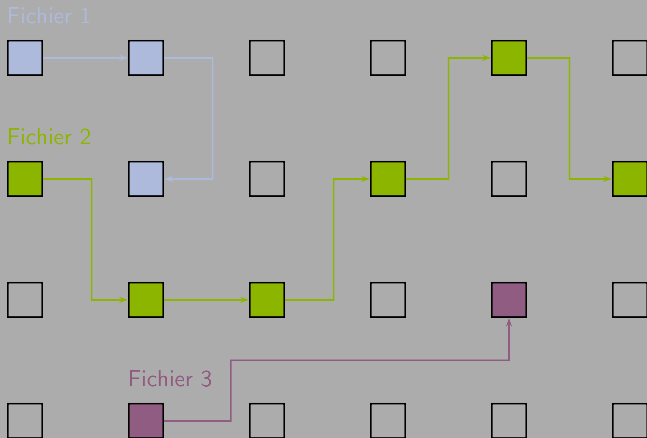
Allocation par blocs chaînés



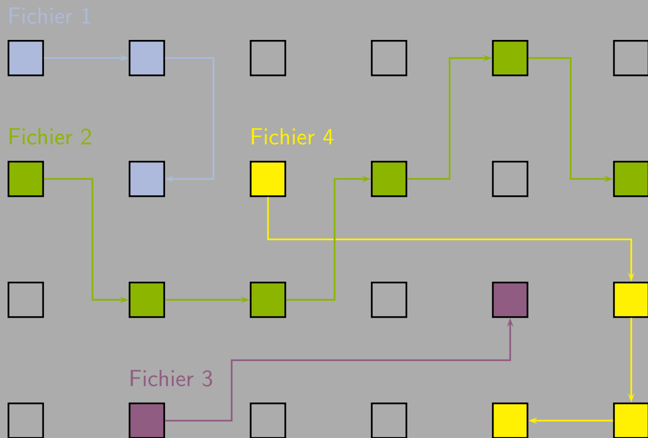
Allocation par blocs chaînés

Allocation par blocs chaînés

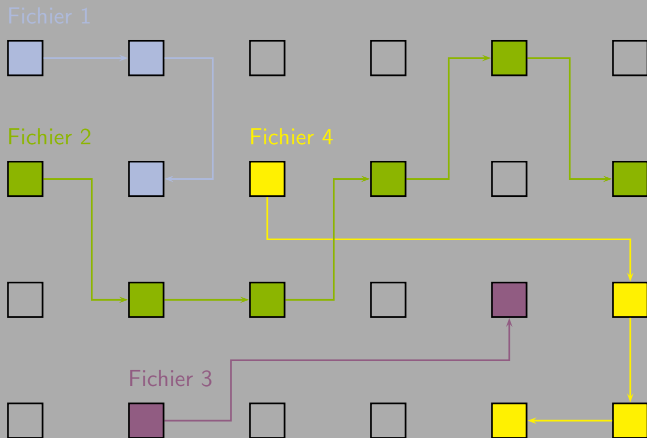
Allocation par blocs chaînés



Allocation par blocs chaînés

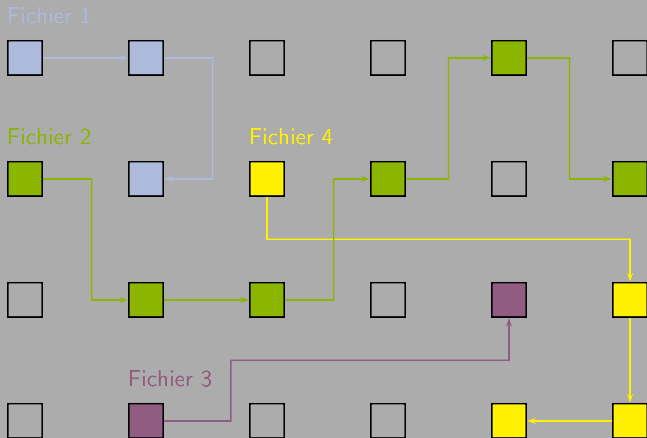


Allocation par blocs chaînés



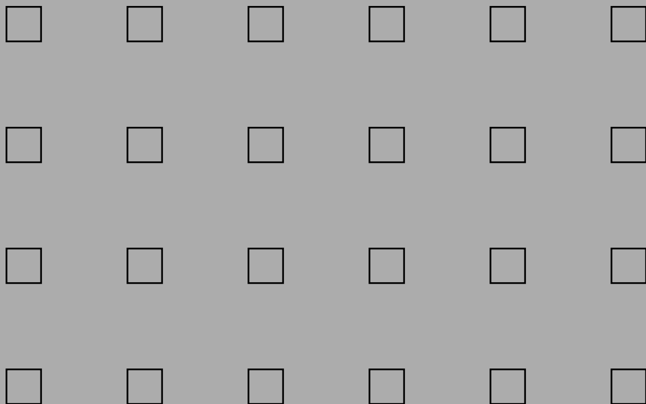
→ *Excellente gestion de l'espace disque*

Allocation par blocs chaînés

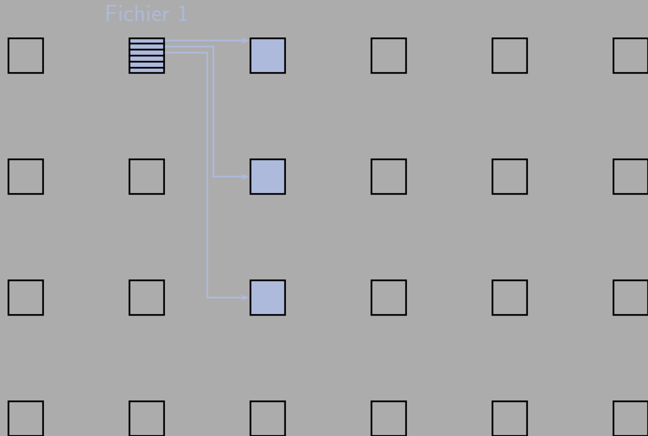


→ *Accès inefficace à un bloc quelconque...*

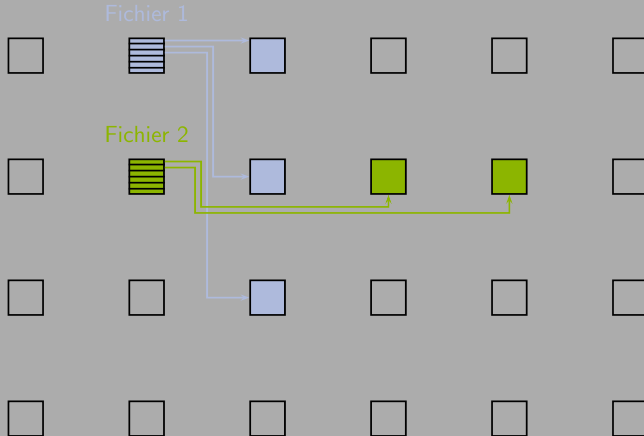
Allocation indexée



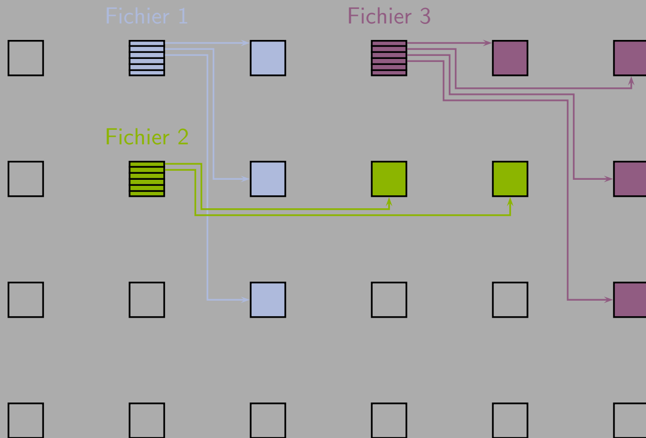
Allocation indexée



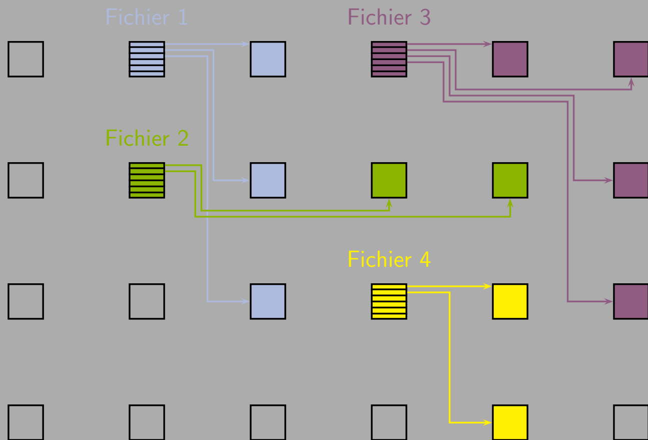
Allocation indexée



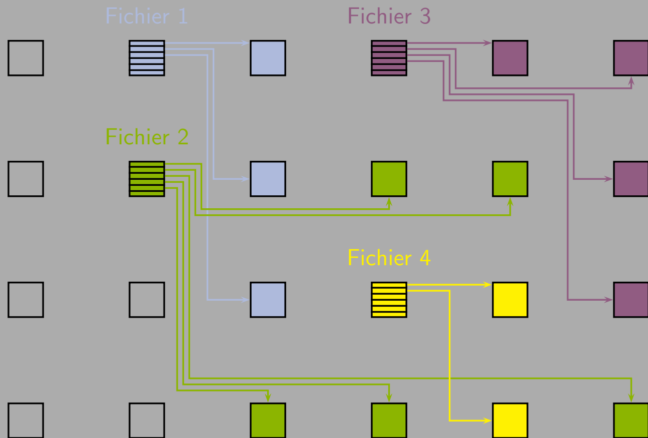
Allocation indexée



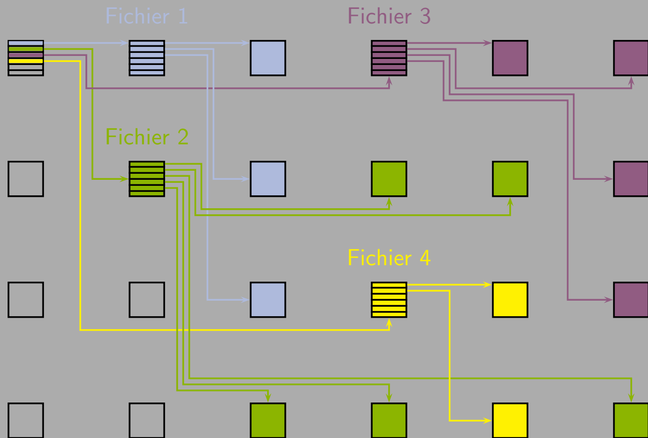
Allocation indexée



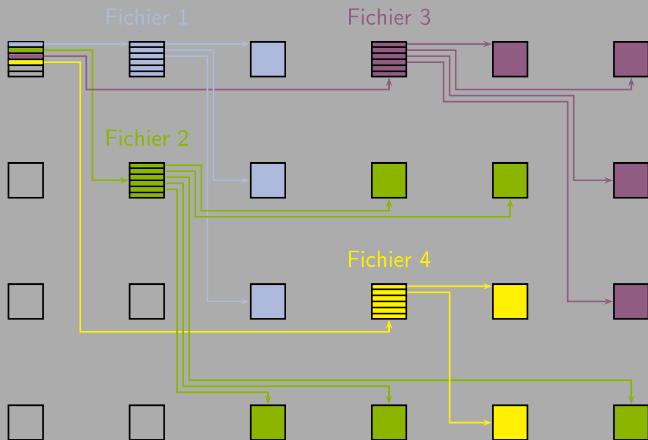
Allocation indexée



Allocation indexée

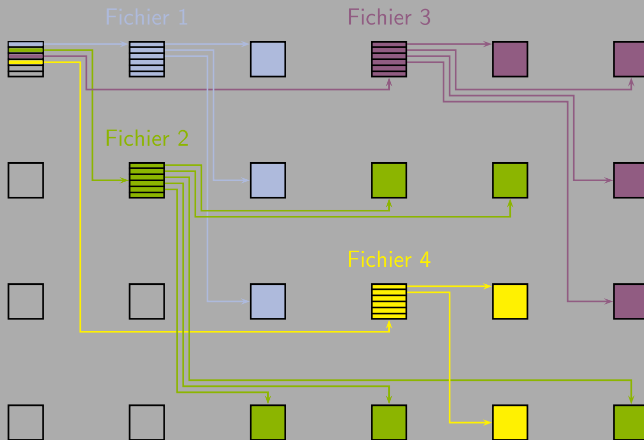


Allocation indexée



→ *Accès à un bloc quelconque en temps constant*

Allocation indexée



→ *Taille de fichier limitée par la taille d'index...*

Systèmes de fichiers UNIX

Systèmes de fichiers

On trouve dans Unix/Linux une grande variété de systèmes de fichiers, dont notamment Linux supporte la plupart :

- le format natif Linux *ext2*
- des systèmes Unix natifs, associés à Minix, System V, ou BSD
- les format Microsoft FAT, FAT32, et NTFS
- le format ISO 9660 associé aux CD
- le format Apple HFS
- les formats SUN NFS, IBM et Microsoft SMB, Novell NCP
- les systèmes journalisés ext3, ext4, Reiserfs, JFS, XFS, Btrfs...

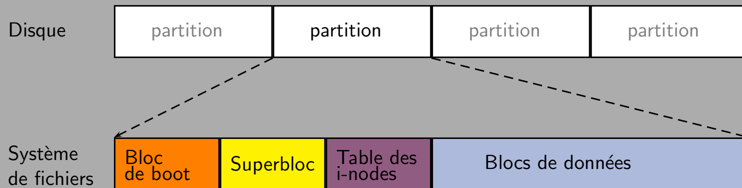
Virtual File System

- Chacun des systèmes de fichiers disponibles sous UNIX/LINUX diffère dans son implémentation (type d'allocation et organisation des blocs, des répertoires...).
- Le noyau LINUX résoud ce problème en proposant une interface générique, appelée *VFS*, qui permet de représenter l'ensemble des opérations de manipulation de fichier sous un format unique. Les primitives disponibles sont :

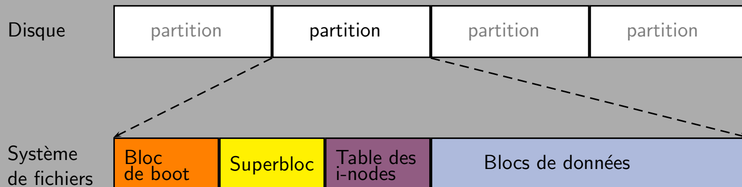
  `open()`, `read()`, `write()`, `lseek()`, `close()`, `truncate()`, `stat()`, `dup2()`, `mount()`, `umount()`, `mmap()`,`mkdir()`, `link()`, `unlink()`, `symlink()`, `rename()`

Organisation des disques

Organisation des disques



Organisation des disques



- Bloc de boot. Informations servant au boot
- Superbloc. Paramètres du système de fichiers (taille de la table des i-nodes, taille d'un bloc logique, taille du système de fichiers en blocs logique, . . .)
- Table des i-nodes. Une entrée par fichiers, contenant des informations diverses sur le fichier correspondant
- Blocs de données. Fichiers et répertoires

i-node

UNIX utilise une structure appelée *i-node* (pour *Index Node* au sein de laquelle est stockée l'information concernant un fichier.

L'I-Node

- d'un *fichier standard* contient des informations sur le positionnement des blocs correspondants sur le disque ;
- d'un *fichier spécial* contient les informations qui permettent l'identification du périphérique correspondant.

A tout i-node est associé un entier unique (à partir de 1) et chaque fichier a exactement un i-node.

Tous les i-nodes sont rangés dans la *table des i-nodes* rangée au début du disque.

i-node

UNIX utilise une structure appelée *i-node* (pour *Index Node* au sein de laquelle est stockée l'information concernant un fichier.

L'I-Node

- d'un *fichier standard* contient des informations sur le positionnement des blocs correspondants sur le disque ;
- d'un *fichier spécial* contient les informations qui permettent l'identification du périphérique correspondant.

A tout i-node est associé un entier unique (à partir de 1) et chaque fichier a exactement un i-node.

Tous les i-nodes sont rangés dans la *table des i-nodes* rangée au début du disque.

→ *Le numéro de l'i-node d'un fichier est accessible par ls -i*

Informations contenues dans un i-node

- type du fichier : standard, répertoire, bloc spécial, ...
- permissions associées au fichier
- id du propriétaire et du groupe
- un compteur de liens physiques
- les dates de dernière modification et d'accés
- dans le cas d'un fichier standard ou d'un répertoire, le positionnement des blocs
- dans le cas d'un fichier spécial, les numéros majeur et mineur associés au périphérique
- dans le cas d'un lien symbolique, la valeur de ce lien
- la table pointant vers les blocs alloués...

Informations contenues dans un i-node

- type du fichier : standard, répertoire, bloc spécial, ...
- permissions associées au fichier
- id du propriétaire et du groupe
- un compteur de liens physiques
- les dates de dernière modification et d'accés
- dans le cas d'un fichier standard ou d'un répertoire, le positionnement des blocs
- dans le cas d'un fichier spécial, les numéros majeur et mineur associés au périphérique
- dans le cas d'un lien symbolique, la valeur de ce lien
- la table pointant vers les blocs alloués...

→ Autrement dit, toutes les informations accessible par ls -l à l'exception du nom de fichier

Association i-nodes – blocs

- Les 12 premiers blocs d'un fichier sont directement pointés *via* la *table des blocs* contenue dans l'i-node
- Les trois entrées suivantes de la table des blocs pointent sur des tables de blocs, et génèrent une structure arborescente

Association i-nodes – blocs

- Les 12 premiers blocs d'un fichier sont directement pointés *via* la *table des blocs* contenue dans l'i-node
- Les trois entrées suivantes de la table des blocs pointent sur des tables de blocs, et génèrent une structure arborescente

→ *Extension de l'allocation indexée, autorisant une plage plus étendue pour la taille d'un fichier*

Table des blocs

Table des blocs
dans l'i-node

Blocs

Table des blocs

Table des blocs

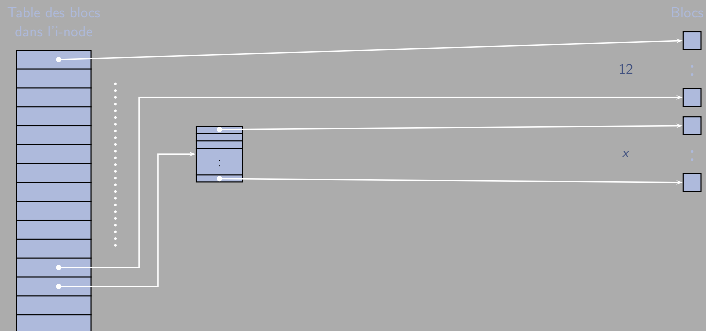


Table des blocs

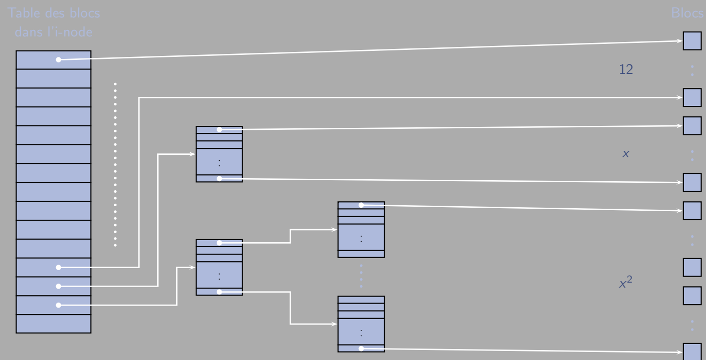
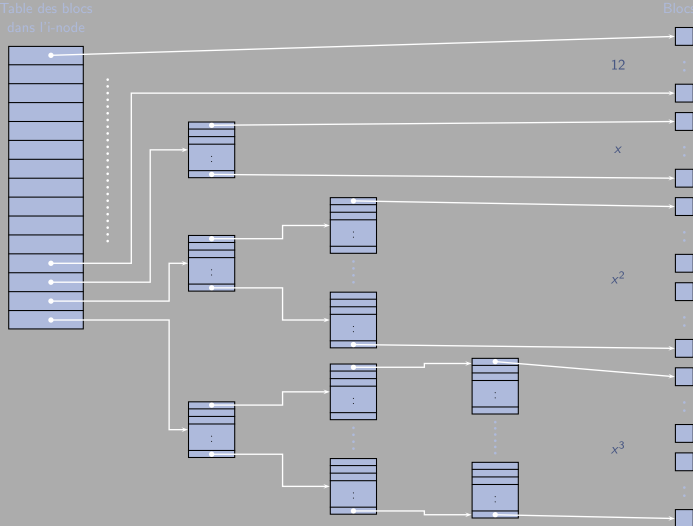


Table des blocs



Taille Max d'un fichier

Si une entrée de la table adresse x blocs ($x \geq 256$), alors

- les 12 premières entrées adressent chacune 1 bloc parmi x

Taille Max d'un fichier

Si une entrée de la table adresse x blocs ($x \geq 256$), alors

- les 12 premières entrées adressent chacune 1 bloc parmi x
- l'entrée 13 adresse une table de blocs dont chaque entrée adresse chacune 1 bloc, soient x blocs au total

Taille Max d'un fichier

Si une entrée de la table adresse x blocs ($x \geq 256$), alors

- les 12 premières entrées adressent chacune 1 bloc parmi x
- l'entrée 13 adresse une table de blocs dont chaque entrée adresse chacune 1 bloc, soient x blocs au total
- l'entrée 14 adresse une table de blocs dont chaque entrée adresse chacune une table de blocs dont chaque entrée adresse chacune un bloc, soient x^2 blocs au total

Taille Max d'un fichier

Si une entrée de la table adresse x blocs ($x \geq 256$), alors

- les 12 premières entrées adressent chacune 1 bloc parmi x
- l'entrée 13 adresse une table de blocs dont chaque entrée adresse chacune 1 bloc, soient x blocs au total
- l'entrée 14 adresse une table de blocs dont chaque entrée adresse chacune une table de blocs dont chaque entrée adresse chacune un bloc, soient x^2 blocs au total
- l'entrée 15 ajoute une indirection supplémentaire, soient x^3 blocs au total

Taille Max d'un fichier

Si une entrée de la table adresse x blocs ($x \geq 256$), alors

- les 12 premières entrées adressent chacune 1 bloc parmi x
- l'entrée 13 adresse une table de blocs dont chaque entrée adresse chacune 1 bloc, soient x blocs au total
- l'entrée 14 adresse une table de blocs dont chaque entrée adresse chacune une table de blocs dont chaque entrée adresse chacune un bloc, soient x^2 blocs au total
- l'entrée 15 ajoute une indirection supplémentaire, soient x^3 blocs au total

→ *On adresse ainsi* $12 + x + x^2 + x^3$ *blocs distincts*

Taille Max d'un fichier

En pratique :

- avec une taille de bloc de $T = 4096$ octets, et un adressage sur 32 bits, (4 octets), chaque bloc peut adresser $T/4 = 1024$ octets distincts

Taille Max d'un fichier

En pratique :

- avec une taille de bloc de $T = 4096$ octets, et un adressage sur 32 bits, (4 octets), chaque bloc peut adresser $T/4 = 1024$ octets distincts
- la taille maximale d'un fichiers est alors

$$(12 + 1024 + 1024^2 + 1024^3) \text{ octets} \approx 4 \text{ To}$$

Taille Max d'un fichier

En pratique :

- avec une taille de bloc de $T = 4096$ octets, et un adressage sur 32 bits, (4 octets), chaque bloc peut adresser $T/4 = 1024$ octets distincts
- la taille maximale d'un fichiers est alors

$$(12 + 1024 + 1024^2 + 1024^3) \text{ octets} \approx 4 \text{ To}$$

Remarque : Cette organisation très souple permet de manipuler aussi facilement des petits fichiers que des très gros...

Primitives d'accès et de manipulation des fichiers

Accés aux informations d'un i-node

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *CheminFic, struct stat *Buf);
int fstat(int fd, struct stat *Buf);
int lstat(const char *CheminFic, struct stat *Buf);
```

→ *renvoie -1 en erreur avec positionnement de errno*

- le système récupère le statut du fichier pointé par `CheminFic` et remplit les champs d'une variable de type `struct stat`
- `fstat()` identique à `stat()` mais en passant un descripteur de fichier au lieu du nom ; `lstat()` pour les liens symboliques
- peut être appelée sans les droits d'accès à la cible, mais il faut les droits d'ouverture sur tous les répertoires intermédiaires y conduisant

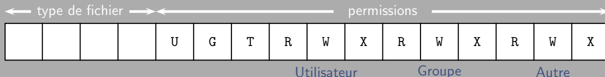
Cf. *man 2 stat*

La struct stat

```
struct stat {
  dev_t      st_dev;     // device
  ino_t      st_ino;     // inode
  mode_t     st_mode;    // type fichier et permissions
  nlink_t    st_nlink;   // nb de liens
  uid_t      st_uid;     // du prop
  gid_t      st_gid      // du groupe
  dev_t      st_rdev     // type device
  off_t      st_size     // taille
  blksize_t  st_blksize  // taille pour E/S
  blkcnt_t   st_blocks   // nb blocs alloues
  time_t     st_atime    // dernier acces
  time_t     st_mtime    // derniere modif
  time_t     st_ctime    // dernier chgt etat
}
```

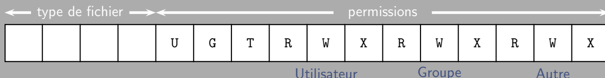
Le champs st_mode de la struct stat

Le champs st_mode prend la forme d'un masque permettant l'identification du type de fichier et des permissions associées.



Le champs st_mode de la struct stat

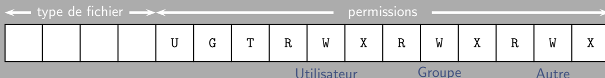
Le champs `st_mode` prend la forme d'un masque permettant l'identification du type de fichier et des permissions associées.



- 9 bits de poids faible pour les permissions *Utilisateur*, *Groupe*, et *Autre*

Le champs st_mode de la struct stat

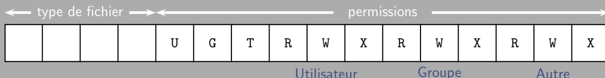
Le champs st_mode prend la forme d'un masque permettant l'identification du type de fichier et des permissions associées.



- 9 bits de poids faible pour les permissions *Utilisateur*, *Groupe*, et *Autre*
- 3 bits additionnels : les bits *set-user-ID* et *set-group-ID* permettent de surclasser les privilèges d'un processus ; le *sticky bit* permet à un processus non-privilégié d'agir sur les fichiers dont il est propriétaire dans un répertoire sur lesquels il a les droits en écriture (par exemple /temp)

Le champs st_mode de la struct stat

Le champs `st_mode` prend la forme d'un masque permettant l'identification du type de fichier et des permissions associées.



- 9 bits de poids faible pour les permissions *Utilisateur*, *Groupe*, et *Autre*
- 3 bits additionnels : les bits *set-user-ID* et *set-group-ID* permettent de surclasser les privilèges d'un processus ; le *sticky bit* permet à un processus non-privilégié d'agir sur les fichiers dont il est propriétaire dans un répertoire sur lesquels il a les droits en écriture (par exemple `/temp`)
- 4 bits (sur Linux) pour identifier le type de fichier associé

Type de fichier dans st_mode

Le type du fichier peut être examiné de deux façons.

- En testant le résultat obtenu à partir d'un '&' logique avec la constante S_IFMT sur des constantes disponibles ; par exemple pour un fichier « régulier » :

```
if (BufferStat.st_mode & S_IFMT) == S_IFREG)
  cout << "Fichier standard";
```

- S'agissant d'opérations usuelles de test, des macros parenthésées sont aussi fournies :

```
if( S_ISREG(BufferStat.st_mode) )
  cout << "Fichier standard";
```

Type de fichier dans st_mode

Constante	Macro	Type de fichier associé
S_IFREG	S_ISREG()	Fichier standard
S_IFDIR	S_ISDIR()	Répertoire
S_IFCHR	S_ISCHR()	Dispositif à base de caractères
S_IFBLK	S_ISBLK()	Dispositif à base de blocs
S_IFIFO	S_ISFIFO()	FIFO ou pipe
S_IFSOCK	S_ISSOCK()	Socket
S_IFLNK	S_ISLNK()	Lien symbolique

Remarque : Les tests sur S_IFLNK ou S_ISLNK() n'ont de sens pour un appel à la variante lstat(), cf. *man 2 stat*...

Masques de permissions dans `st_mode`

Constante	Valeur octale	Bit de permission
`S_ISUID`	04000	Set-User-ID
`S_ISGID`	02000	Set-Group-ID
`S_ISVTX`	01000	Sticky
`S_IRUSR`	0400	User-read
`S_IWUSR`	0200	User-write
`S_IXUSR`	0100	User-execute
`S_IRGRP`	040	Group-read
`S_IWGRP`	020	Group-write
`S_IXGRP`	010	Group-execute
`S_IOGRP`	04	Other-read
`S_IOGRP`	02	Other-write
`S_IOGRP`	01	Other-execute
`S_IRWXU`	0700	User-rwx
`S_IRWXG`	070	Group-rwx
`S_IRWXO`	07	Other-rwx

Création et manipulation de fichiers

Rappel : au sens Unix, tout flux de données est un fichier

- On dispose de primitives système pour créer et ouvrir, lire, écrire, fermer, détruire un fichier
- Tout fichier ouvert est identifié par un entier positif, appelé *descripteur de fichier* (i.e. *fd* pour *file descriptor*)
- Chaque processus dispose d'une *table de descripteurs des fichiers* qui lui sont associés : un fichier ouvert plusieurs fois sera identifié par autant de descripteurs

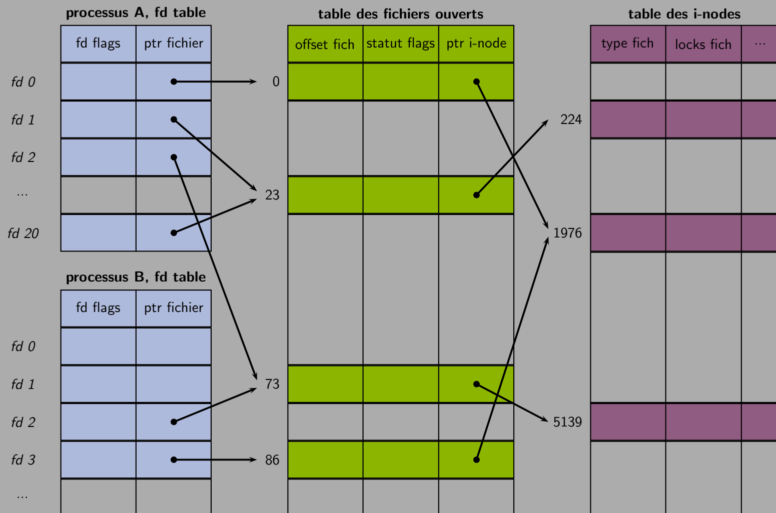
Descripteurs standard

fd	fonction	constante POSIX	flux *stdio*
0	entrée standard	STDIN_FILENO	*stdin*
1	sortie standard	STDOUT_FILENO	*stdout*
2	erreur standard	STDERR_FILENO	*stderr*

- En mode normal, ces trois descripteurs sont toujours ouvert dans le shell
- Les processus héritent de copies des descripteurs du shell
- Dans un shell interactif, les trois descripteurs font référence au terminal sous lequel tourne le shell

$\rightarrow$ *Les descripteurs suivants sont alloués à chaque ouverture de fichier à partir du plus petit entier disponible*

Relation entre descripteurs et i-nodes



Relation entre descripteurs et i-nodes : commentaires

On constate :

- les *fd* 1 et 20 du seul processus A font référence au même fichier dans la table des fichiers ouverts
- le *fd* 2 du processus A et le *fd* 2 du processus B, distincts, font référence au même fichier dans la table des fichiers ouverts
- le *fd* 0 du processus A et le *fd* 3 du processus B font référence à deux éléments distincts *a priori* dans la table des fichiers ouverts, mais qui pointent vers le même fichier

Relation entre descripteurs et i-nodes : commentaires

On constate :

- les *fd* 1 et 20 du seul processus A font référence au même fichier dans la table des fichiers ouverts [résulte de `dup()`, `dup2()`, ou `fcntl()`]
- le *fd* 2 du processus A et le *fd* 2 du processus B, distincts, font référence au même fichier dans la table des fichiers ouverts [résulte de `fork()` ou du passage d'un descripteur ouvert d'un processus à l'autre *via* un socket]
- le *fd* 0 du processus A et le *fd* 3 du processus B font référence à deux éléments distincts *a priori* dans la table des fichiers ouverts, mais qui pointent vers le même fichier [résulte d'appels indépendants de chaque processus à `open()` sur le même fichier]

Primitive d'ouverture open()

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *chemin, int flags);
int open(const char *chemin, int flags, mode_t mode);
```

→ *renvoie le plus petit descripteur disponible si succès ou -1 en erreur*

- `flags`
  - type d'ouverture (création, troncature, ...)
  - type d'opération sous-jacente (lecture, écriture)
  - comportement (écriture immédiate ou différée, ...)
- `mode`
  - permissions sur le fichier données en octal, pertinent seulement avec `OCREAT` dans les flags

Drapeaux associés à l'ouverture

flags...	Propos
O_RDONLY	Lecture uniquement
O_WRONLY	Ecriture uniquement
O_RDWR	Lecture et/ou écriture
O_CREAT	Forcément en écriture, crée le fichier lors de l'ouverture s'il n'existe pas
O_EXCL	Avec O_CREAT, crée le fichier s'il n'existe pas sinon renvoie l'erreur EEXIST
O_TRUNC	Tronque le fichier à la longueur 0
O_APPEND	Les opérations d'écriture sont effectuées en fin de fichier
O_SYNC	Ecritures synchrones

→ *Les flags peuvent être combinés avec l'opérateur binaire | (OU logique) équivalent à l'union ensembliste, par exemple :*

O_CREAT | O_EXCL

→ *ATTENTION :* O_RDONLY | O_WRONLY *n'est (évidemment) pas identique à* O_RDWR *et renvoie une erreur*

Exemple : copie entre fichiers

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

int main {
  const int fdSource = Open("FicSource.txt",
                            O_RDONLY);
  const int fdDest = Open("FicDest.txt",
                            O_WRONLY|O_CREAT, 0700);
...
}
```

→ *Les noms de fichiers sont des NTCS, attention au transtypage entre* `char*` *et* `string`

Primitive de lecture read()

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t NbALire);
```

→ *renvoie le nb d'octets lus ou -1 en erreur*

- tente de lire NbALire octets à partir du fichier pointé par fd
- et remplit avec la zone mémoire pointée par buf

→ *La zone mémoire* buf *doit avoir été préalablement allouée, et* fd *est un descripteur préalablement ouvert en lecture*

Primitive d'écriture write()

```
#include <unistd.h>

ssize_t write(int fd, void *buf, size_t NbAEcrire);
```

→ *renvoie le nb d'octets écrits ou -1 en erreur*

- tente d'écrire `NbALire` octets vers le fichier pointé par `fd`
- à partir de la zone mémoire pointée par `buf`

→ *La zone mémoire* `buf` *doit avoir été préalablement allouée, et* `fd` *est un descripteur préalablement ouvert en écriture*

Primitive de fermeture `close()`

```
#include <unistd.h>

int close (int fd);
```

→ *renvoie 0 si succès ou -1 en erreur*

- ferme le descripteur `fd`, réutilisable lors de prochains appels à `open()`
- sans `O_SYNC`, l'écriture passe par un tampon : le retour avec succès de l'appel à `close` n'est pas garant d'une écriture effective dans le fichier

Exemple complet : copie entre fichiers

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

int main {
  const int fdSource = Open("FicSource.txt",
                            O_RDONLY);
  const int fdDest = Open("FicDest.txt",
                            O_WRONLY|O_CREAT, 0700);
  const size_t NbBytes=512; char Tampon[NbBytes];
  size_t NbLus=0;
  while ((NbLus=Read(fdSource, Tampon, NbBytes))>0){
    Write(fdDest, Tampon, NbLus);
  }
  Close(fdSource); Close (fdDest);
  return 0;
}
```

Primitive de positionnement lseek()

```
#include <sys/types.h>
#include <unistd.h>

int lseek(int fd, off_t offset, int direction);
```

→ *renvoie la position relative au début de fichier ou -1 en erreur*

- fixe la position courante d'E/S dans le fichier pointé par `fd`
- trois valeurs pour `direction` :
  - `SEEK_SET` exactement `offset`
  - `SEEK_CUR` position actuelle + `offset`
  - `SEEK_END` taille du fichier + `offset`
- `off_t` est un sous-type de `long int`
- un positionnement au delà de la fin de fichier sans changer explicitement sa taille forcera l'écriture et la création de trous

Manipulation des drapeaux d'un fichier : fcntl()

```
#include <sys/types.h>
#include <fcntl.h>

int fcntl(int fd, int cmd); // operation
int fcntl(int fd, int cmd, long arg); // parametres
```

→ *renvoie 0 si succès ou -1 en erreur*

- `cmd` pour la manipulation des drapeaux du fichier `fd` :
  - `F_GETFL` pour l'obtention en valeur de retour
  - `F_SETFL` pour changement en fonction de `arg` :
    - `O_APPEND`
    - `O_DIRECT` minimisation des tampons système
    - `O_NOATIME` non-chgt temps accès `read()`
    - `O_NONBLOCK`, `O_ASYNC` pour terminaux, sockets, ...

→ *LA fonction servant de « couteau suisse »...*