h e p i a

Haute école du paysage, d'ingénierie
et d'architecture de Genève

Systèmes d'exploitation

Daniele Raffo

HEPIA 2025-2026

(slides: Guillaume Chanel, Jean-Luc Falcone, Florent Glück, Mickaël Hoerdt, Daniele Raffo)

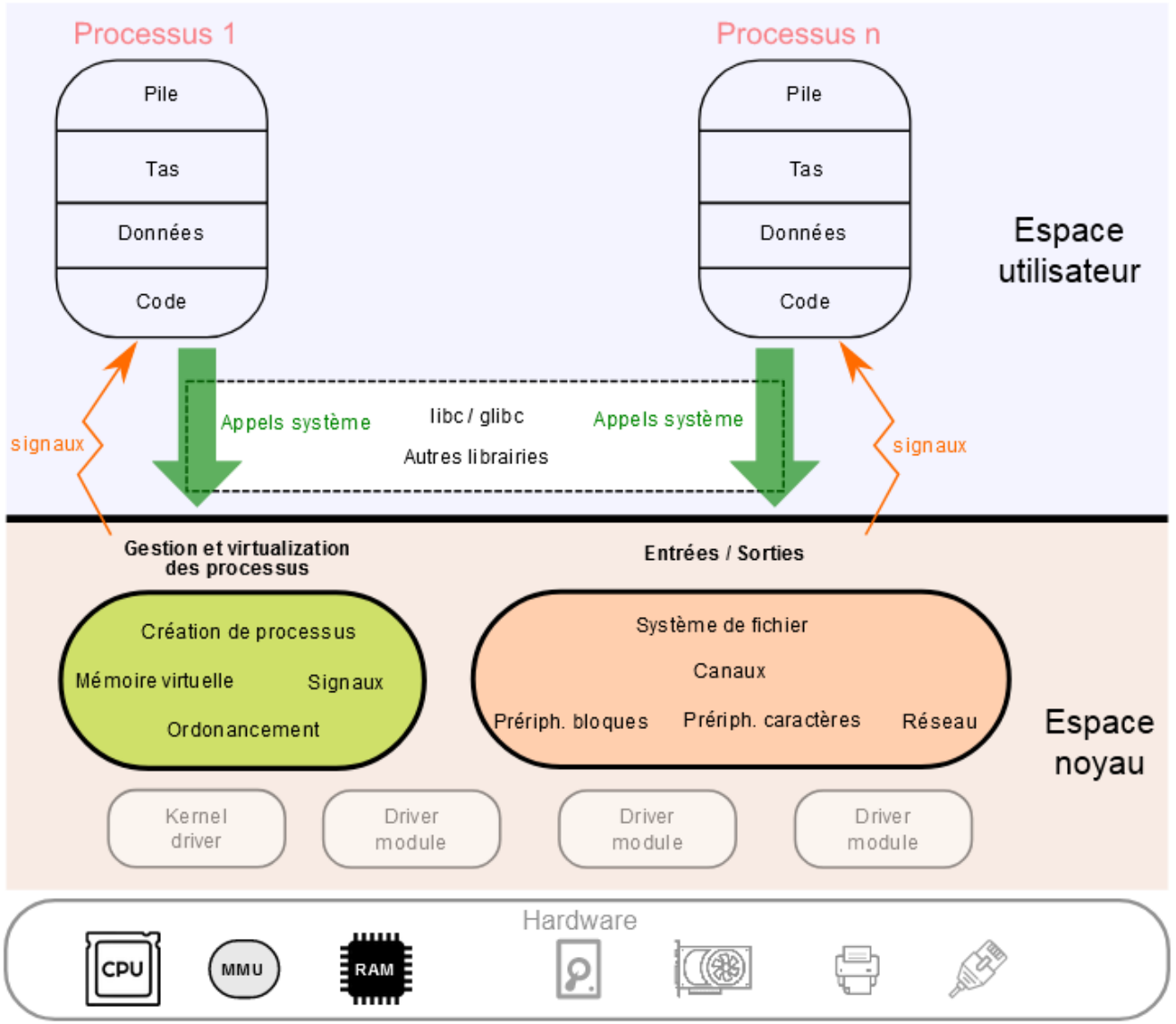
Qu'est-ce qu’un système d'exploitation?

Un système d'exploitation (*Operating System, OS*) offre aux programmes des services - par exemple une API - leur permettant de facilement exploiter les ressources disponibles.

Exemples de ressources exploitées par le système:

- la mémoire vive (RAM)
- la mémoire de masse (disque)
- les processeurs
- les périphériques

Vue générale



Objectifs

Objectifs théoriques

- Décrire ce qu'est un processus
- Résumer les mécanismes de mémoire virtuelle
- Lister les moyens de communication entre processus
- Définir le rôle d'un module

Objectifs pratiques

- Coder en C, analyser et comprendre un code existant
- Identifier un appel système
- Créer des processus
- Communiquer par réseau, signaux et mémoires partagées
- Implémenter un module

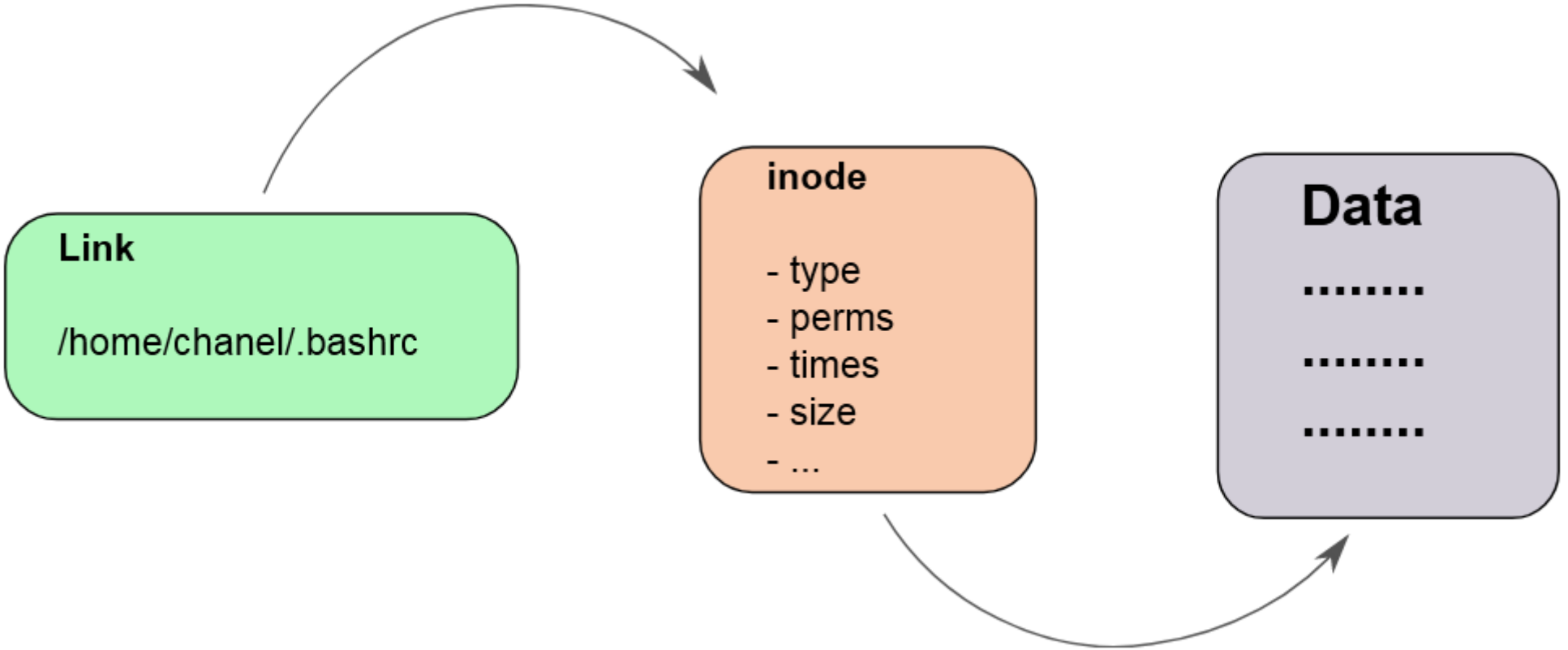
Fichiers (*files*) et répertoires (*directories*)

Inodes

Un inode est une structure de données contenant des information sur un "fichier" (fichier, directory, socket, device, pipe, etc.).
Il ne contient pas le nom du fichier; celui-ci est stocké dans le structure d'informations du répertoire contenant le fichier.
Il contient généralement (POSIX) des informations sur:

- Numéro d'inode
- Périphérique contenant le fichier (device ID)
- Propriétaire et groupe, permissions, temps d'accès/modification
- Taille du fichier
- Nombre de liens pointant vers l'inode
- **Pointeurs vers les données**

Inodes



Inodes

Un inode contient trois temps différents:

- `atime` date du dernier accès à l'inode (ou aux données)
- `mtime` date de la dernière modification des données
- `ctime` date de la dernière modifications des méta-données
- `crtime` date de création du fichier (non-POSIX)

Pour gagner en performance, il est possible de désactiver la mise à jour de `atime` lorsque la partition est montée:

```
mount /dev/sda1 /mnt/mypart -o noatime
```

Inodes

La commande `stat` permet d'afficher des données sur un inode.

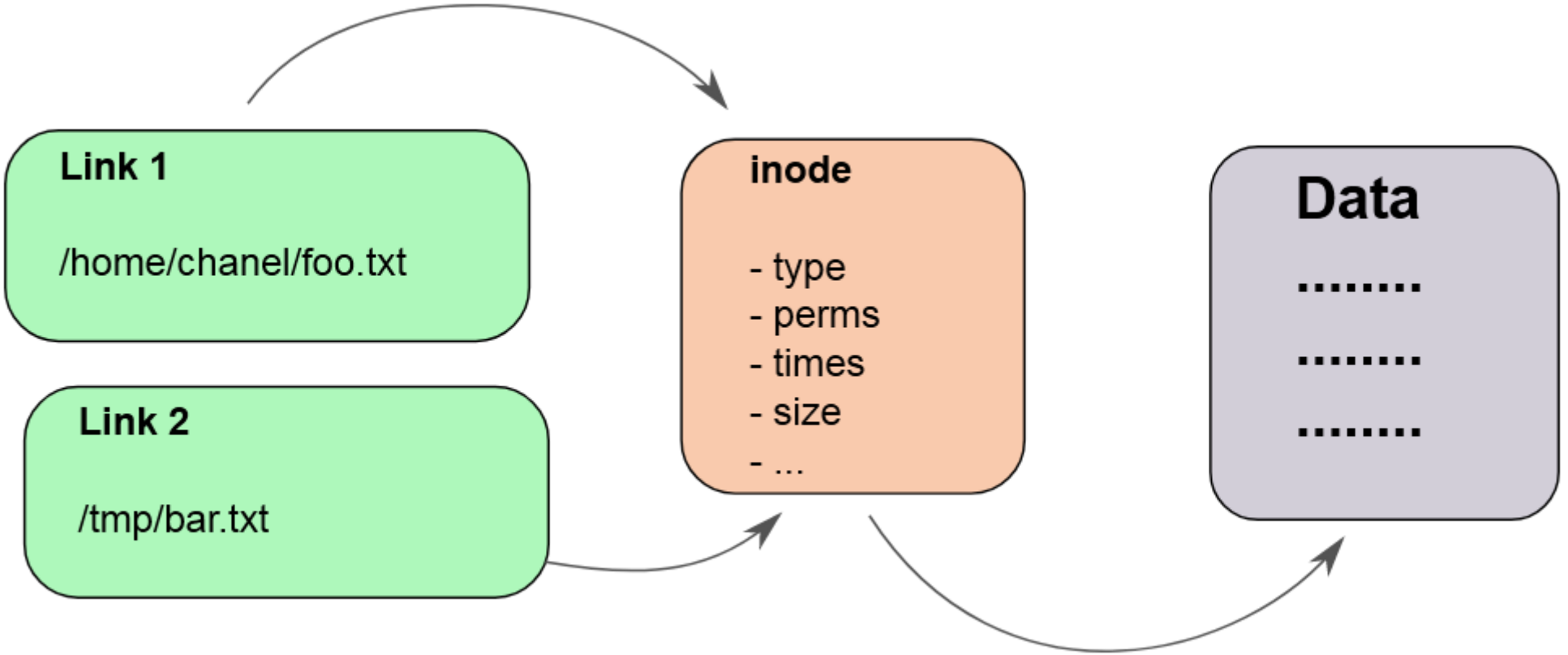
```
$ touch /tmp/myfile  # met à jour les dates (crée un fichier si inexistant)
$ stat /tmp/myfile
  File: /tmp/myfile
  Size: 0               Blocks: 0          IO Block: 4096   regular empty file
Device: 2fh/47d Inode: 422766      Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/  chanel)   Gid: ( 1000/  chanel)
Access: 2019-07-19 16:56:35.540113838 +0200
Modify: 2019-07-19 16:56:35.540113838 +0200
Change: 2019-07-19 16:56:35.540113838 +0200
 Birth: -
```

Hard link

Les entrées des répertoires sont des **liens durs (hard links)** pointant vers des inodes.

On peut créer plusieurs liens vers un fichier.
Un fichier est "effacé" lorsqu'il n'y a plus de liens pointant sur son inode (sauf si un processus maintient le fichier ouvert): commandes `rm` et `unlink`.

Hard link


Link 1
/home/chanel/foo.txt
Link 2
/tmp/bar.txt
inode
- type
- perms
- times
- size
- ...
Data
.....
.....
.....

Hard link

`ls -i` montre le numéro d'inode pour chaque fichier.

`ls -l` montre le link count pour chaque fichier.

On peut créer un hard link avec la commande `ln`:

```
$ ln /tmp/myfile /tmp/newlink
$ stat /tmp/myfile
  File: /tmp/myfile
  Size: 0               Blocks: 0          IO Block: 4096   regular empty file
Device: 2fh/47d Inode: 422766      Links: 2
Access: (0644/-rw-r--r--)  Uid: ( 1000/  chanel)   Gid: ( 1000/  chanel)
Access: 2019-07-19 16:56:35.540113838 +0200
Modify: 2019-07-19 16:56:35.540113838 +0200
Change: 2019-07-19 16:56:35.540113838 +0200
 Birth: -
```

Hard link

Un répertoire ne peux posséder qu'un seul hard link, afin d'éviter les cycles: la structure du système de fichier doit rester acyclique!

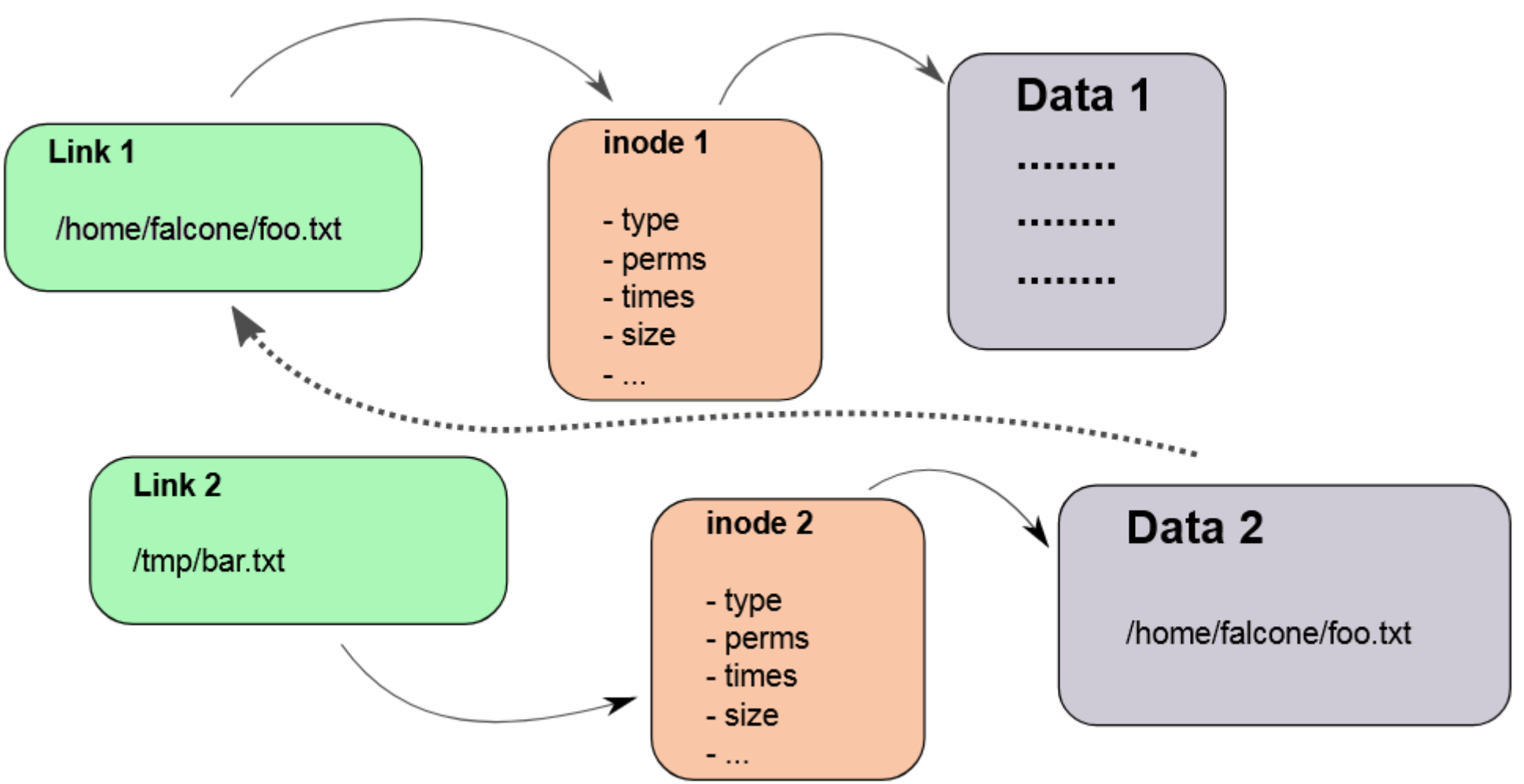
Tous les hard links pointant sur un inode doivent se trouver sur le même système de fichier que cet inode.

Lien symbolique

Un lien **symbolique (symlink)** possède son propre inode qui pointe vers un nom.
Contrairement aux hard links, on peut créer des symlinks vers un répertoire, ou vers un fichier/répertoire sur un autre système de fichier.
On crée un symlink avec la commande `ln` en utilisant l'option `-s`:

```
$ ln -s /tmp/myfile /tmp/newlink2
$ stat /tmp/newlink2  # comme indiqué ci-dessous on a bien à faire à un autre inode
  File: /tmp/newlink2 -> /tmp/myfile
    Size: 11              Blocks: 0          IO Block: 4096   symbolic link
  Device: 2fh/47d Inode: 563876      Links: 1
  Access: (0777/lrwxrwxrwx)  Uid: ( 1000/  chanel)   Gid: ( 1000/  chanel)
  Access: 2019-07-19 18:16:26.343945684 +0200
  Modify: 2019-07-19 18:16:13.240259297 +0200
  Change: 2019-07-19 18:16:13.240259297 +0200
   Birth: -
```

Lien symbolique



Hard link vs Lien symbolique

	Hard link	Symbolic link
Definition	A link to an already existing inode	A path to a filename; a shortcut
Command to create it	`ln` *`file hardlink`*	`ln -s` *`file symlink`*
Link is still valid if the original file is moved or deleted	Yes (because the link still references the inode to which the original file pointed)	No (because the path now references a non-existent file)
Can link to a file in another filesystem	No (because inode numbers make sense only within a determinate filesystem)	Yes
Can link to a directory	No	Yes
Link permissions	Reflect the original file's permissions, even when these are changed	`rwxrwxrwx`
Link attributes	`-` (regular file)	`l` (symbolic link)
Inode number	The same as the original file	A different inode number (since it's a different file)

link(), symlink()

Créer un hard link

```
int link(const char *oldpath, const char *newpath);
```

oldpath est le nom existant et newpath le nouveau nom.

Retourne 0 si succès et -1 si erreur.

Si newpath existe, code d'erreur EEXIST (cf. `errno`).

Créer un lien symbolique

```
int symlink(const char *oldpath, const char *newpath);
```

Utilisation identique à `link()`.

unlink()

Effacer le nom (lien) d'un fichier

```
int unlink(const char *pathname);
```

pathname est le nom à supprimer.

Retourne 0 en cas de succès et -1 en cas d'erreur.

Voir `man 2 unlink` pour les codes d'erreurs.

Fonctionne avec les liens durs ou symboliques.

Si le lien est le dernier à pointer sur un inode:

- si aucun processus n'a ouvert le fichier, l'inode est supprimé immédiatement
- sinon, l'inode sera supprimé lorsque le dernier processus le ferme

/lost+found

A la suite d'une erreur du système de fichier (p.ex. suite à une mise hors tension brutale), un inode peut se retrouver sans lien.

Lors d'un contrôle de fichier (`fsck`), il sera copié dans le répertoire `/lost+found` , à la racine du système de fichier.

Structure stat (sys/stat.h)

```
struct stat{
  dev_t     st_dev;     //device ID
  ino_t     st_ino;     //i-node number
  mode_t    st_mode;    //protection and type
  nlink_t   st_nlink;   //number of hard links
  uid_t     st_uid;     //user ID of owner
  gid_t     st_gid;     //group ID of owner
  dev_t     st_rdev;    //device type (if special file)
  off_t     st_size;    //total size, in bytes
  blksize_t st_blksize; //blocksize for filesystem I/O
  blkcnt_t  st_blocks;  //number of 512B blocks
  time_t    st_atime;   //time of last access
  time_t    st_mtime;   //time of last modification
  time_t    st_ctime;   //time of last change
};
```

stat()

L'appel système `stat()` permet de garnir une structure stat:

```
int stat(const char *path, struct stat *buf);
```

La fonction retourne 0 si tout s'est bien passé ou -1 en cas d'erreur.

```
struct stat infos;
char *filename = "/tmp/foo.txt";
if(stat(filename, &infos) < 0)
   fprintf(stderr, "Cannot stat %s: %s\n", filename, strerror(errno));
else
   printf("Filesize: %d\n", infos.st_size);
```

Type de inode

Le champ `st_mode` est un champ de bits contenant les permissions et le type d'un inode.
Il existe plusieurs macro POSIX permettant de tester les types:

`S_ISREG(m)`	fichier de données ?
`S_ISDIR(m)`	répertoire ?
`S_ISCHR(m)`	character device ?
`S_ISBLK(m)`	block device ?
`S_ISFIFO(m)`	FIFO (named pipe) ?
`S_ISLNK(m)`	lien symbolique ?
`S_ISSOCK(m)`	socket ?

```
if(S_ISDIR(info.st_mode)) {
   printf("L'inode est un repertoire.\n");
}
```

Permissions d'un inode

On peut utiliser plusieurs flags pour accéder aux valeurs du champs de bits:

`S_IRUSR`	`00400`	user (proprietaire) has read permission
`S_IWUSR`	`00200`	user (proprietaire) has write permission
`S_IXUSR`	`00100`	user (proprietaire) has execute permission
`S_IRGRP`	`00040`	group has read permission
`S_IWGRP`	`00020`	group has write permission
`S_IXGRP`	`00010`	group has execute permission
`S_IROTH`	`00004`	others have read permission
`S_IWOTH`	`00002`	others have write permission
`S_IXOTH`	`00001`	others have execute permission

lstat()

Si le fichier A est un lien symbolique vers B, `stat("A", ...)` retourne les informations sur l'inode de B.
On peut éviter ce comportement et obtenir les informations sur le lien symbolique lui-même grâce à `lstat()`:

```
int lstat(const char *path, struct stat *buf);
```

Le reste du comportement est identique à `stat()`.

fstat()

Pour connaitre les informations sur un fichier déjà ouvert, il faut utiliser l'appel système `fstat()`.
Il fonctionne comme `stat()` mais permet d'utiliser un file descriptor (descripteur de fichier) à la place d'un nom:

```
int fstat(int fd, struct stat *buf);
```

access()

On peut tester si le processus en cours à le droit de lire/écrire/exécuter un fichier grâce à l'appel système `access()`:

```
int access(const char *pathname, int mode);
```

Le paramètre mode est un champs de bits formé des flags:

`R_OK` lecture possible
`W_OK` écriture possible
`X_OK` éxécution possible

On peut aussi tester le flag `F_OK` qui indique si le fichier existe.
Le test se fait en fonction de l'utilisateur/groupe courant.
`access()` retourne 0 si le test réussit, -1 sinon.

access()

```
char *fn = "/tmp/foo.txt";
if (access(fn, R_OK|W_OK) == 0)
   printf("On peut lire et ecrire sur %s\n", fn);
else if (errno == EACCES)
   printf("Pas droit de lire et/ou ecrire sur %s\n", fn);
else
   perror(fn);
```

chmod(), fchmod()

On peut changer les permissions d'un fichier grâce à l'appel système `chmod()`, similaire à la commande shell homonyme.

Via un nom de fichier:

```
int chmod(const char *path, mode_t mode);
```

Via un file descriptor:

```
int fchmod(int fd, mode_t mode);
```

Le paramètre mode est un champs de bits formé des mêmes flags que le champs `st_mode` de la structure stat.

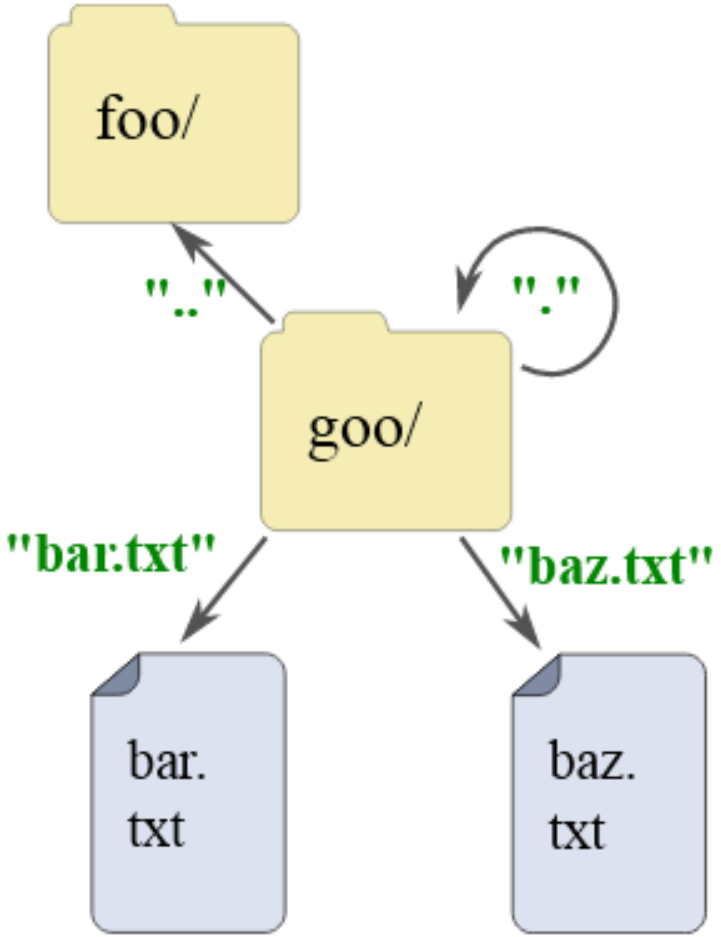
Répertoires (directories)

Les **répertoires (directories)** représentent l'organisation des fichiers sous forme d'arborescence.

Ce sont des inodes dont le contenu est une liste d'entrées (structure `dirent`, voir aussi le header `dirent.h`) associants un lien à un inode.

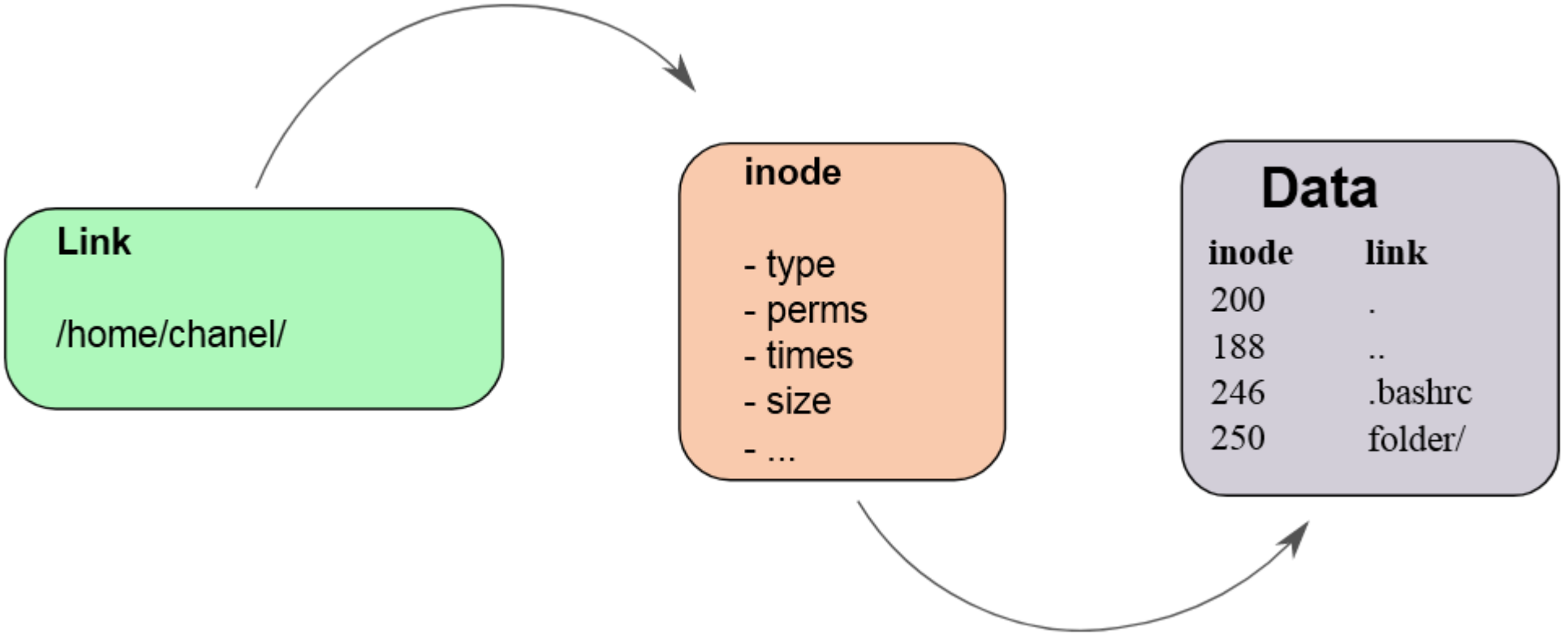
Chaque liste d'entrée contient au moins `.` (le répertoire lui-même) et `..` (le répertoire parent).

Répertoires (directories)




```
foo/
foo/goo/
foo/goo/bar.txt
foo/goo/baz.txt
```

Inodes des répertoires



Répertoire courant

Un processus possède un **répertoire courant (current working directory)** qui permet d'interpréter les chemins relatifs, p.ex. `src/monfichier.c`.

Au démarrage du programme ce répertoire est celui depuis lequel le programme est lancé. Ce n'est donc pas forcément le répertoire de l'exécutable.

Répertoire courant

Pour connaitre le répertoire courant:

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

buf: chaine de charactères (déclarée par l'utilisateur) qui contiendra le répertoire courant.
size: taille du buffer.

Retourne NULL en cas d'erreur, l'adresse de buf sinon.

Répertoire courant

Pour changer le répertoire courant:

```
#include <unistd.h>
int chdir(const char *path);
```

path: chaine de caractères indiquant le nouveau répertoire (relatif ou absolu).

Retourne -1 en case d'erreur, 0 sinon.

Structure dirent

Les entrées d'un répertoire sont représentées par la structure:

```
struct dirent {                 /* dirent.h */
    ino_t           d_ino;      /* inode number */
    off_t           d_off;      /* opaque value used to get next dirent (do not use) */
    unsigned short  d_reclen;   /* length of this record */
    unsigned char   d_type;     /* type of file; not supported by all file systems */
    char            d_name[256]; /* filename (NULL terminated), sometimes d_name[0] */
};
```

Seulement deux champs sont décrit par POSIX: d_ino et d_name.

Ne jamais compter sur la taille du tableau d_name, uniquement sur la constante MAX_NAME, qui indique la longueur maximale des noms d'entrées, ou sur strlen.

Structure dirent

Le champs d_type est un champs de bits contenant des informations sur le type de l'inode associé:

`DT_DIR`	Répertoire
`DT_LNK`	Lien symbolique
`DT_REG`	Fichier de données
`DT_UNKNOWN`	Type inconnu
`DT_...`	Voir "man readdir" pour tous les types

Même sous GNU/Linux, tous les systèmes de fichiers ne donnent pas un accès au type par la structure dirent. Dans ce cas, le d_type est toujours égal à DT_UNKNOWN.

Accéder aux entrées d'un répertoire

Pour accéder aux entrées d'un répertoire, il faut:

1) "Ouvrir" le répertoire avec `opendir()`
2) "Lire" l'entrée suivante avec `readdir()`
3) Répéter 2, jusqu'à épuisement des entrées ou tout autre critère
4) "Fermer" le répertoire avec `closedir()`

Note: les fonctions ci-dessus ne sont pas des appels système. Pour manipuler des dossiers avec des appels système il faut utiliser les appels `open()` et `getdents()`. Toutefois en pratique on utilise les fonctions ci-dessus.

opendir()

On peut ouvrir un répertoire grâce aux fonctions:

```
DIR *opendir(const char *name);
DIR *fdopendir(int fd);
```

DIR est un type opaque.
En cas d'erreur DIR sera NULL.

Exemples de codes d'erreurs (voir man):

EACCESS	Opération interdite (permissions)
ENOENT	Le répertoire n'existe pas ou le nom est une chaîne vide
ENOTDIR	Le nom existe mais n'est pas un répertoire

readdir()

On peut lire l'entrée suivante d'un répertoire ouvert avec:

```
struct dirent *readdir(DIR *dirp);
```

Retourne soit un pointeur sur une instance de la structure dirent, soit NULL s'il n'y a plus d'entrée ou en cas d'erreur.

A chaque appel, une nouvelle entrée est retournée (s'il y en a encore).

Un seul code d'erreur:
EBADF Le descripteur dirp n'est pas valide

readdir()

La structure retournée est susceptible d'être modifiée par chaque appel.

Ne jamais appeler `free()` sur le pointeur retourné.

`readdir` n'est pas thread-safe.

closedir()

On peut fermer un répertoire ouvert avec:

```
int closedir(DIR *dirp);
```

Retourne 0 en cas de succès et -1 en cas d'erreur.

Un seul code d'erreur:

EBADF Le descripteur dirp n'est pas valide.

listDir.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <string.h>  //snprintf
#include <errno.h>
#include <dirent.h>
#include <limits.h>  //PATH_MAX

static void list_dir (const char * dir_name){

  DIR *d = opendir(dir_name);
  struct dirent *entry;
  const char *d_name;   //nom d'une entrée

  //En cas d'erreur d'ouverture
  if (! d) {
    fprintf(stderr, "Cannot open directory '%s': %s\n",
              dir_name, strerror(errno));
    exit(EXIT_FAILURE);
  }

  //Boucle sur chaque entrée
  while( (entry = readdir(d)) != NULL ) {

    // Obtient le nom de l'entrée et affiche
    d_name = entry->d_name;
    printf("%s/%s\n", dir_name, d_name);

    //Est-ce que 'entry' est un sous-répertoire
    if (entry->d_type & DT_DIR) {
      //Est-ce que 'entry' n'est pas '..' ou '.'
      if (strcmp(d_name, "..") != 0 && strcmp(d_name, ".") != 0) {
          char path[PATH_MAX];

          //forme le nom du sous-répertoire et affiche
          int path_length = snprintf (path, PATH_MAX,
                                      "%s/%s", dir_name, d_name);
          printf("%s\n", path);

          //Vérifie que le nom du sous-répertoire n'est pas trop long
          if (path_length >= PATH_MAX) {
            fprintf(stderr, "Path length has got too long.\n");
            exit(EXIT_FAILURE);
          }

          //Appel récursif
          list_dir(path);
      }
    }
  } //while(1)

  //On ferme le répertoire
  if( closedir(d) ) {
    fprintf(stderr, "Could not close '%s': %s\n",
             dir_name, strerror (errno));
    exit (EXIT_FAILURE);
  }
}

int main () {
  list_dir("/var/log/");
  return EXIT_SUCCESS;
}
```

mkdir()

On peut créer un répertoire avec:

```
int mkdir(const char *pathname, mode_t mode);
```

`pathname` est le nom du répertoire

`mode` spécifie les permissions à utiliser, il est modifié par le umask du processus: `mode & ~umask & 0777`

Retourne 0 en cas de succès et -1 en cas d'erreur.

Voir `man 2 mkdir` pour les codes d'erreur.

rmdir()

On efface un répertoire vide avec:

```
int rmdir(const char *pathname);
```

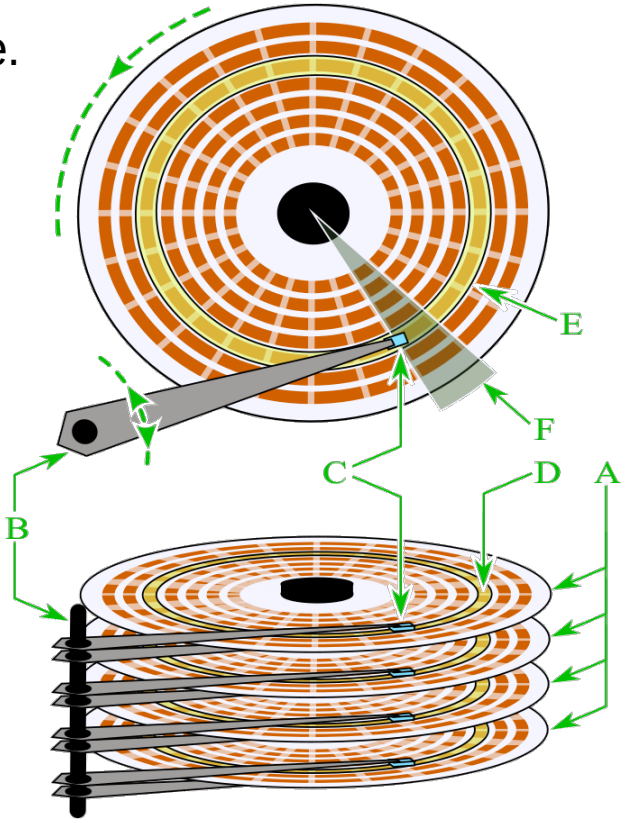
Retourne 0 en cas de succès et -1 en cas d'erreur.

Voir `man 2 rmdir` pour les codes d'erreurs.

Systèmes de fichiers (*filesystems*)

Historique

Les systèmes de fichiers ont été conçu pour accéder, avec une interface simple, à des périphériques de stockage de masse (persistents).
Les premiers périphériques de stockage de masse étaient basés sur des disques magnétiques appelés Hard Disk Drive.
La plus petite unité de lecture/écriture adressable est le secteur.
Adressable par le triplet CHS : { Cylinder, Head, Sector }



A. Plateaux
B. Bras mobile
C. Têtes de lecture/écriture reliées par le bras mobile
D. Cylindre = coupe transversale des plateaux
E. Pistes = groupe de secteurs contigüs
F. Secteur angulaire

Historique

Premier disque dur conçu en 1956 pour le supercomputer IBM 305 Ramac

- Capacité : 4.8MB
- 50 plateaux de 60cm (24"), contenant chacun 100 pistes/faces, contenant chacune 5 secteurs de 100 octets
- Débit : 8.8KB/sec.
- Vitesse : 1200 tours/min.
- 2 têtes lecture/écriture, 1 sec. pour passer d'un plateau à l'autre
- Prix du système complet : $33'200/mois

Aujourd'hui

Les **disques durs mécaniques (HDD, Hard Disk Drive)** utilisent les mêmes principes (cylindres, pistes, têtes, des secteurs), mais :

- performances et capacités ont augmenté exponentiellement
- adressage physique CHS remplacé par adressage logique : LBA (Logical Block Addressing)
- taille physique d'un secteur est souvent 4KB avec émulation à 512 bytes par le firmware

Les **disques durs à base de flash (SSD, Solid State Drive)** ne comportent pas de partie mécanique, mais présentent aussi des secteurs de 512 bytes adressés logiquement en LBA (émulation réalisée par le firmware).

La plupart des périphiques de stockage à base de flash sont similaires (clés USB, cartes SD, etc.)

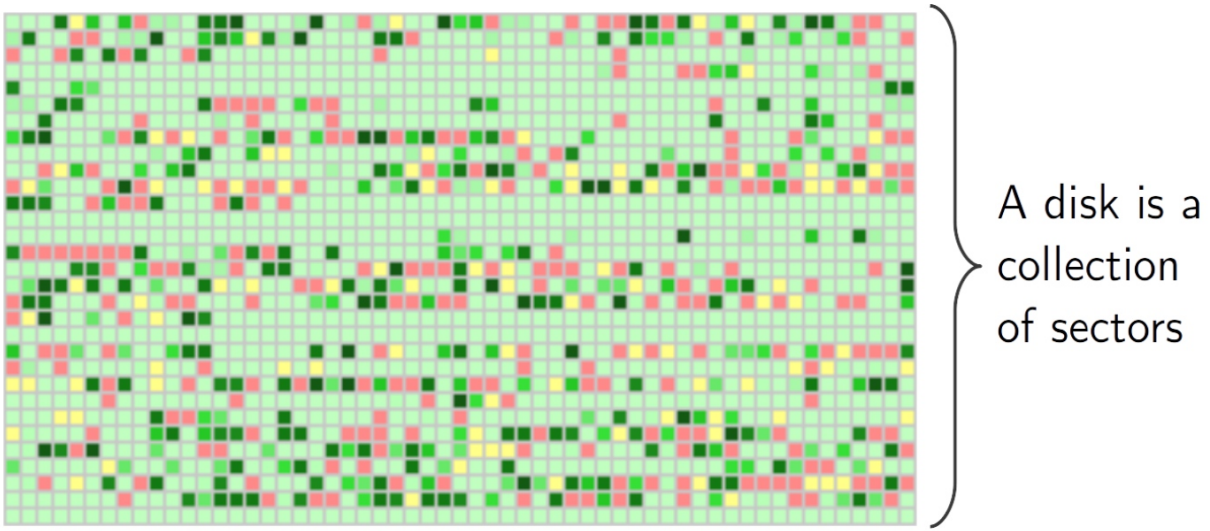
Structure d'un disque

Un disque est divisé en unités de taille identique appelées **secteurs**

La taille d'un secteur dépend du support: 512 bytes, 2KB, 4KB, etc.

Un secteur est la plus petite unité physique pouvant être lue ou écrite

La lecture ou l'écriture d'un secteur est une opération **atomique**



UNIX et blocs

Dans un OS de type UNIX, on parle de **blocs** (Microsoft appelle ça des **clusters**) plutôt que de secteurs

But : s'abstraire du type de périphérique

Tout périphérique dont on peut lire/écrire les données par unités de 512, 1024, etc. bytes est géré par le module noyau de gestion de lecture/écriture par blocs

La taille d'un bloc est potentiellement plus grande que la taille d'un secteur

Permet d'être indépendant du type de périphérique → généricité

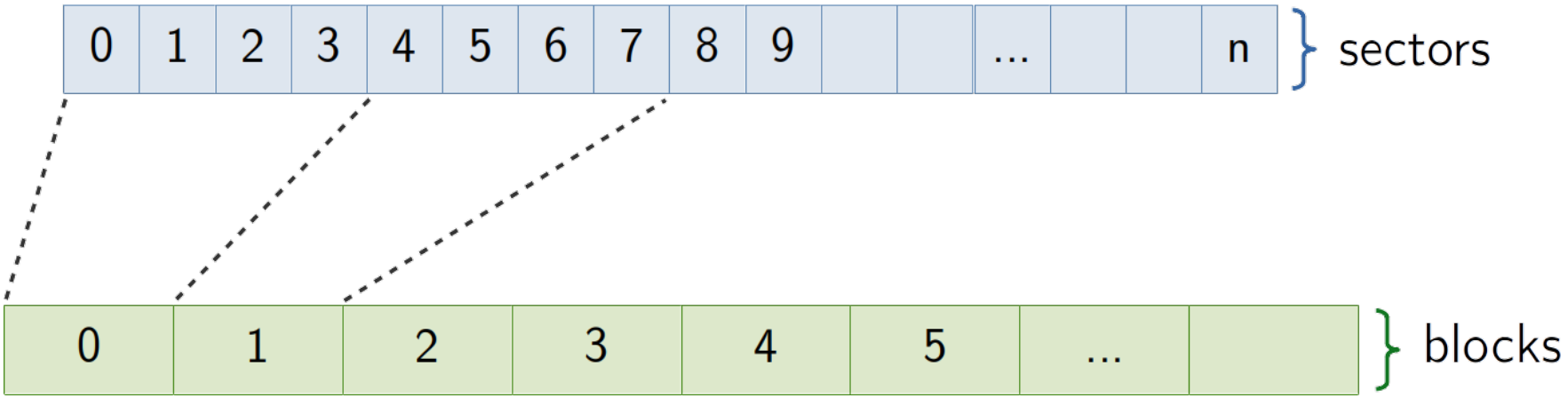
Secteurs, blocs et clusters

Un bloc est une collection contiguë de secteurs

Un système de fichiers (FS) divise l'espace disque en blocs de taille égale

Généralement, un bloc fait entre 1KB et 8KB

La commande `stat fichier` affiche la taille d'un bloc du FS où se trouve un fichier (champ "IO Block")



Interface périphérique bloc

Si on fait abstraction de la gestion du cache, l'interface d'accès à un périphérique de type bloc est très simple :

Après initialisation de la taille d'un bloc, chaque bloc est adressable logiquement via son numéro par une fonction d'écriture et une fonction de lecture :

```
init_block_dev(dev, size)
read_blocks(dev, block_nb, buf, count)
write_blocks(dev, block_nb, buf, count)
```

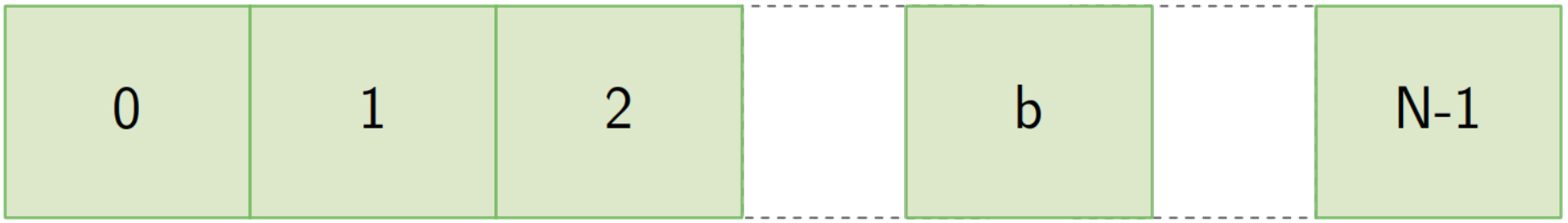
Adressage

Les blocs d'un périphérique de taille N blocs de n bytes chacun sont adressables de 0 à $N-1$ (comme un tableau en C)

La lecture/écriture se fait uniquement par bloc, donc par unité de n bytes

La lecture/écriture du bloc 0 traite les bytes de l'offset 0 à l'offset $n-1$ du périphérique

La lecture/écriture du bloc b traite les bytes de l'offset $n*b$ à l'offset $n*(b+1)-1$



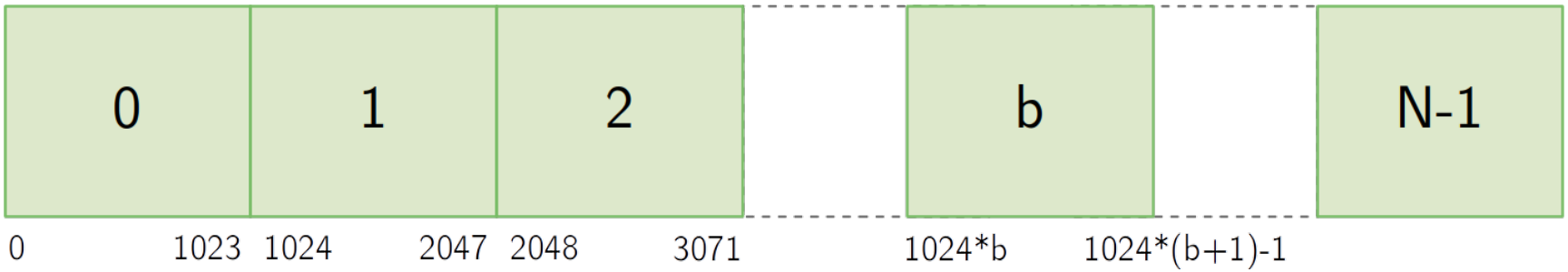
Adressage: exemple

Ex. Périphérique de 10MB (1024 * 1024 * 10 bytes), taille de bloc de 1024 bytes

Quelle est l'intervalle des offsets des bytes du bloc numéro 17 ?
$[1024*17, 1024*(17+1)-1] = [17408, 18431]$

Dans quel bloc se trouve le byte localisé à l'offset 7000 du périphérique ?
$\lfloor 7000/1024 \rfloor = \lfloor 6.8359375 \rfloor = 6$

À quel offset du bloc est localisé le byte se trouvant à l'offset 7000 du périphérique ? $7000 - 6*1024 = 856 = 7000\%1024$



Abstraction

Avec cet adressage logique, on peut facilement émuler un périphérique bloc à partir d'un fichier image. On peut y accéder via les appels systèmes usuels `read` et `write`. Ces appels système peuvent simplement être mappés sur les fonctions décrites précédemment :

```
init_block_dev(dev, size) ⟶ fd = open(file)
read_blocks(dev, block_nb, buf, count) ⟶ read(fd)
write_blocks(dev, block_nb, buf, count) ⟶ write(fd)
```

Le fichier image peut-être distant, si on implémente l'abstraction au dessus des sockets et d'un protocole qui reste très simple :

```
init_block_dev(dev, size) ⟶ s = socket()
read_blocks(block_nb, buf, count) ⟶ read(s)
write_blocks(block_nb, buf, count) ⟶ write(s)
```

Conclusion

On peut lire/écrire des blocs adressés logiquement, mais on est encore loin d'un FS

Comment à partir de ces opérations simples, peut-on construire un FS ?

Pour cela, il est nécessaire de comprendre l'allocation des blocs de données ainsi que l'organisation sur disque d'un FS

Stratégies d’allocation

Rappel:

Un système de fichiers (FS) est un ensemble de fichiers

Les fichiers stockent des données

Les fichiers et les répertoires sont tous deux des fichiers, mais de types différents

Du point de vue du FS: un fichier ordinaire ou régulier contient les données du fichier, un répertoire contient des entrées de répertoire

Fichiers et répertoires sont représentés de la même manière, mais leurs contenus sont différents

Stratégies d’allocation

Rappel:

Un fichier est composé de deux parties :

Métadonnées

type, permissions, (nom), propriétaire, pointeurs vers les données, etc.

Données (contenu)

fichier "ordinaire" : blocs contenant le contenu du fichier

répertoire : blocs contenant les références vers les fichiers présents dans le répertoire

Stratégies d’allocation

Problème:

Comment organiser un FS hiérarchique en utilisant des blocs comme unité de base ?

Comment rendre le FS performant ?

- en termes d'espace (pas d'espace perdu)
- en terme de lecture et d'écriture de fichiers

Stratégies d’allocation

La création d'un fichier ou d'un répertoire nécessite d'allouer des blocs

La stratégie d'allocation des blocs de données (contenu) doit :

- être efficace en termes d'espace
- garantir un accès rapide aux fichiers

Stratégies d’allocation: métriques

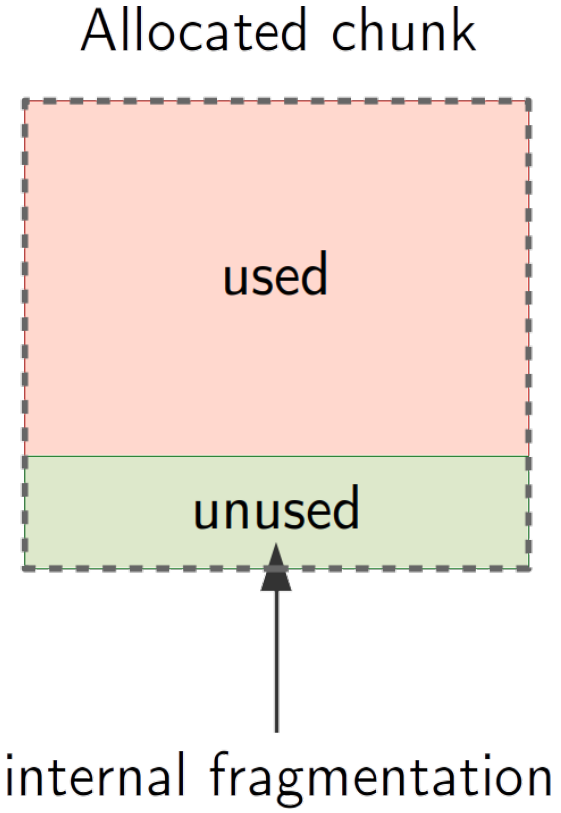
Métriques principales:

- Les fichiers peuvent-ils grandir ?
- Performance des accès séquentiels
- Performance des accès aléatoires
- Facilité d'implémentation
- Efficacité du stockage/overhead (espace utilisé pour les mécanismes de stockage vs. données effectivement stockées)
- Fragmentation

Fragmentation interne

L'espace alloué peut être plus grand
que l'espace demandé

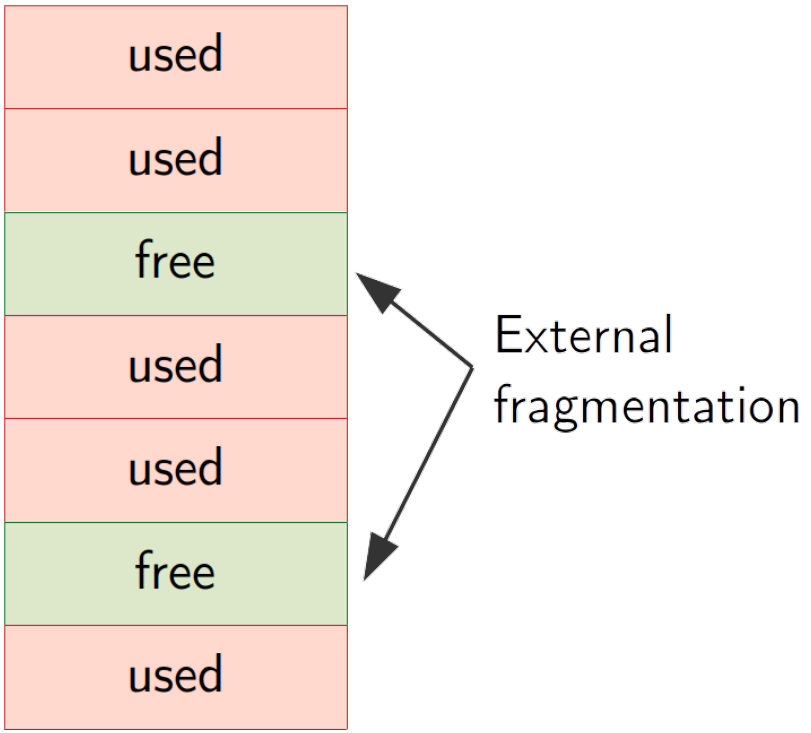
L'espace alloué non utilisé est gaspillé !



Fragmentation externe

Bien qu'il y ait suffisamment d'espace libre au total, il n'y a pas assez d'espace libre contigu pour satisfaire la demande d'allocation

Impossible de satisfaire la demande d'allocation !



Principales stratégies d'allocation de blocs

Le choix de la stratégie d'allocation dépend :

- de la technologie de stockage (SSD, HDD, etc.)
- de la stratégie d'accès et le type d'utilisation (lecture seule, gros fichiers, petits fichiers, fichiers grandissant beaucoup, etc.

Stratégies d'allocation principales:

1) contiguë
2) liste chaînée
3) indexée

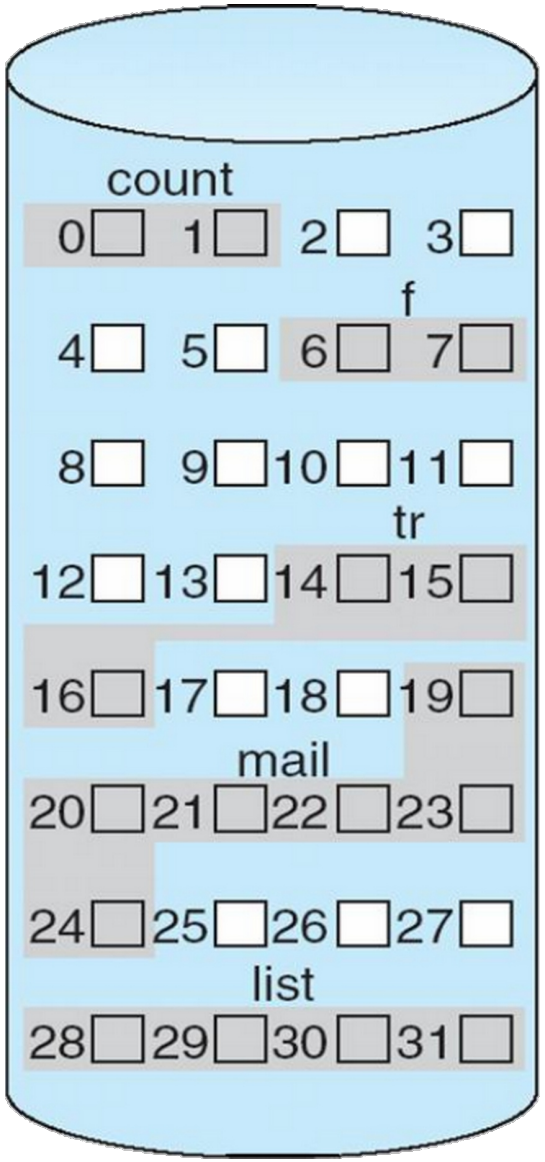
1) Allocation contiguë

Chaque fichier utilise un ensemble de blocs contigus
Un fichier est représenté par:
- l'index du premier bloc du fichier
- la taille du fichier

Avantages
- très facile à implémenter
- accès séquentiel rapide
- accès aléatoire rapide
- faible overhead stockage (structure de données très simple)

Inconvénients
- les fichiers ne peuvent pas (ou peu) grandir
- fragmentation externe importante
- fragmentation interne lorsque taille fichier < taille bloc

1) Allocation contiguë



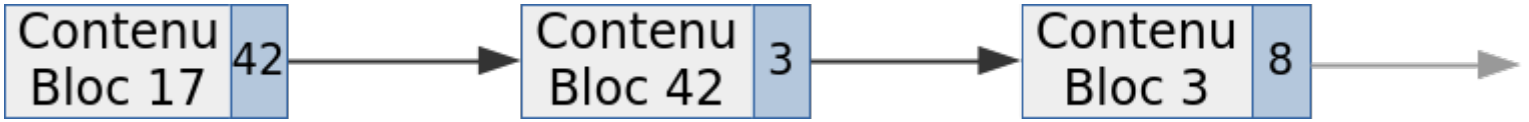
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

2) Allocation par liste chaînée

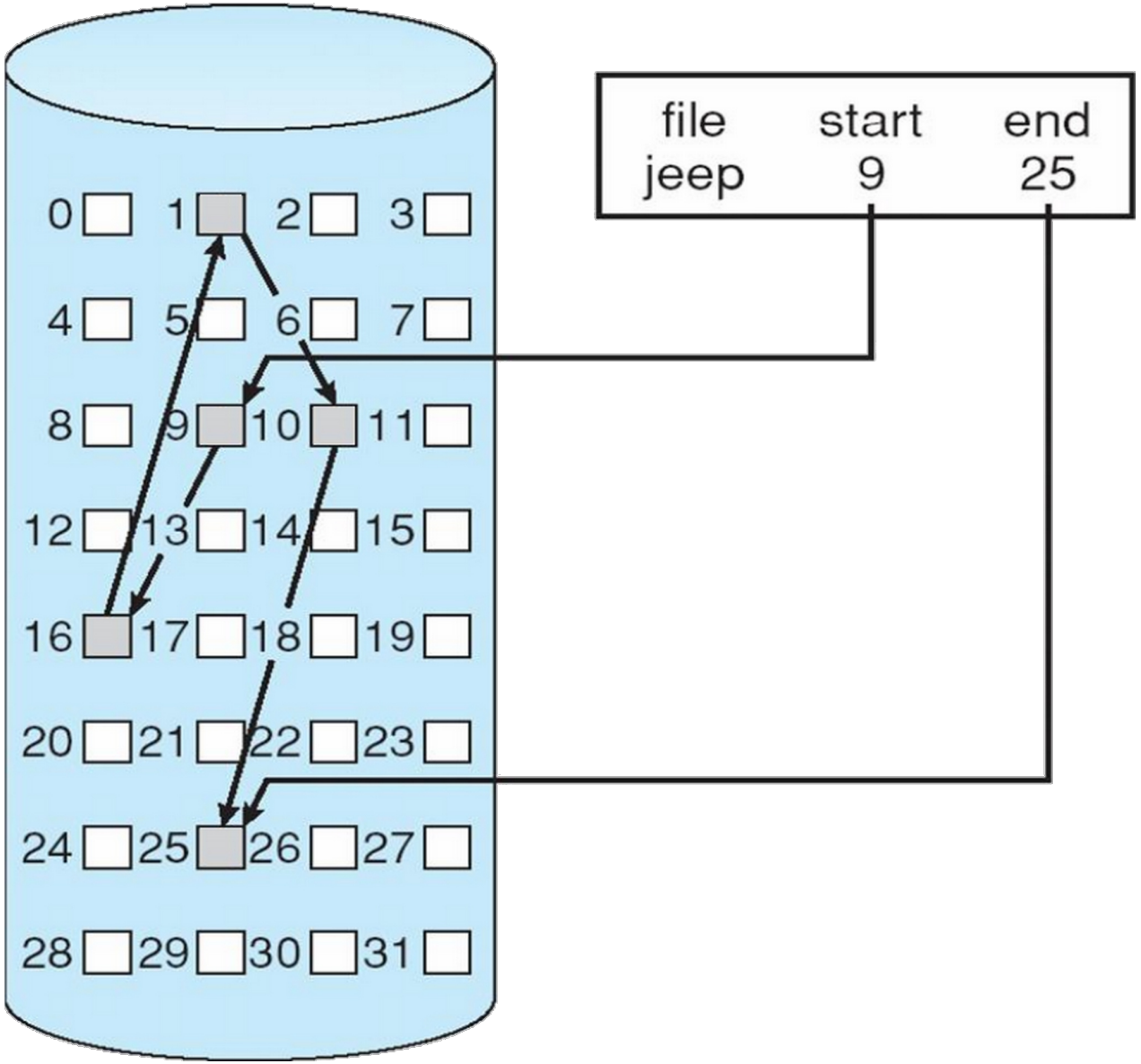
Un fichier est représenté par:

- un pointeur (index) sur le premier bloc
- la taille du fichier (ou un pointeur sur le dernier bloc)

Les bloc sont stockés sous forme de liste chainée: chaque bloc possède un pointeur (index) sur le bloc suivant



2) Allocation par liste chaînée



2) Allocation par liste chaînée

Avantages

- les fichiers peuvent facilement grandir, sans limite
- facile à implémenter, mais le stockage des pointeurs est délicat
- pas de fragmentation externe

Inconvénients

- accès séquentiel lent
- accès aléatoire lent : adresse difficile à calculer
- fragmentation interne lorsque taille fichier < taille bloc

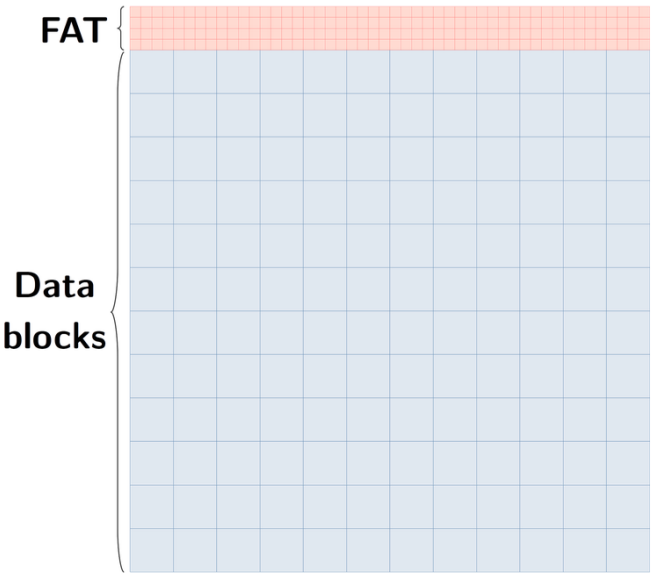
2bis) Allocation par FAT

Allocation par FAT (File Allocation Table), variante de allocation par liste chaînée
Les pointeurs de blocs sont stockés dans une table dédiée sise au début du FS:

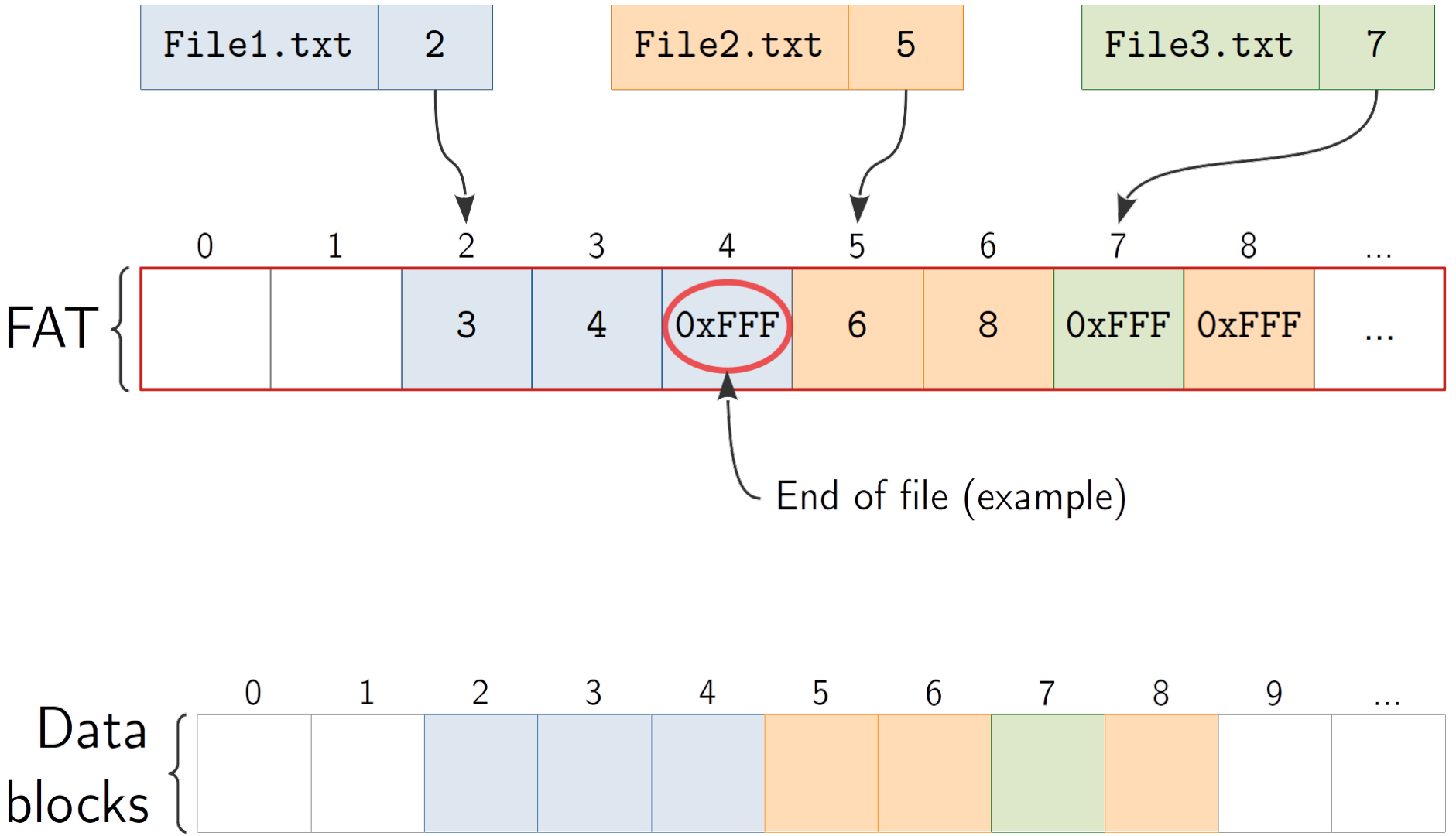
- taille de la table d'allocation des fichiers (FAT)
- une entrée FAT par bloc de données
- liste chaînée d'entrées FAT par fichier
- valeur spéciale indique la fin de la liste (EOC = End of Chain)

Système fichiers de MS-DOS et Windows,
utilisé dans cartes SD et clés USB

Nombreuses variantes:
FAT16, FAT32, exFAT, etc.



2bis) Allocation par FAT: exemple



2bis) Allocation par FAT: exemple

Soit le FS de type FAT suivant :

- Le FS comporte 1'000 blocs de données
- Taille des blocs de données : 4 KB
- Chaque entrée dans la FAT est codée sur 16 bits (une valeur est réservée pour représenter la EOC - End Of Chain)

Quelle est la taille de la FAT en bytes pour ce FS ?

$1000 * (16 \text{ bits}) = 2000 \text{ bytes}$

Quelle est la taille de fichier maximum supportée pour ce FS?

$1000 * 4096 = 4096000 \text{ bytes} = 4000 \text{ KB} = 3.9 \text{ MB}$

Combien de blocs de données cette FAT pourrait-elle gérer au maximum théoriquement ? $(2^{16}) - 1 = 65535$ blocs

2bis) Allocation par FAT

Avantages

- relativement facile à implémenter
- les fichiers peuvent facilement grandir (tant qu'il y a de l'espace dans la FAT)
- accès aléatoire rapide
- pas de fragmentation externe

Inconvénients

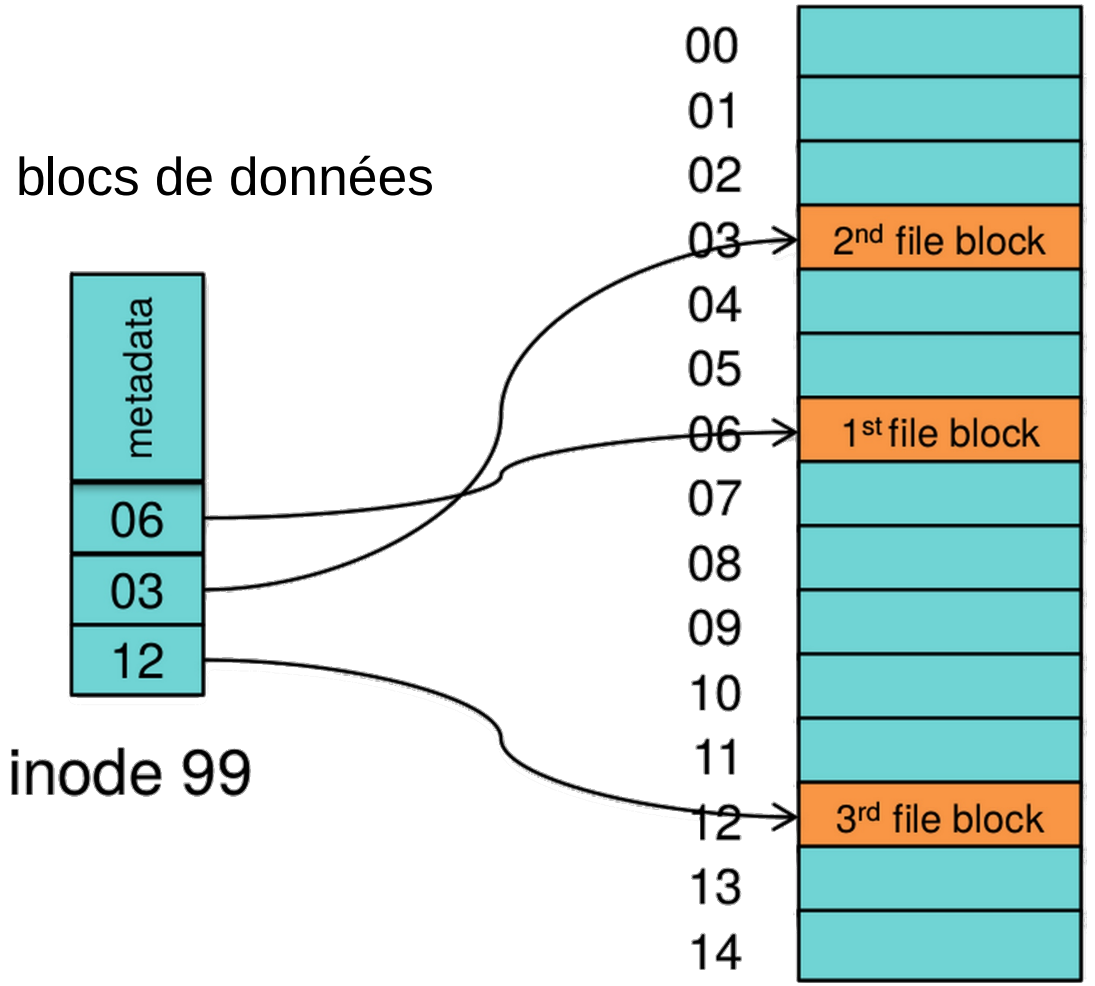
- accès séquentiel lent si les blocs ne sont pas contigus (HDD uniquement)
- la FAT doit être chargée en RAM (sinon performances catastrophiques)
- fragmentation interne lorsque taille fichier < taille bloc
- overhead important pour le stockage de la FAT (disque et RAM), en particulier avec un grand nombre de blocs/clusters

3) Allocation indexée

Chaque fichier est associé à un **inode** de taille fixe

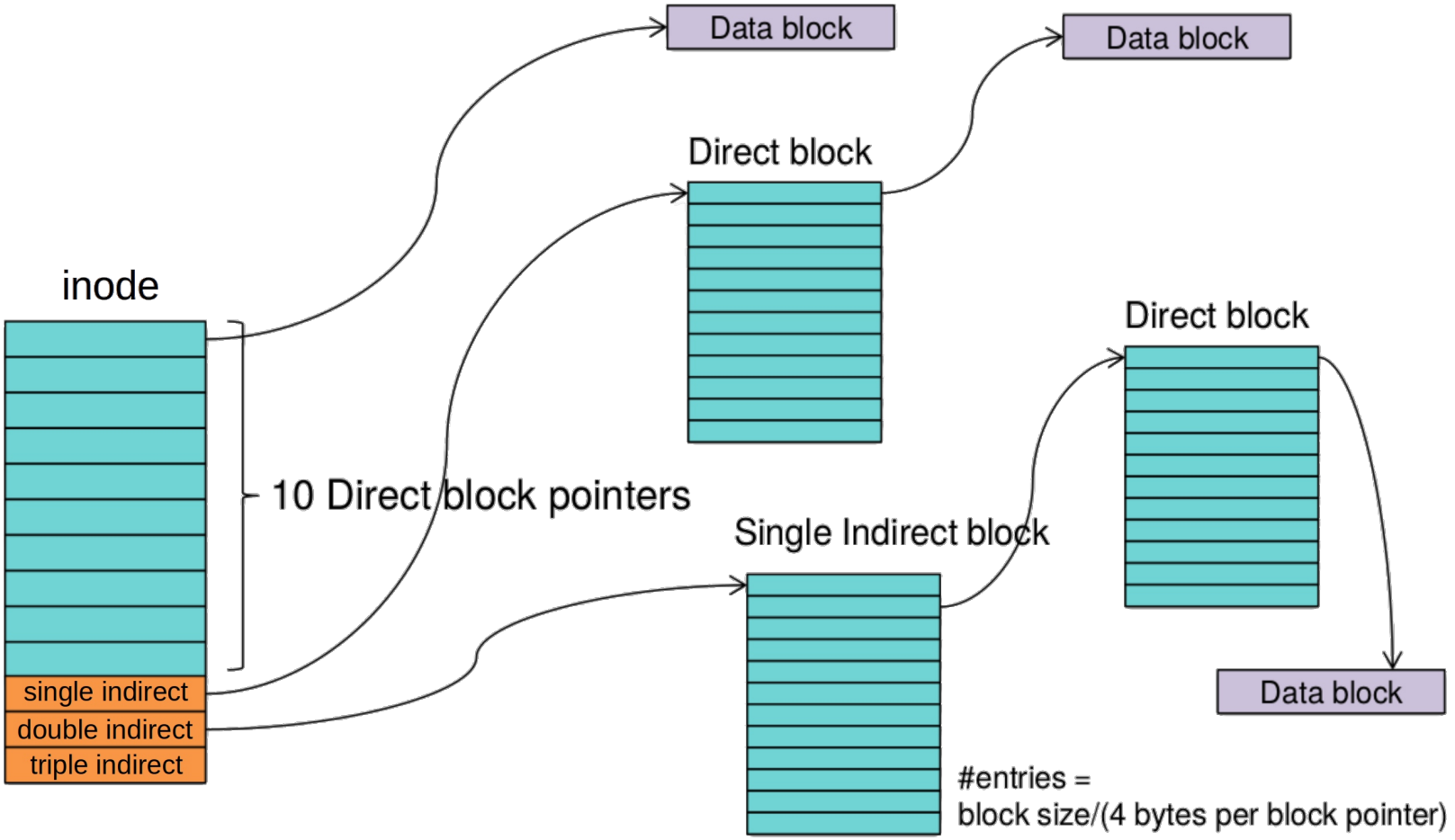
L'inode contient:

- les métadonnées du fichier
- la liste des pointeurs vers les blocs de données



3bis) Allocation indexée multi-niveau

L'inode stocke pointeurs directs, indirects, doublement indirects, etc.
Utilisé par tous les FS dans les systèmes UNIX : minix-fs, ext2, ext3, etc.



3bis) Allocation indexée multi-niveau: exemple

Soit le FS de type indexé multi-niveau suivant :

Un inode fait 64 bytes

Un inode contient 8 pointeurs directs, 2 pointeurs indirects et 1 pointeur doublement indirect

Un pointeur de bloc est stocké sur 16 bits (2 bytes)

Taille des blocs de données : 1 KB (1024 bytes)

Combien peut-on stocker de pointeurs par bloc ? 1024/2=512

Quelle est la taille de fichier maximum supportée pour ce FS?

$8 * 1024 + 2 * (512 * 1024) + (512^2) * 1024 = 269492224$ bytes

= 263176 KB = 257 MB

3bis) Allocation indexée

Avantages

- relativement facile à implémenter
- les fichiers peuvent facilement grandir (tant qu'il y a des pointeurs libres)
- accès aléatoire rapide
- pas de fragmentation externe

Inconvénients

- overhead de stockage pour les pointeurs
- l'accès rapide nécessite l'allocation de blocs contigus (HDD uniquement), sinon l'accès est potentiellement lent
- fragmentation interne lorsque taille fichier < taille bloc

3ter) Allocation par extent

Principe similaire à l'allocation indexée, avec une différence: un pointeur référence un **extent** plutôt qu'un bloc.

extent = ensemble de blocs contigüs représenté par le tuple { FirstBlockAddress, Length }

Exemple: 64 bits par extent :

- 48 bits pour l'indice du 1er bloc
- 16 bits pour la longueur (= nombre de blocs)

Avantages: moins de blocs à stocker si la plupart des allocations sont contiguës

Utilisé par les FS "modernes": ext4, btrfs, ntfs, xfs, etc.

3ter) Allocation par extent

Avantages

- les fichiers peuvent grandir (tant qu'il y a des extents libres)
- accès séquentiel rapide
- accès aléatoire rapide
- faible overhead stockage (structure de données simple)

Inconvénients

- fragmentation externe potentielle
- plus complexe que l'allocation indexée
- fragmentation interne lorsque taille fichier < taille bloc

Structure d'un FS sur disque

1) Quelles sont les structures de données stockées sur le disque qui organisent les méta-données et le contenu des fichiers?

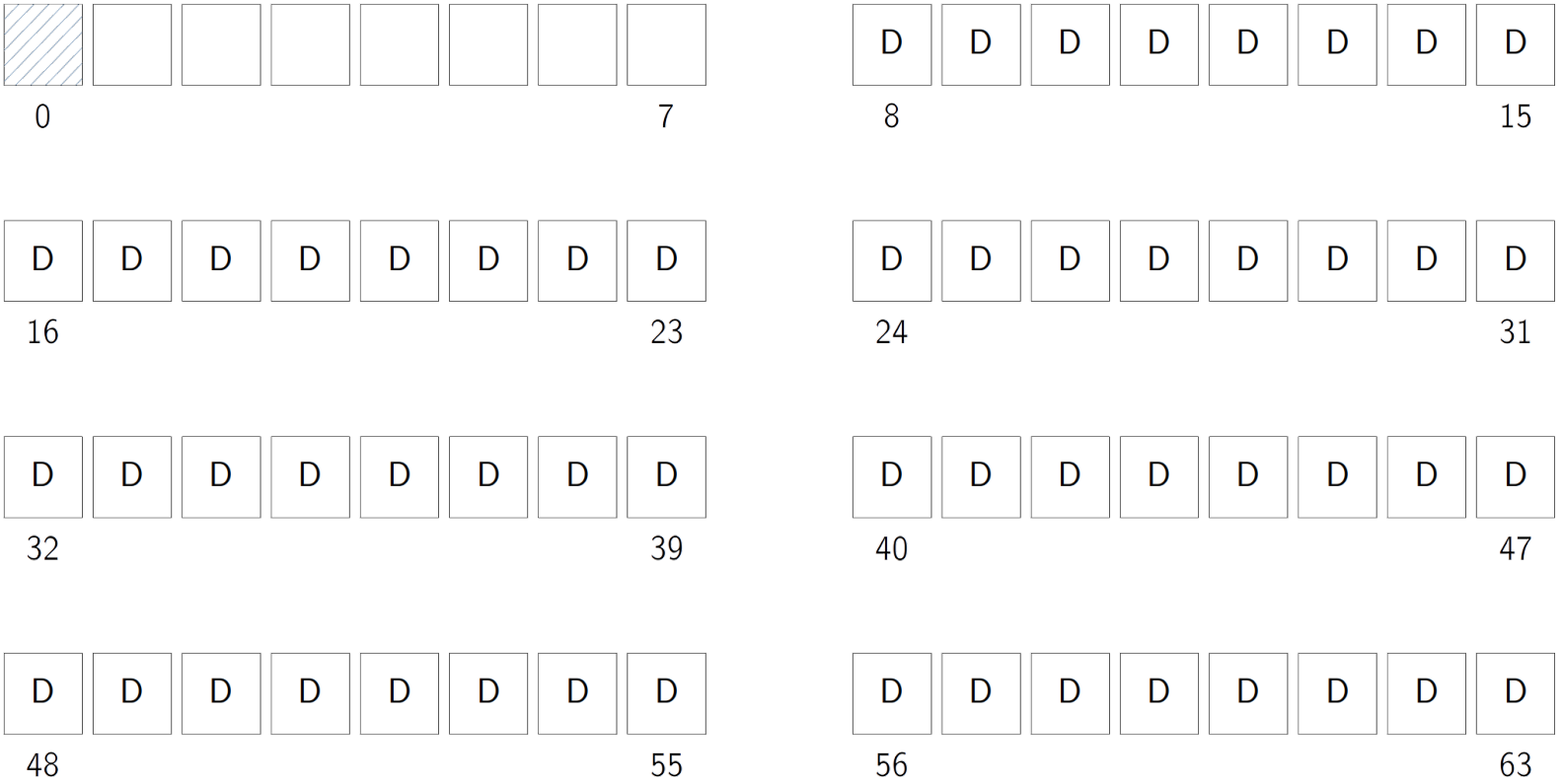
2) Comment est-ce que ces structures de données sont-elles reliées les unes aux autres?

3) Quelles sont les interfaces d'accès au système de fichiers (FS)?

p.ex. Comment est-ce que le système associe `open()`, `read()` et `write()` aux points 1 et 2 ?

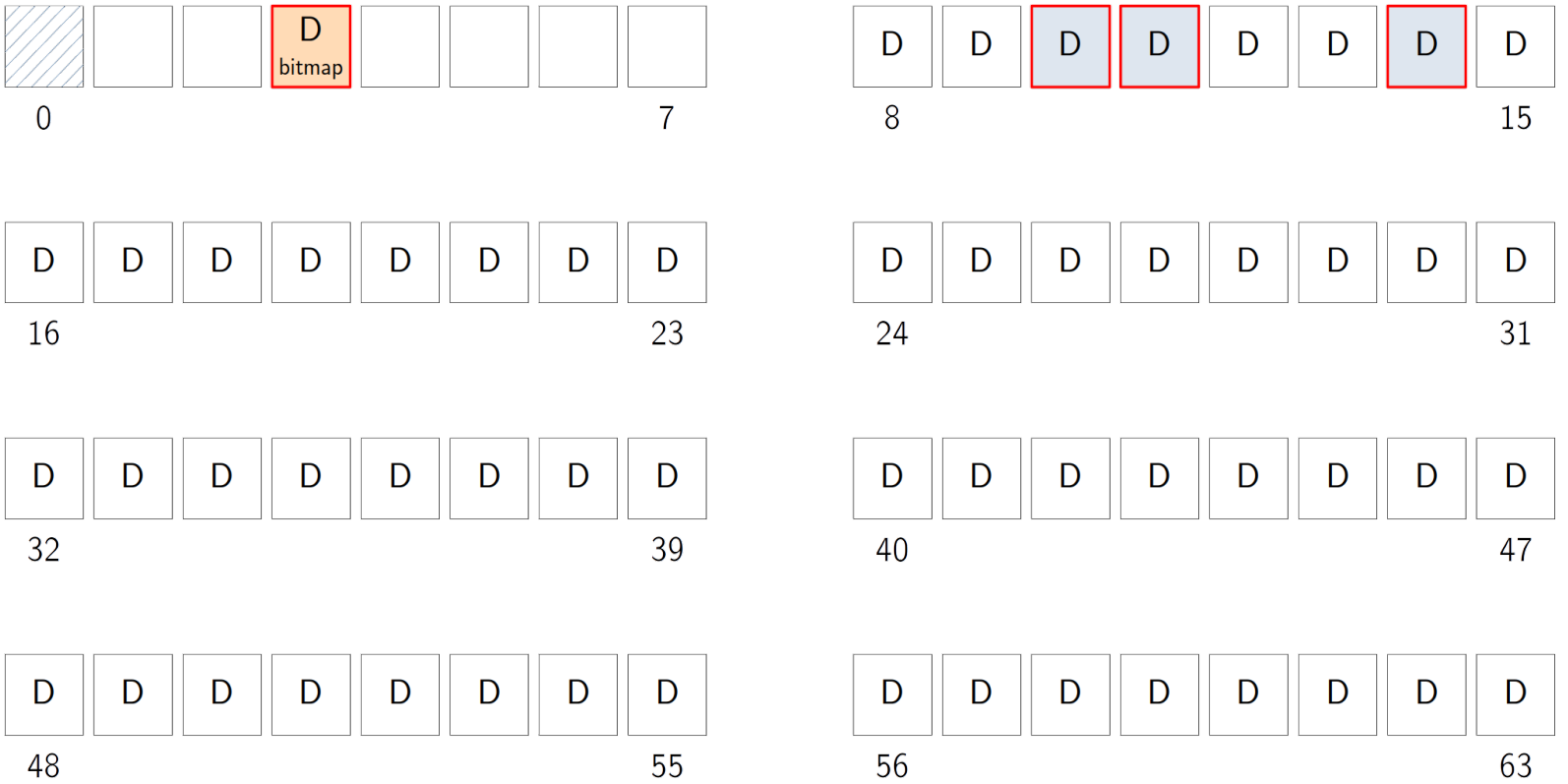
Structure d'un FS: exemple

Soit un FS constitué de 64 blocs. Le **bloc 0** est réservé et contient potentiellement le **secteur de boot et la table de partitions**.
Pour stocker des données il est préférable que la majorité des blocs soient réservés pour cela (**blocs de données**).



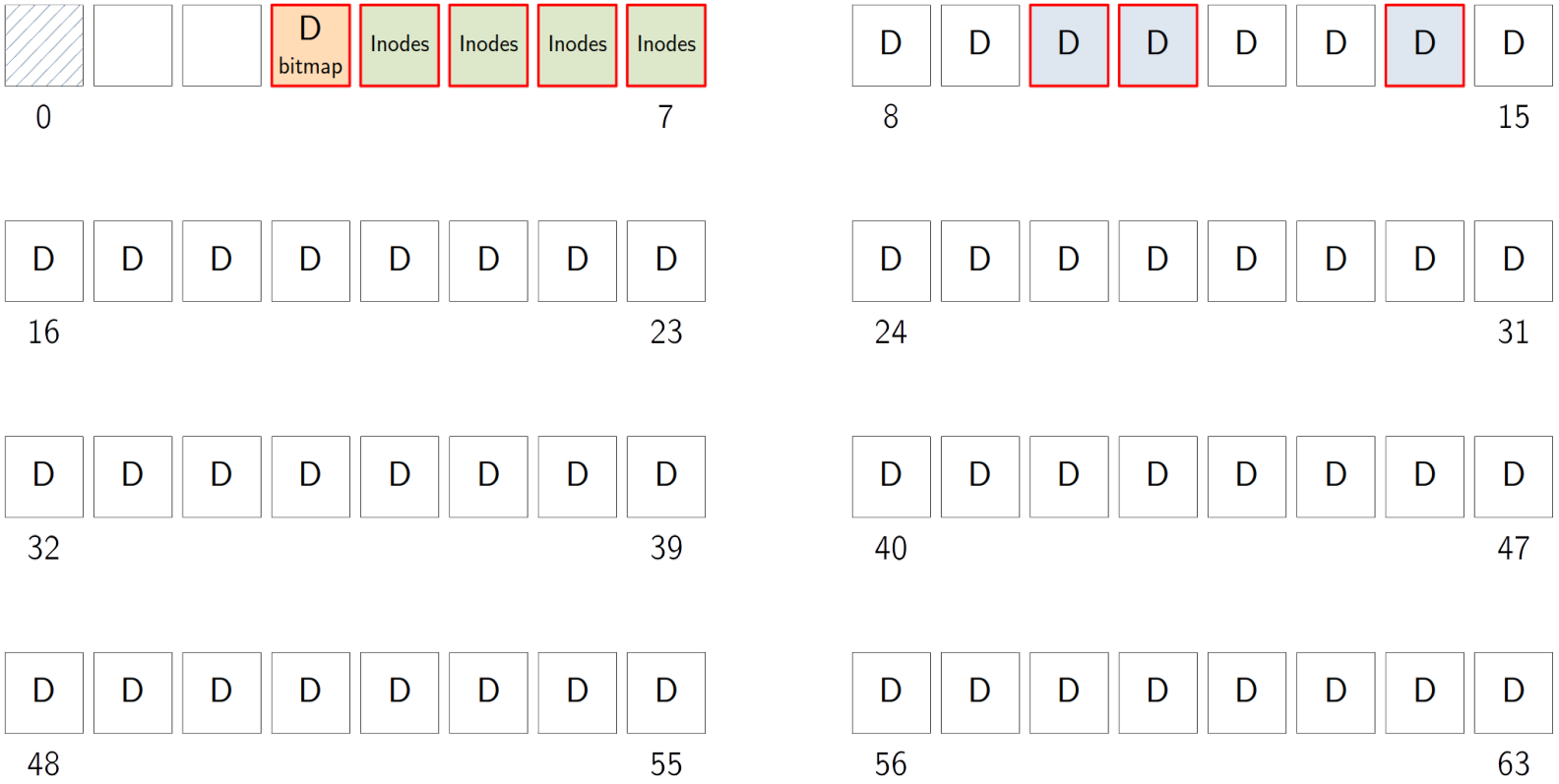
Structure d'un FS: exemple

Un tableau de bits, appelé **bitmap de blocs**, indique quels blocs de données sont disponibles ou utilisés. Contient autant de bits qu'il y a de blocs de données. P.ex. soit 3 blocs de données utilisés: 2, 3 et 6; alors la bitmap contient 0,0,1,1,0,0,1,0,0,... (D'autres structures sont possibles: listes, arbres, etc.)



Structure d'un FS: exemple

Les **inodes** ont comme fonction principale d'associer des blocs de données aux fichiers.
Les inodes contiennent également les méta-données des fichiers (propriétaire, permissions, date, etc.)



Inodes

La taille d'un inode est fixe pour un FS donné.
Les inodes sont stockés de manière contigüe au début du FS dans la table d'inodes.
Selon le FS, la taille varie (typiquement divise la taille d'un secteur, 512 bytes).
La taille d'un inode est définie à la création du FS.
Un petit FS (= faible nombre de blocs) aura typiquement une taille d'inode inférieure à un grand FS.

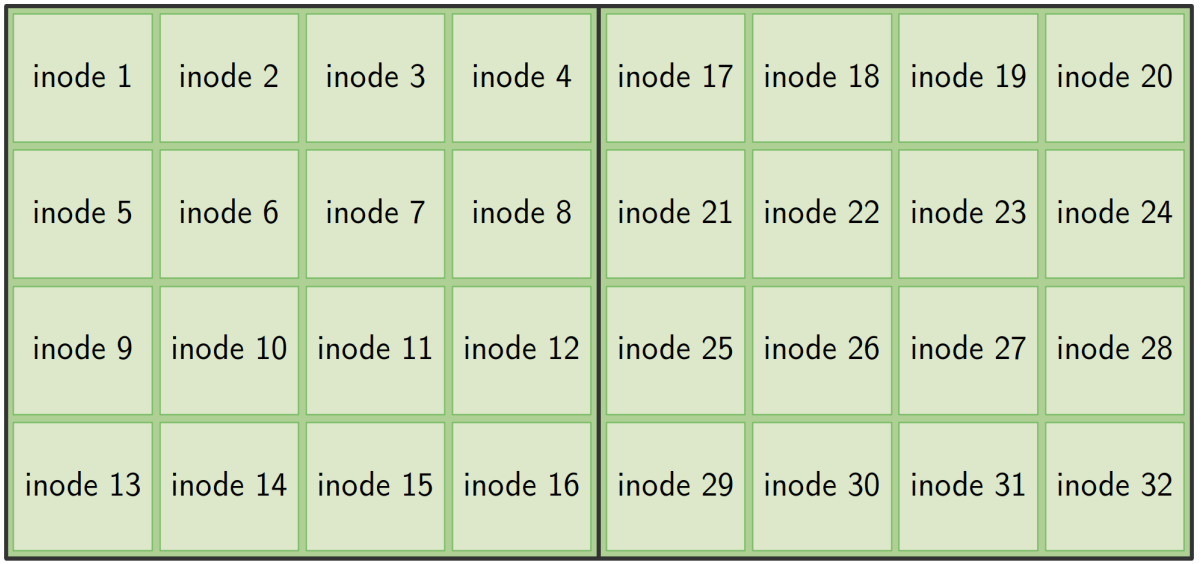
Soit des blocs de 2 KB et une taille d'inode de 128 bytes, combien d'inodes il y aura par bloc ?
2048 bytes / 128 bytes = 16 inodes par bloc

Inodes

Exemple d'une table d'inodes utilisant 2 blocs

Taille de bloc : 2 KB
Taille d'inode : 128 bytes

Attention: les inodes sont indexés à 1



Inodes

Chaque fichier est associé exactement à 1 inode.

Chaque n° d'inode est unique à un FS donné.

Des FS différents peuvent utiliser les mêmes n° d'inodes.

Le n° d'inode d'un fichier supprimé est réutilisé par le FS.

Sur Linux, utiliser la commande `ls -i` pour visualiser les n° des inodes.

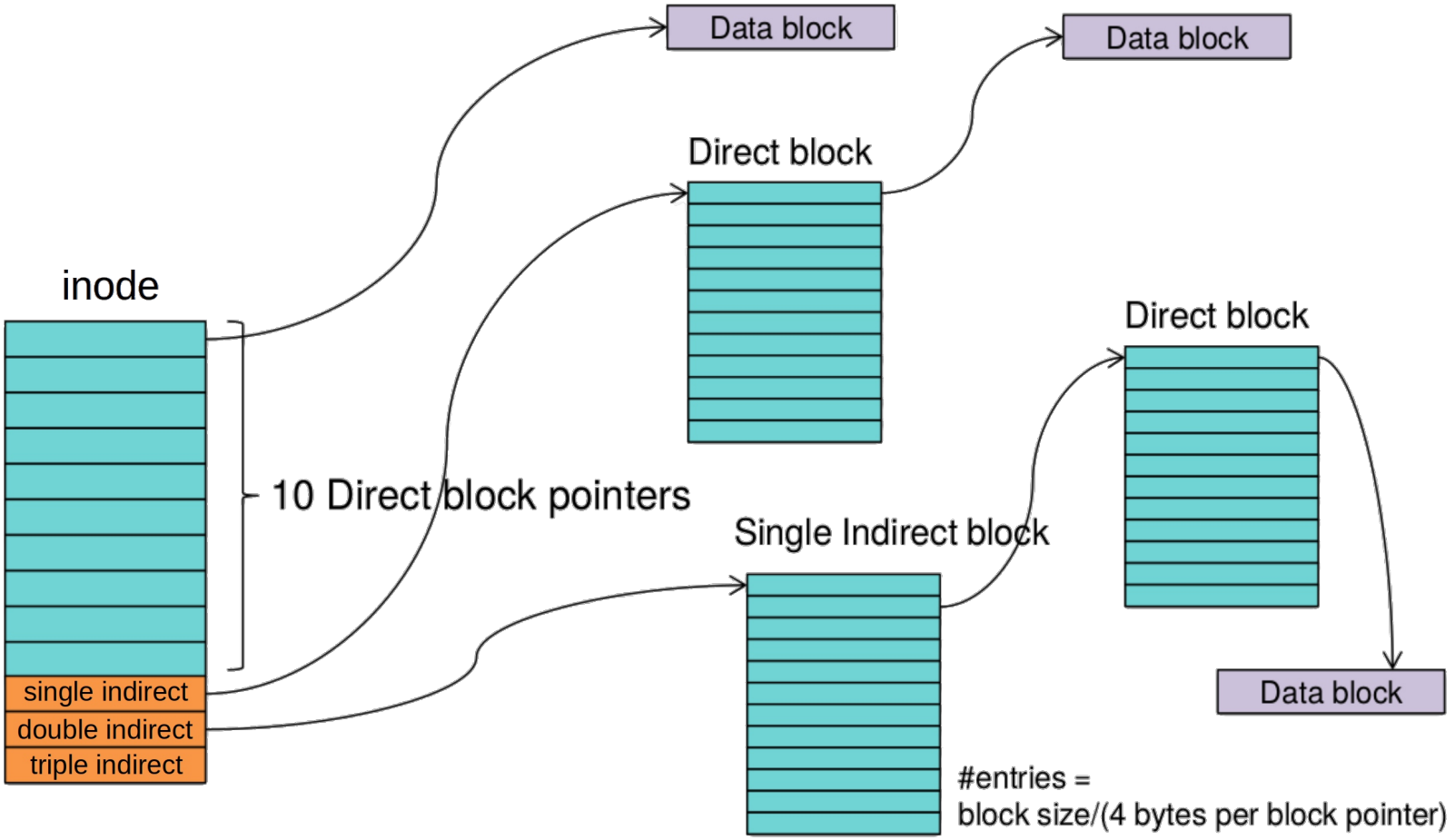
Structure d'un inode

Peut varier selon le FS. Un n° d'inode = 0 indique une entrée non utilisée.

Champ	Description
`type`	**Fichier**, **répertoire**, device, lien, etc.
`uid`	UID du propriétaire
`gid`	GID du propriétaire
`rwx`	Permissions (droits d'accès)
`size`	Taille en bytes
`time`	Date du dernier accès
`n_links`	Nombre de liens/chemins d'accès vers l'inode
`direct[N]`	Pointeurs directs vers les blocs du contenu du fichier, **dans l'ordre**
`indir[M]`	Pointeurs indirects vers les blocs du contenu du fichier, **dans l'ordre**
`dbl_indir[P]`	Pointeurs doublement indirects vers les blocs du contenu du fichier, **dans l'ordre**

Allocation blocs de données dans un inode

Exemple avec: 10 pointeurs directs, 1 pointeur indirect, 1 pointeur doublement indirect, 1 pointeur triplement indirect



Fichiers

Un fichier **régulier** est un fichier habituel dont les données (contenu) peuvent être:

- du texte si c'est un fichier texte
- des données d'image si c'est un fichier PNG, JPG, ...
- des données audio si c'est un fichier MP4, FLAC, ...
- etc.

Un fichier de type **répertoire** (*directory*) est un fichier qui contient des entrées de répertoires (*directory entries*).

Répertoire

Une entrée de répertoire associe simplement un nom à un inode

Structure d'une entrée de répertoire (`dir_entry`) :

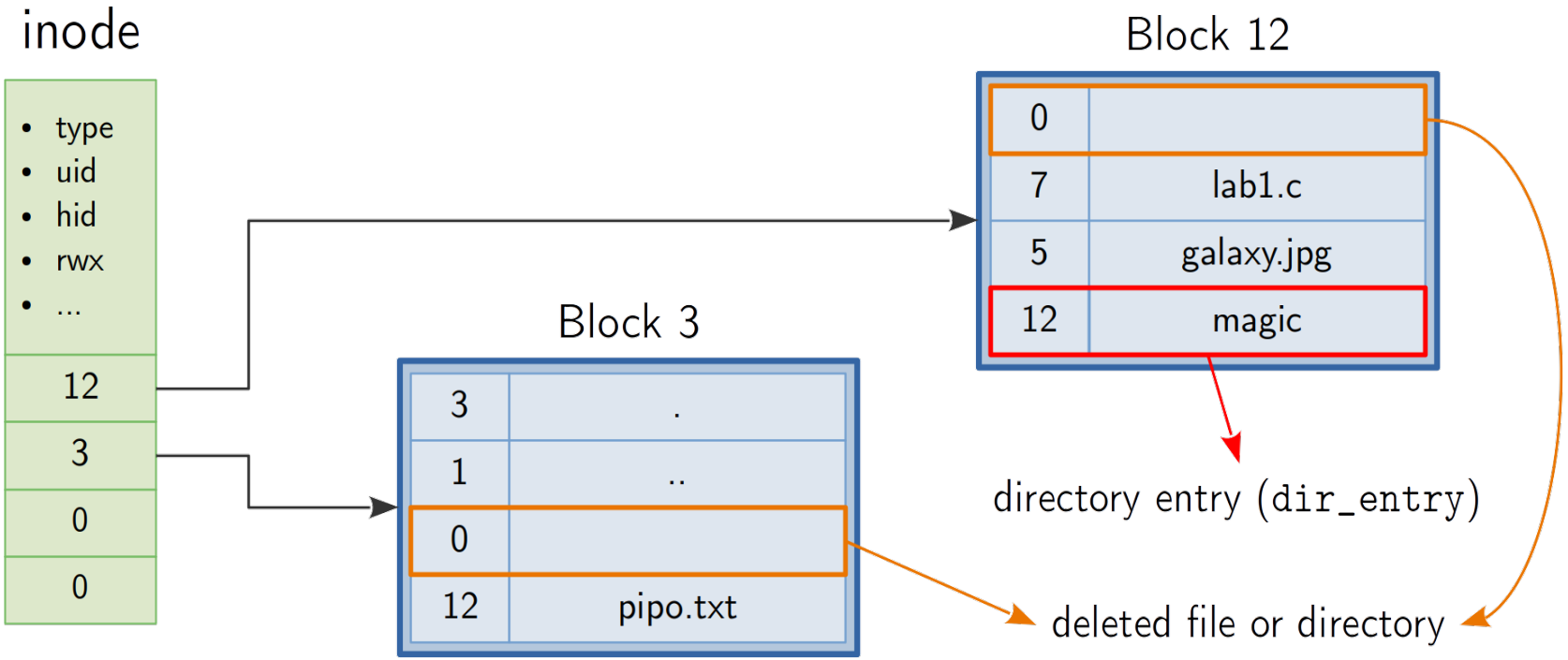
inode	numéro d'inode du fichier
name	nom associé à cet inode (fichier)

Un fichier n'est donc pas caractérisé par son nom, mais par son inode !

Répertoire

Le contenu d'un fichier de type "répertoire" contient des entrées de répertoires

Par convention, dans la plupart des FS, l'inode n° 1 est le répertoire racine du filesystem: /



Liens symboliques (*symlinks* ou *soft links*)

Un fichier de type lien symbolique a comme contenu une chaîne de caractères représentant le chemin de destination du lien (1 seul bloc de données).

Lors de l'analyse d'un chemin d'accès, si le système rencontre un lien symbolique, alors son contenu est concaténé avec le chemin déjà parcouru.

Pour créér un symlink pipo.c vers code/src/prog.c:

```
ln -s code/src/prog.c pipo.c
```

Un lien symbolique peut pointer vers un fichier ou un répertoire.

Un lien symbolique peut traverser les FSs.

Si la destination d'un lien symbolique est déplacée ou supprimée, le lien symbolique devient alors invalide.

Liens durs (*hard links*)

Un lien dur est simplement une nouvelle entrée de répertoire (`dir_entry`) pointant vers un inode donné. Il s'agit donc simplement d'un nom supplémentaire pointant vers le même inode.

Pour créér un hard link pipo.c vers code/src/prog.c :

```
ln code/src/prog.c pipo.c
```

Un lien dur ne peut pas pointer vers un répertoire.

Un lien dur ne peut pas traverser les FSs car un numéro d'inode est seulement unique à un FS !

Liens durs (*hard links*)

La valeur Links affichée par la commande `stat` ou `ls` indique le nombre de dir_entry pointant sur le fichier (inode).

Un fichier (inode) n'est réellement supprimé que lorsque le dernier dir_entry (lien dur) pointant dessus est supprimé !
Aussi, si un descripteur de fichier ouvert référence ce fichier (inode), alors le fichier ne sera supprimé que lorsque le dernier descripteur sera fermé.

Les liens durs (ou dir_entry) sont donc simplement des références vers un inode (fichier).

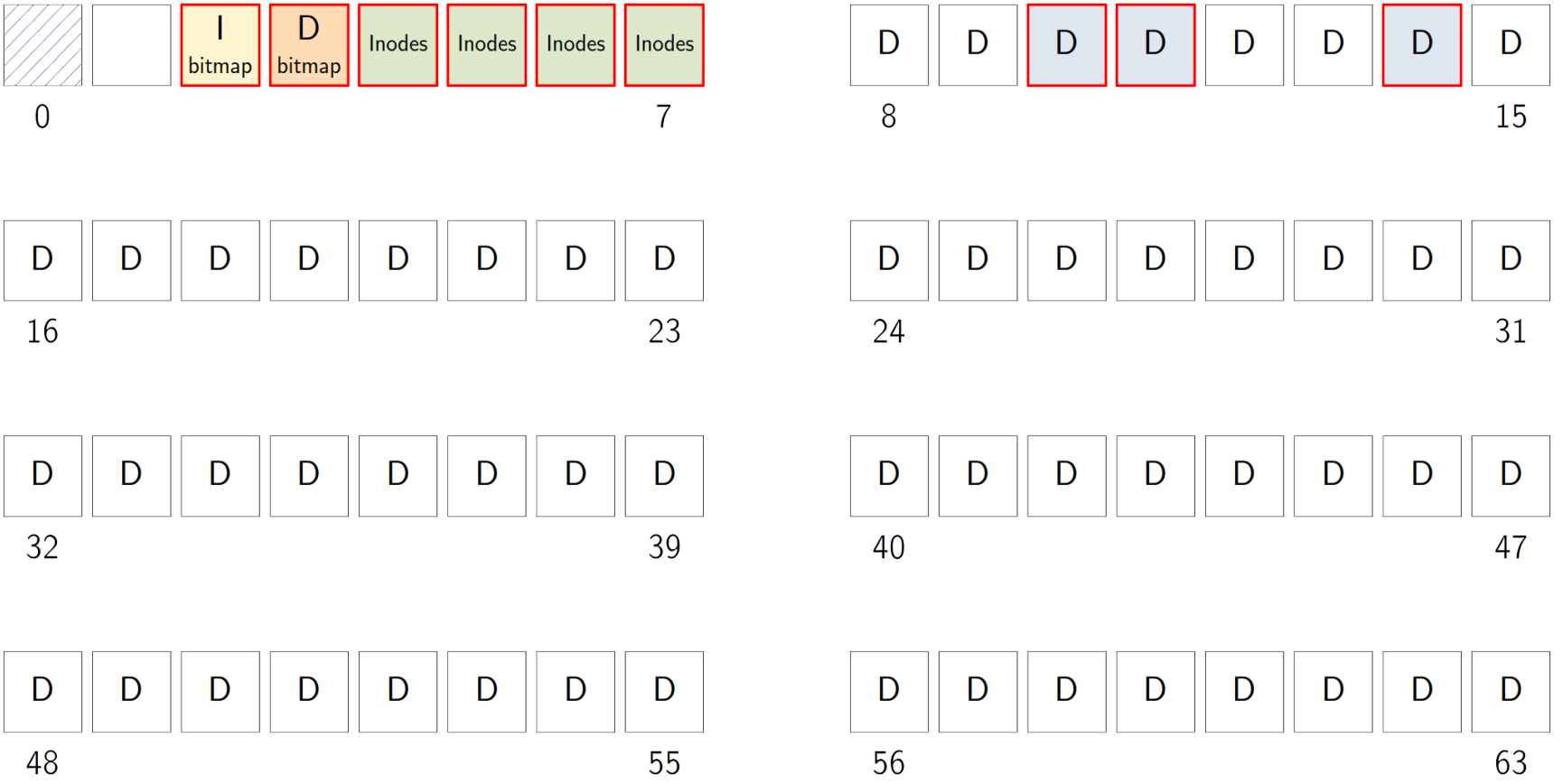
Liens durs (*hard links*)

```
[dr@dr0 ~]$ ls -i
total 20
117440690 drwx------. 3 dr   dr    111 Jul 15 14:41 ./
 25171124 drwxr-xr-x. 3 root root   16 Nov  3  2024 ../
117440694 -rw-------. 1 dr   dr   3893 Jul 13 21:13 .bash_history
117440692 -rw-r--r--. 1 dr   dr    961 Dec 13  2024 .bash_profile
117440693 -rw-r--r--. 1 dr   dr    492 Jan 23  2023 .bashrc
117440702 -rw-------. 1 dr   dr     20 Jul 15 14:41 .lesshst
125832384 drwx------. 2 dr   dr    103 Jun 14  2024 .ssh/

[dr@dr0 ~]$ stat .bashrc
  File: .bashrc
  Size: 492             Blocks: 8          IO Block: 4096   regular file
Device: 805h/2053d      Inode: 117440693   Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1001/      dr)   Gid: ( 1001/      dr)
Access: 2025-07-15 14:18:53.516611732 +0200
Modify: 2023-01-23 23:42:13.000000000 +0100
Change: 2024-06-14 15:25:00.590054349 +0200
 Birth: 2024-06-14 15:25:00.590054349 +0200
```

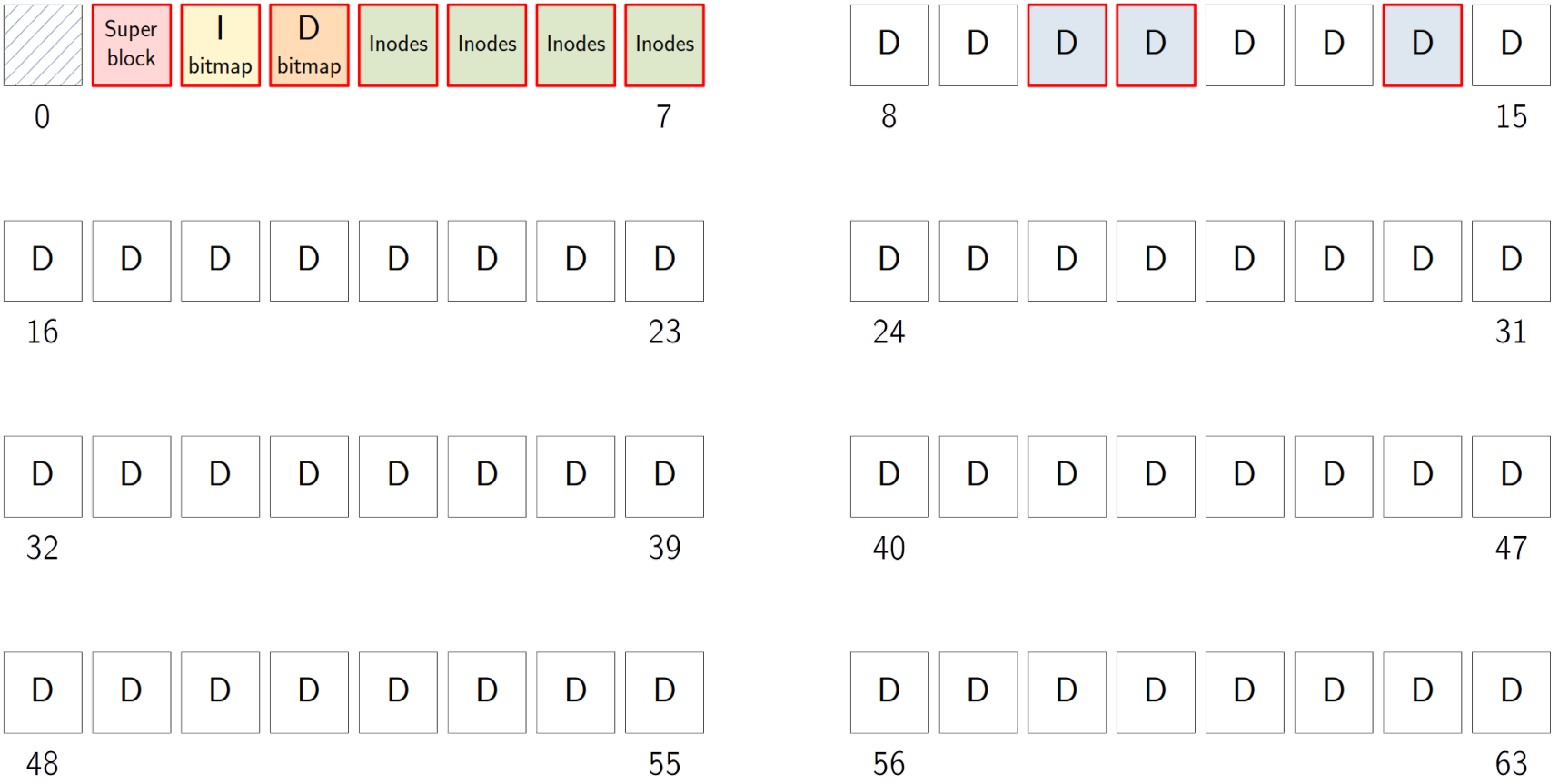
Structure d'un FS: exemple

Similairement à l'allocation des blocs de données, on utilise un **bitmap pour les inodes** utilisés. (D'autres structures sont possibles: listes, arbres, etc.)



Structure d'un FS: exemple

Le **superblock** stocke des informations globales sur le FS: signature du FS, nombre d'inodes total, nombre de blocs total, taille du bitmap des inodes, etc.



Nombre d'inodes

Le nombre d'inodes détermine le nombre maximum de fichiers pouvant être créés sur un FS.

Combien d'inodes faut-il réserver pour un FS d'une taille donnée ?

Nombre d'inodes: MINIX

`mkfs.minix` se base sur la taille du disque (DS); ci-dessous *fs_blocks* = nombre total de blocs dans le FS

Si DS > 2 GB alors inodes = *fs_blocks*/16
Si DS > 500 MB alors inodes = *fs_blocks*/8
Sinon : inodes = *fs_blocks*/3

Ensuite, arrondir au prochain multiple de la taille d'un inode (32 bytes)

Exemple: disque de 64 MB et des blocs de 1 KB →
inodes = (64*1024/3 + 32) & 0xFFFFFFE0 = 21856

Nombre d'inodes: EXT2/3/4

`mke2fs` se base sur un paramètre nommée *bytes-per-inode*

Indique de créér un inode pour chaque *bytes-per-inode* bytes d'espace disque

Exemple: disque de 64 MB et *bytes-per-inode* = 4096 → inodes = (64 * 1024 * 1024) / 4096 = 16384

MINIX-FS

MINIX-fs est le système de fichiers (FS) natif du système d'exploitation MINIX écrit par Andrew S. Tanenbaum en 1987.
Le but de MINIX-fs était :

- de répliquer la structure du FS de UNIX, mais sans les fonctionnalités avancées et complexes de ce dernier
- de fournir une aide à l'enseignement des systèmes de fichiers et des OS

Les premières versions du noyau Linux (1992) utilisaient MINIX-fs comme FS.
1992 : Rémy Card implémente ext (Extended Filesystem) pour remplacer MINIX-fs; ext est basé sur une structure similaire à MINIX-fs
1993 : Rémy Card publie ext2 (amélioration de performances)
2001 : Theodore Ts'o et d'autres publient ext3 (ajout de journalisation)
2006 : la version unstable de ext4 est publiée
2008 : la version stable de ext4 est publiée (amélioration de performances et augmentations des limites de tailles)

MINIX-FS

Il existe 3 versions de MINIX-fs : v1, v2 et v3.
Nous présentons ici la v1 car c'est la plus simple.

Il existe deux variantes de MINIX-fs v1 :

- variante avec noms de fichiers de 14 caractères max (superblock magic value 0x137F)
- variante avec noms de fichiers de 30 caractères max (superblock magic value 0x138F)

Comparaison MINIX-FS et EXT2/3/4

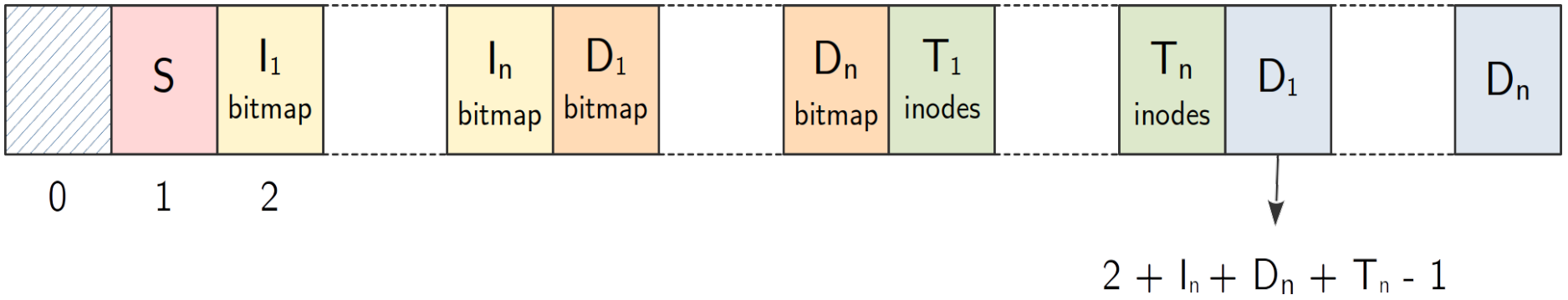
Valeurs max pour les caractéristiques principales des FS :

Name	FS	File	Block	Extent	File length
MINIX-fs v1.0	256 MB	256 MB	1 KB (fixed)	n/a	14/30
ext	2 GB	2 GB	?	n/a	255
ext2	32 TB	2 TB	8 KB	n/a	255
ext3	32 TB	2 TB	8 KB	n/a	255
ext4	1 EB	16 TB	4 KB	128 MB	255

Remarque : ext3 est simplement ext2 avec l'ajout d'un journal

MINIX-FS

Structure générale sur disque de MINIX-fs v1.0:



MINIX-FS

Structure d'un inode dans MINIX-fs v1.0:

```
struct minix_inode {
    u16 i_mode;     // file type and permissions for file
    u16 i_uid;      // user id
    u32 i_size;     // file size in bytes
    u32 i_time;     // inode modification time
    u8 i_gid;       // group id
    u8 i_nlinks;    // number of dir entry pointing to
                    // this inode
    u16 i_zone[7];          // direct pointers
    u16 i_indir_zone;       // indirect pointer
    u16 i_dbl_indir_zone;   // double indirect pointer
};
```

Taille d'un inode? 32 bytes

Inodes par bloc? 1024 bytes (taille du bloc) / 32 bytes (taille d'un inode) = 32

Structure sur disque stockée selon l'ordre little-endian

Endianness

L'***endianness*** défini la manière dont les entiers sont stockés en mémoire.

Big-endian:

le byte le plus significatif (d'un mot) est stocké à l'adresse la plus basse
le byte le moins significatif est stocké à l'adresse la plus haute

Little-endian:

le byte le plus significatif (d'un mot) est stocké à l'adresse la plus haute
le byte le moins significatif est stocké à l'adresse la plus basse

Taille de mot	Valeur	Big-endian	Little-endian
8 bits (1 byte)	4c	4c	4c
16 bits (2 bytes)	4c3f	4c 3f	3f 4c
32 bits (4 bytes)	4c3f85ed	4c 3f 85 ed	ed 85 3f 4c

MINIX-FS

Format du champs `i_mode` (16 bits) de l'inode:

Bits	Description	
`12-15`	Type de fichier	`0x1` : named pipe (FIFO) `0x2` : char device `0x4` : directory `0x6` : block device `0x8` : regular file `0xA` : symbolic link
`11`	SUID bit	
`10`	SGID bit	
`9`	Sticky bit	
`6-8`	user permissions (rwx)	
`3-5`	group permissions (rwx)	
`0-2`	others permissions (rwx)	

MINIX-FS

Structure d'une entrée de répertoire dans MINIX-fs v1.0 :

```
// Variante : superblock magic 0x137F
#define MINIX_NAME_LEN 14

struct minix_dir_entry {
    u16 inode;
    char name[MINIX_NAME_LEN];
};
```

Taille d'un dir_entry? 2 bytes (16 bits) + 14 * sizeof(char) = 16 bytes
si on considère sizeof(char) équivalent à 1 byte

MINIX-FS - Superblock

Structure du superblock dans MINIX-fs v1.0 :

```
struct minix_super_block {
    u16 s_ninodes;        // total number of inodes
    u16 s_nzones;         // total number of blocks (including superblock)
    u16 s_imap_blocks;    // inodes bitmap size in blocks
    u16 s_zmap_blocks;    // data blocks bitmap size in blocks
    u16 s_firstdatazone;  // index of first data block
    u16 s_log_zone_size;  // block size in bytes = 1024*2^s_log_zone_size
    u32 s_max_size;       // max file size in bytes
    u16 s_magic;          // 0x137f = v1 with 14 characters dir_entry
                          // 0x138f = v1 with 30 characters dir_entry
    u16 s_state;          // was the FS properly unmounted?
};
```

Taille du superblock? 20 bytes

MINIX-FS - Exemple de superblock

Exemple de superblock pour un disque de 20 MB et une taille de bloc de 1 KB:
`fs_blocks` = nombre de blocs au total = 20480
Nombre inodes = (`fs_blocks`/3 + 32) & 0xFFFFFFE0 = 6848
(on arrondit au prochain multiple de la taille d'un inode)

```
struct minix_super_block {
    u16 s_ninodes = 6848;         // total number of inodes
    u16 s_nzones = 20480;         // total number of blocks
                                  // = (20*1024^2)/1024
    u16 s_imap_blocks = 1;        // inodes bitmap size in blocks
                                  // need 6848 bits
                                  // -> 1 block = 8192 bits
    u16 s_zmap_blocks = 3;        // data blocks bitmap size in blocks
                                  // need 20480 - 220 bits -> 3 blocks = 8192*3 bits
    u16 s_firstdatazone = 220;    // index of first data block
    u16 s_log_zone_size = 0;      // block size = 1024*2^s_log_zone_size
                                  // = 1024*2^0 = 1024
    u32 s_max_size = 268966912;   // max file size in bytes
                                  // = (7+(1024/2)+(1024/2)^2)*1024
    u16 s_magic = 0x137F;         // 0x137f Minix filesystem version 1
    u16 s_state = 1;              // was the FS properly unmounted?
};
```

MINIX-FS - Exemple de superblock

Le calcul pour arriver à 220 comme premier bloc de données (`s_firstdatazone`) est le suivant:

- 1 bloc pour le superbloc
- 1 bloc pour la bitmap des inodes `s_imap_blocks`
- 3 blocs pour la bitmap des zones (= blocs de données) `s_zmap_blocks`
- 6848 inodes / 32 inodes par bloc = 214 blocs pour la table d'inodes

Total: 1 + 1 + 3 + 214 = 219 blocs

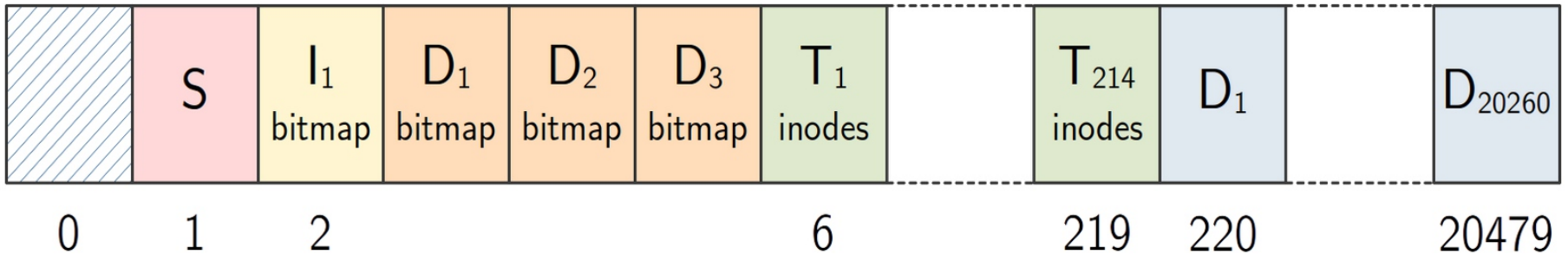
MINIX-FS - Exemple de superblock

Le calcul pour arriver à 268966912 bytes pour la taille max d'un fichier (`s_max_size`) est le suivant:

- 7 pointeurs directs: 7 * 1024 = 7168 bytes (Les 7 premiers pointeurs de l'inode pointent directement vers des blocs de données)
- 1 pointeur d'indirection simple: 512 * 1024 = 524288 bytes (Un bloc d'indirection simple contient 1024 / 2 = 512 pointeurs, car chaque pointeur fait 2 bytes)
- 1 pointeur d'indirection double: 512 * 512 * 1024 = 268435456 bytes (Le bloc d'indirection double pointe vers 512 blocs d'indirection simple, chacun pointant vers 512 blocs de données)

Total: 1024 * 7 + 512 * 1024 + 512 * 512 * 1024 = 268966912 bytes ≈ 256 MB

MINIX-FS - Exemple de superblock




```
struct minix_super_block {
    u16 s_ninodes = 6848;         // total number of inodes
    u16 s_nzones = 20480;         // total number of blocks
                                  // = (20*1024^2)/1024
    u16 s_imap_blocks = 1;        // inodes bitmap size in blocks
                                  // need 6848 bits
                                  // -> 1 block = 8192 bits
    u16 s_zmap_blocks = 3;        // data blocks bitmap size in blocks
                                  // need 20480 - 220 bits -> 3 blocks = 8192*3 bits
    u16 s_firstdatazone = 220;    // index of first data block
    u16 s_log_zone_size = 0;      // block size = 1024*2^s_log_zone_size
                                  // = 1024*2^0 = 1024
    u32 s_max_size = 268966912;   // max file size in bytes
                                  // = (7+(1024/2)+(1024/2)^2)*1024
    u16 s_magic = 0x137F;         // 0x137f Minix filesystem version 1
    u16 s_state = 1;              // was the FS properly unmounted?
};
```

MINIX-FS

Exemple d'inode pour un fichier répertoire: contenu de l'inode 1, la racine du FS

```
struct minix_inode {
  u16 i_mode = 0x41ED;      // 0x4 = directory
                            // 0x1ED = 755 in octal
                            //       = rwx r-x r-x
  u16 i_uid = 0;            // 0 = root
  u32 i_size = 256;         // 256/16 = 16 dir_entry
  u32 i_time = 1701436374; // seconds since 1/1/1970
  u8 i_gid = 0;             // 0 = root
  u8 i_nlinks = 8;          // 8 dir entries point to
                            // this inode
  u16 i_zone[7] = {220, 0, 0, 0, 0, 0, 0 };  // 1 data block ⚠ index 0 = superbloc
  u16 i_indir_zone = 0;      // no indirect block
  u16 i_dbl_indir_zone = 0; // no double indirect block
};
```

MINIX-FS

Exemple de bloc de données pour le contenu d'un répertoire:
1er bloc de données référencé par l'inode 1
bloc 220 (offset = 220 * 1024 = 0x37000)

```
00037000   01 00 2E 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037010   00 00 79 6F  00 00 00 00  00 00 00 00  00 00 00 00  ..yo............
00037020   01 00 2E 2E  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037030   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037040   02 00 72 6F  6F 74 00 00  00 00 00 00  00 00 00 00  ..root..........
00037050   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037060   03 00 62 69  6E 00 00 00  00 00 00 00  00 00 00 00  ..bin...........
00037070   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037080   04 00 75 73  72 00 00 00  00 00 00 00  00 00 00 00  ..usr...........
00037090   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000370A0   0A 00 65 74  63 00 00 00  00 00 00 00  00 00 00 00  ..etc...........
000370B0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000370C0   0F 00 74 6D  70 00 00 00  00 00 00 00  00 00 00 00  ..tmp...........
000370D0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000370E0   32 00 64 65  76 00 00 00  00 00 00 00  00 00 00 00  2.dev...........
000370F0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
```

MINIX-FS

Numéro d'inode: premiers 2 bytes. Attention: little endian!

Nom du fichier: 14 bytes successifs

Une entrée de répertoire: 16 bytes (une ligne dans l'affichage courant)

```
00037000   01 00 2E 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037010   00 00 79 6F  00 00 00 00  00 00 00 00  00 00 00 00  ..yo............
00037020   01 00 2E 2E  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037030   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037040   02 00 72 6F  6F 74 00 00  00 00 00 00  00 00 00 00  ..root..........
00037050   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037060   03 00 62 69  6E 00 00 00  00 00 00 00  00 00 00 00  ..bin...........
00037070   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037080   04 00 75 73  72 00 00 00  00 00 00 00  00 00 00 00  ..usr...........
00037090   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000370A0   0A 00 65 74  63 00 00 00  00 00 00 00  00 00 00 00  ..etc...........
000370B0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000370C0   0F 00 74 6D  70 00 00 00  00 00 00 00  00 00 00 00  ..tmp...........
000370D0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000370E0   32 00 64 65  76 00 00 00  00 00 00 00  00 00 00 00  2.dev...........
000370F0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
```

MINIX-FS

Les noms . (0x2E) et .. (0x2E2E) pointent sur l'inode 1 (la racine).
6 autres répertoires: root, bin, usr, etc, tmp, dev (inodes 2, 3, 4, 0xA, 0xF, 0x32).
8 dir_entry inutilisées (n° inode à 0x0000)

```
00037000   01 00 2E 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037010   00 00 79 6F  00 00 00 00  00 00 00 00  00 00 00 00  ..yo............
00037020   01 00 2E 2E  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037030   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037040   02 00 72 6F  6F 74 00 00  00 00 00 00  00 00 00 00  ..root..........
00037050   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037060   03 00 62 69  6E 00 00 00  00 00 00 00  00 00 00 00  ..bin...........
00037070   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037080   04 00 75 73  72 00 00 00  00 00 00 00  00 00 00 00  ..usr...........
00037090   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000370A0   0A 00 65 74  63 00 00 00  00 00 00 00  00 00 00 00  ..etc...........
000370B0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000370C0   0F 00 74 6D  70 00 00 00  00 00 00 00  00 00 00 00  ..tmp...........
000370D0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000370E0   32 00 64 65  76 00 00 00  00 00 00 00  00 00 00 00  2.dev...........
000370F0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
```

MINIX-FS

Offset 0x37010: n° inode = 00 00, nom = 79 6F = caractères ASCII "yo"
Cette entrée était donc utilisée pour un fichier ou un répertoire appelé "yo";
une fois supprimée, le n° inode a été mis à 0 mais le nom n'a pas été effacé

```
00037000   01 00 2E 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037010   00 00 79 6F  00 00 00 00  00 00 00 00  00 00 00 00  ..yo............
00037020   01 00 2E 2E  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037030   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037040   02 00 72 6F  6F 74 00 00  00 00 00 00  00 00 00 00  ..root..........
00037050   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037060   03 00 62 69  6E 00 00 00  00 00 00 00  00 00 00 00  ..bin...........
00037070   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037080   04 00 75 73  72 00 00 00  00 00 00 00  00 00 00 00  ..usr...........
00037090   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000370A0   0A 00 65 74  63 00 00 00  00 00 00 00  00 00 00 00  ..etc...........
000370B0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000370C0   0F 00 74 6D  70 00 00 00  00 00 00 00  00 00 00 00  ..tmp...........
000370D0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000370E0   32 00 64 65  76 00 00 00  00 00 00 00  00 00 00 00  2.dev...........
000370F0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
```

MINIX-FS

La commande `hexdump -C` peut être utilisée pour obtenir un dump hexadecimal d'une image disque (avec output similaire)

```
00037000   01 00 2E 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037010   00 00 79 6F  00 00 00 00  00 00 00 00  00 00 00 00  ..yo............
00037020   01 00 2E 2E  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037030   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037040   02 00 72 6F  6F 74 00 00  00 00 00 00  00 00 00 00  ..root..........
00037050   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037060   03 00 62 69  6E 00 00 00  00 00 00 00  00 00 00 00  ..bin...........
00037070   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037080   04 00 75 73  72 00 00 00  00 00 00 00  00 00 00 00  ..usr...........
00037090   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000370A0   0A 00 65 74  63 00 00 00  00 00 00 00  00 00 00 00  ..etc...........
000370B0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000370C0   0F 00 74 6D  70 00 00 00  00 00 00 00  00 00 00 00  ..tmp...........
000370D0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
000370E0   32 00 64 65  76 00 00 00  00 00 00 00  00 00 00 00  2.dev...........
000370F0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
```

MINIX-FS

Exemple d'inode pour un fichier répertoire: contenu de l'inode 2, répertoire `/root`

```
struct minix_inode {
  u16 i_mode = 0x41ED;      // 0x4 = directory
                            // 0x1ED = 755 in octal
                            //       = rwx r-x r-x
  u16 i_uid = 0;            // 0 = root
  u32 i_size = 160;         // 160/16 = 10 dir_entry
  u32 i_time = 1701436512; // seconds since 1/1/1970
  u8 i_gid = 0;             // 0 = root
  u8 i_nlinks = 2;          // 2 dir entries point to
                            // this inode
  u16 i_zone[7] = {221, 0, 0, 0, 0, 0, 0 };  // 1 data block
  u16 i_indir_zone = 0;      // no indirect block
  u16 i_dbl_indir_zone = 0; // no double indirect block
};
```

MINIX-FS

Exemple de bloc de données pour le contenu d'un répertoire:
1er bloc de données référencé par l'inode 2
Bloc 221 (offset = 221*1024 = 0x37400)

```
00037400   02 00 2E 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037410   6C 03 6C 69  6E 75 78 00  00 00 00 00  00 00 00 00  l.linux.........
00037420   01 00 2E 2E  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037430   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037440   09 00 73 79  73 74 65 6D  00 00 00 00  00 00 00 00  ..system........
00037450   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037460   0B 00 68 65  6C 6C 6F 2E  63 00 00 00  00 00 00 00  ..hello.c.......
00037470   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037480   0E 00 2E 62  61 73 68 5F  6C 6F 67 69  6E 00 00 00  ...bash_login...
00037490   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
```

MINIX-FS

Le nom . (0x2E) pointe sur l'inode 2 (lui même)
Le nom .. (0x2E2E) pointe sur le répertoire parent, l'inode 1 (la racine)
Il y a 2 répertoires: "linux" (inode 0x36C) et "system" (inode 0x9)
Il y a 2 fichiers : "hello.c" (inode 0xB) et ".bash_login" (inode 0xE)
Il y a 4 dir_entry inutilisées (inodes à 0)

```
00037400   02 00 2E 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037410   6C 03 6C 69  6E 75 78 00  00 00 00 00  00 00 00 00  l.linux.........
00037420   01 00 2E 2E  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037430   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037440   09 00 73 79  73 74 65 6D  00 00 00 00  00 00 00 00  ..system........
00037450   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037460   0B 00 68 65  6C 6C 6F 2E  63 00 00 00  00 00 00 00  ..hello.c.......
00037470   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00037480   0E 00 2E 62  61 73 68 5F  6C 6F 67 69  6E 00 00 00  ...bash_login...
00037490   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
```

MINIX-FS

Exemple d'inode pour un fichier régulier: contenu de l'inode 11 (0xB), fichier "hello.c"

```
struct minix_inode {
  u16 i_mode = 0x81A4;      // 0x8 = regular file
                            // 0x1A4 = 644 in octal
                            //       = rw- r-- r--
  u16 i_uid = 0;            // 0 = root
  u32 i_size = 63;          // 1 data block (1 KB/block)
  u32 i_time = 1426700385; // seconds since 1/1/1970
  u8 i_gid = 0;             // 0 = root
  u8 i_nlinks = 1;          // 1 dir entry points to this inode
  u16 i_zone[7] = { 583 0 0 0 0 0 0 };  // 1 data block
  u16 i_indir_zone = 0;      // no indirect block
  u16 i_dbl_indir_zone = 0; // no double indirect block
};
```

MINIX-FS

Exemple de bloc de données pour le contenu d'un fichier régulier:
1er bloc de données référencé par l'inode 11 (0xB)
Bloc 583 (offset = 583*1024 = 0x91C00)

```
00091C00   23 69 6E 63  6C 75 64 65  20 3C 73 74  64 69 6F 2E  #include <stdio.
00091C10   68 3E 0A 0A  76 6F 69 64  20 6D 61 69  6E 28 29 0A  h>..void main().
00091C20   7B 0A 09 09  70 72 69 6E  74 66 28 22  48 65 6C 6C  {...printf("Hell
00091C30   6F 20 57 6F  72 6C 64 5C  6E 22 29 3B  0A 7D 0A 00  o World\n");.}..
00091C40   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00091C50   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00091C60   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00091C70   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00091C80   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00091C90   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00091CA0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00091CB0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00091CC0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00091CD0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
```

MINIX-FS

Exemple d'inode pour un fichier régulier: contenu de l'inode 16 (0x10)

```
struct minix_inode {
  u16 i_mode = 0x81C9;      // 0x8 = regular file
                            // 0x1C9 = 711 in octal
                            //       = rwx --x --x
  u16 i_uid = 0;            // 0 = root
  u32 i_size = 283652;      // 278 data blocks (1 KB/block)
  u32 i_time = 1426700385; // seconds since 1/1/1970
  u8 i_gid = 0;             // 0 = root
  u8 i_nlinks = 2;          // 2 dir entries point to this inode
  u16 i_zone[7] = { 588 589 590 591 592 593 594 };
                             // 7 data blocks
  u16 i_indir_zone = 595;   // block 595 contains 271 pointers
  u16 i_dbl_indir_zone = 0; // no double indirect block
};
```

MINIX-FS

Exemple de bloc de données pour le contenu d'un fichier régulier:
Contenu du bloc indirect (pointeurs d'indirection simple)
référencé par l'inode 16 (0x10)
Bloc 595 (offset = 595*1024 = 0x94C00)

```
00094C00   54 02 55 02  56 02 57 02  58 02 59 02  5A 02 5B 02  T.U.V.W.X.Y.Z.[.
00094C10   5C 02 5D 02  5E 02 5F 02  60 02 61 02  62 02 63 02  \.].^._.`.a.b.c.
00094C20   64 02 65 02  66 02 67 02  68 02 69 02  6A 02 6B 02  d.e.f.g.h.i.j.k.
00094C30   6C 02 6D 02  6E 02 6F 02  70 02 71 02  72 02 73 02  l.m.n.o.p.q.r.s.
00094C40   74 02 75 02  76 02 77 02  78 02 79 02  7A 02 7B 02  t.u.v.w.x.y.z.{.
00094C50   7C 02 7D 02  7E 02 7F 02  80 02 81 02  82 02 83 02  |.}.~...........
00094C60   84 02 85 02  86 02 87 02  88 02 89 02  8A 02 8B 02  ................
00094C70   8C 02 8D 02  8E 02 8F 02  90 02 91 02  92 02 93 02  ................
00094C80   94 02 95 02  96 02 97 02  98 02 99 02  9A 02 9B 02  ................
00094C90   9C 02 9D 02  9E 02 9F 02  A0 02 A1 02  A2 02 A3 02  ................
00094CA0   A4 02 A5 02  A6 02 A7 02  A8 02 A9 02  AA 02 AB 02  ................
...
```

MINIX-FS

Le nombre total d'inodes pouvant être créés dans le FS est indiqué par le champs `ninodes` du superbloc.

La table des inodes possède `ninodes` entrées.
Le premier inode de la table (à l'indice 0) est l'inode 1; il n'y a pas d'inode 0 !
Le bit 0 du bitmap indique l'état de l'inode 0. Le bit 0 est toujours à 1 (utilisé), bien que l'inode 0 n'existe pas.
Donc, un FS vide aura deux bits à 1 dans le bitmap: le bit 0 (inode 0) et le bit 1 (inode 1 = répertoire racine).

Pour lire le contenu de l'inode n, il faut lire l'entrée à l'indice n-1 dans la table des inodes.
Pour savoir si l'inode n est utilisé, il faut lire le bit à l'indice n dans le bitmap des inodes.
Pour lister les inodes il faut lire de l'inode 1 à l'inode `ninodes` inclut.

MINIX-FS

Les n° de blocs dans l'inode sont relatifs au début du FS.
Le bitmap des blocs de données indique uniquement l'état des blocs de données utilisés (donc pas superbloc, bitmaps, etc.)

Le bit 0 du bitmap des blocs de données est à ignorer; il est toujours à 1 et ne représente aucun bloc de données!
Le bit 1 du bitmap correspond au bloc `firstdatazone` indiqué dans le superbloc; il correspond donc au premier bloc de données.

Donc, tout bloc n indiqué dans l'inode correspond à l'indice
(n - `firstdatazone` + 1) dans le bitmap des blocs de données.

Nombre total de blocs de données = (`nzones` - `firstdatazone`)

Sérialisation / déserialisation

Sérialisation de structure = écriture, en une opération, d'une structure sur un fichier, un socket, un pipe, etc.
Effectué typiquement via un appel à `write()` ou `fwrite()`

Déserialisation de structure = lecture, en une opération, depuis un fichier, un socket, un pipe, etc. dans une structure.
Effectué typiquement via un appel à `read()` ou `fread()`

Sérialisation / déserialisation

```c
struct st1 {
    uint8_t tab[7];
    uint16_t n;
    uint32_t big;
    uint8_t x;
    uint8_t y;
};
```

```c
struct __attribute__ ((__packed__)) st2 {
    uint8_t tab[7];
    uint16_t n;
    uint32_t big;
    uint8_t x;
    uint8_t y;
};
```

```c
int main() {
    printf("%ld %ld\n", sizeof(struct st1), sizeof(struct st2));
    return 0;
}
```

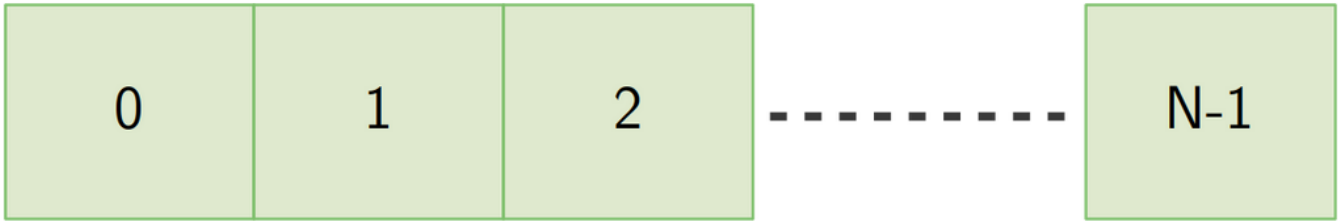
Affichage? 20 15

Pourquoi? Parce que le compilateur ajoute des bytes de padding pour optimiser l'alignement des structures.

Conséquence: toujours utiliser l'attribut `__packed__` sur les structures sérialisées!

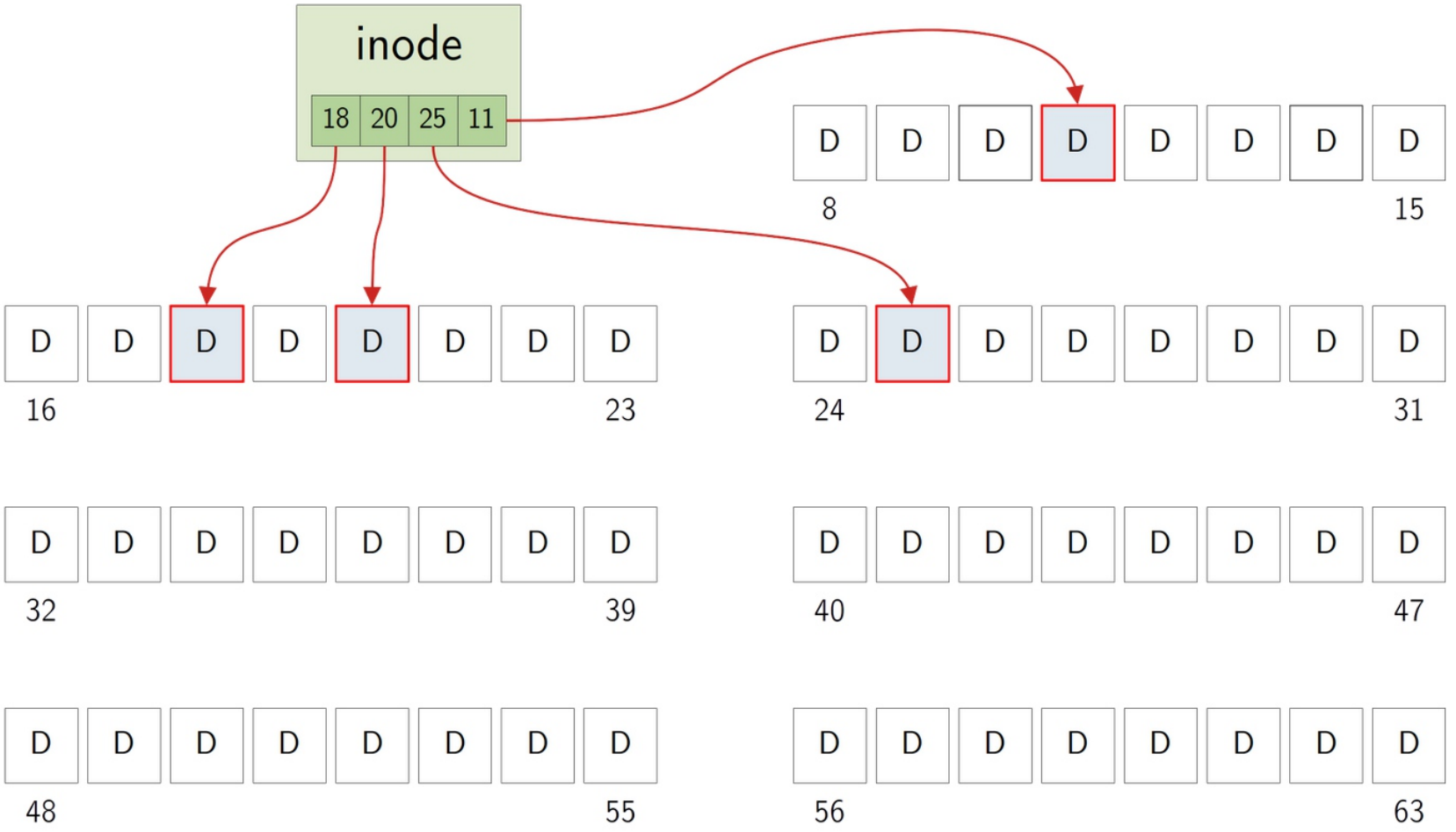
Implémentation: adressage des blocs de données

On désire obtenir un accès ordonné aux blocs qui composent le contenu d'un fichier: début du fichier en bloc 0, suite en bloc 1, etc.



Adressage: problème 1

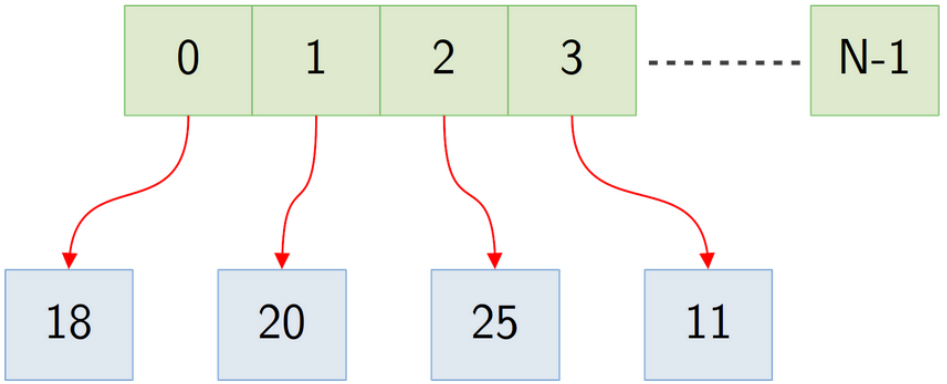
Les blocs de données d'un fichier ne sont pas toujours contigus sur le disque.



Adressage souhaité

On désire une fonction `bmap()` qui mappe un numéro de bloc du fichier (0, 1, 2, ...) vers un numéro de bloc du disque:

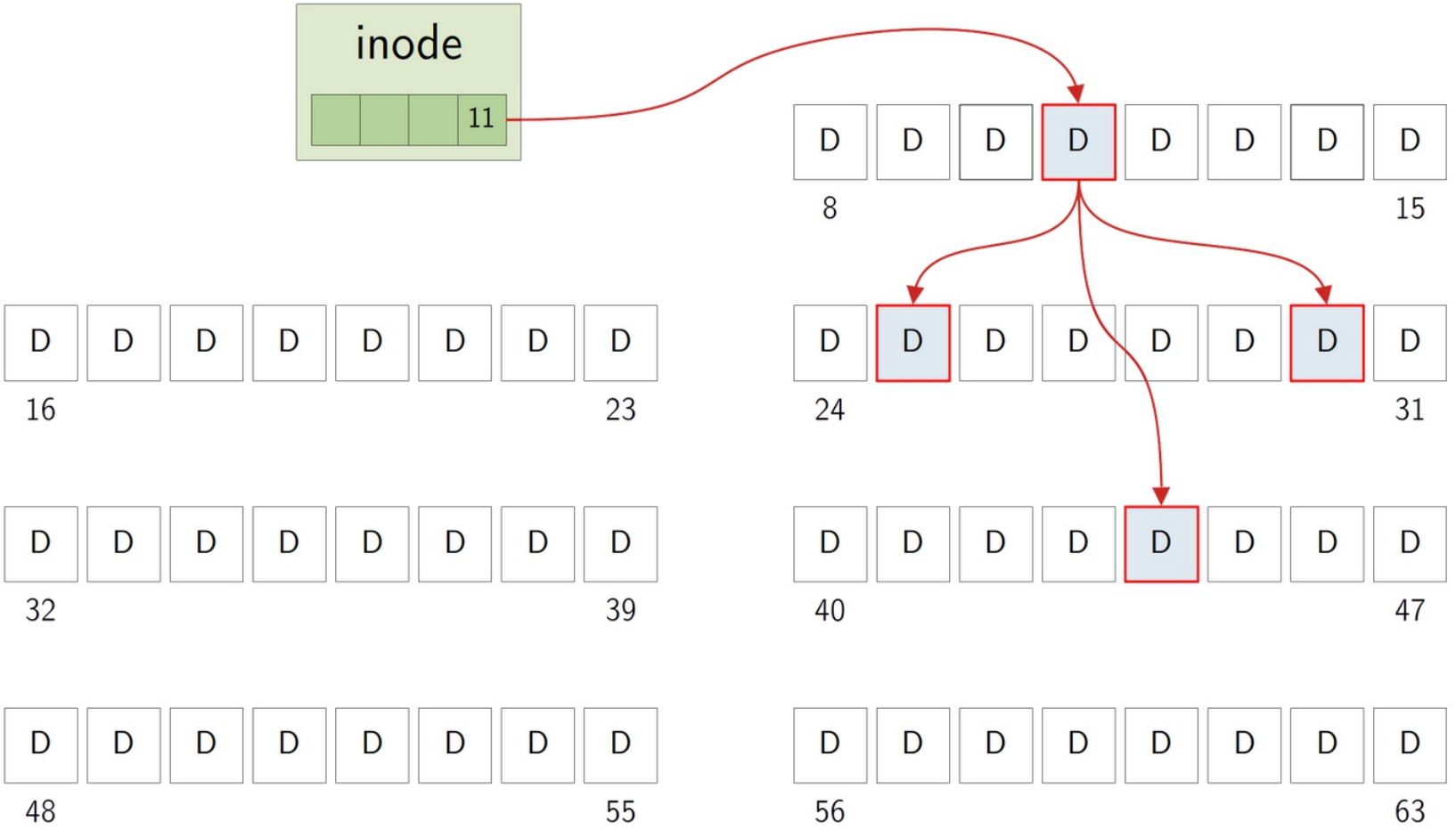
```
// i : numero du bloc (logique) du fichier
// renvoie le numero du bloc physique sur disque
int bmap(inode, i);
```



- `bmap(inode, 0)` → `18`
- `bmap(inode, 1)` → `20`
- `bmap(inode, 2)` → `25`
- `bmap(inode, 3)` → `11`

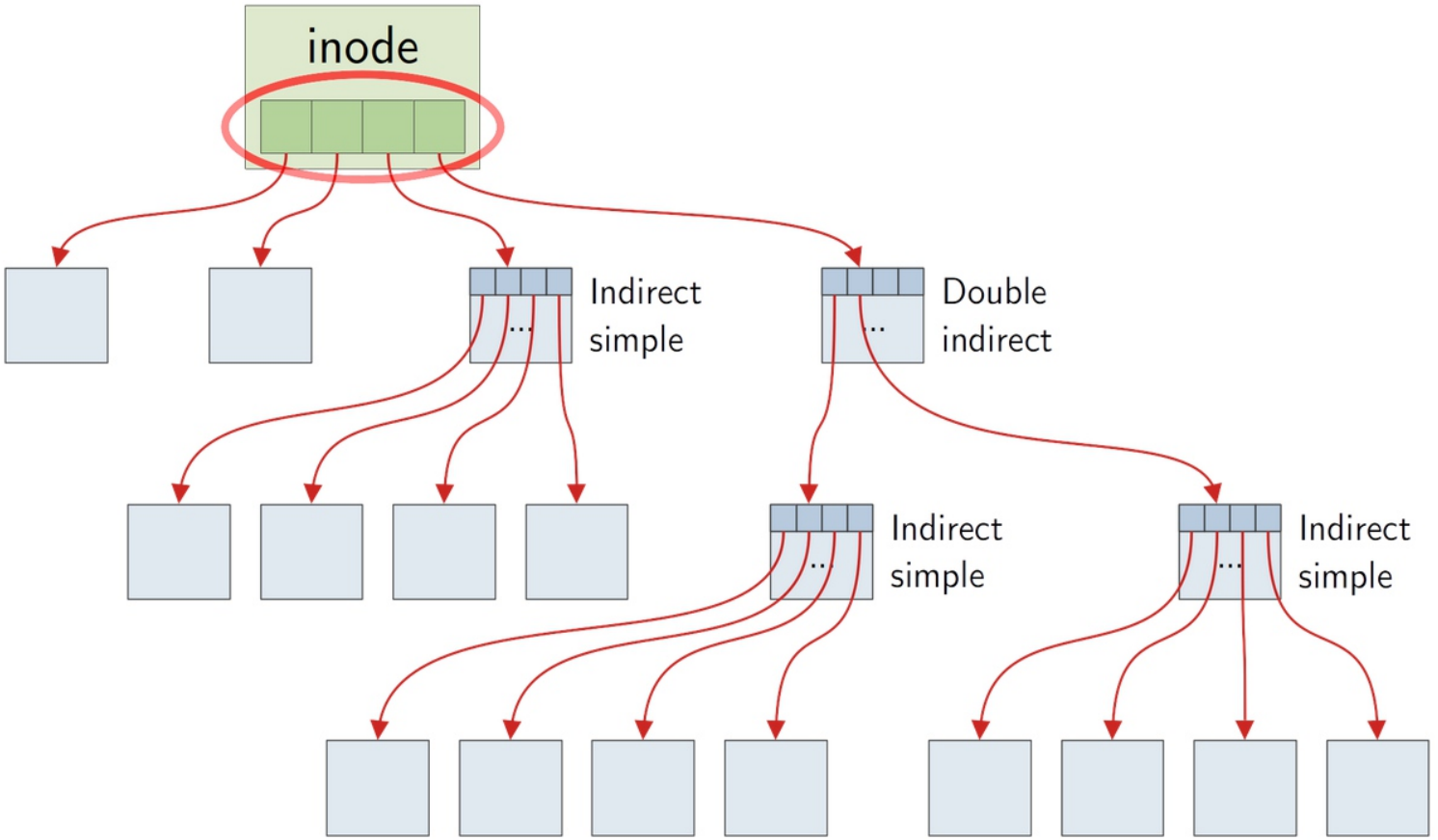
Adressage: problème 2

Certains blocs peuvent être des blocs indirects, c'est à dire contenir les adresses d'autres blocs.



Problème des indirections

Selon le même principe, on peut ajouter des blocs doublement, triplement, etc. indirects. La distinction entre bloc simple, bloc indirect et bloc double indirect se fait sur sa position dans le tableau des adresses de l'inode.



Repenser la structure de données

Ainsi la structure des inodes de MINIX...

```
struct minix_inode {
    ...

    u16 i_zone[7];         // direct pointers
    u16 i_indir_zone;      // indirect pointer
    u16 i_dbl_indir_zone;  // double indirect pointer
};
```

... peut s'écrire...

```
struct minix_inode {
    ...

    u16 i_zone[9];  // i_zone[7] = indirect pointers
                    // i_zone[8] = double indirect pointers
};
```

Algorithme pour une fonction bmap()

Pseudocode avec 2 niveaux d'indirections:

```
if (i < nb_blocs_adressables_directement) {  // cas 1
    return inode.i_zone[i]
}

i = i-nb_blocs_adressables_directement

if (i < nb_blocs_adressables_avec_une_seule_indir) {  // cas 2
    indir_bloc = read_block(inode.i_zone[indice_du_bloc_des_indir_simples])
    return indir_bloc[i]
}

i = i-nb_blocs_adressables_avec_une_seule_indir

if (i < nb_blocs_adressables_avec_une_double_indir) {  // cas 3
    indir_bloc = read_block(inode.i_zone[indice_du_bloc_des_indir_doubles])
    indir2_bloc = read_block(
                      indir_bloc[i/nb_blocs_adressables_avec_une_seule_indir])
    return indir2_bloc[i % nb_blocs_adressables_avec_une_seule_indir]
}
```

Algorithme pour une fonction bmap()

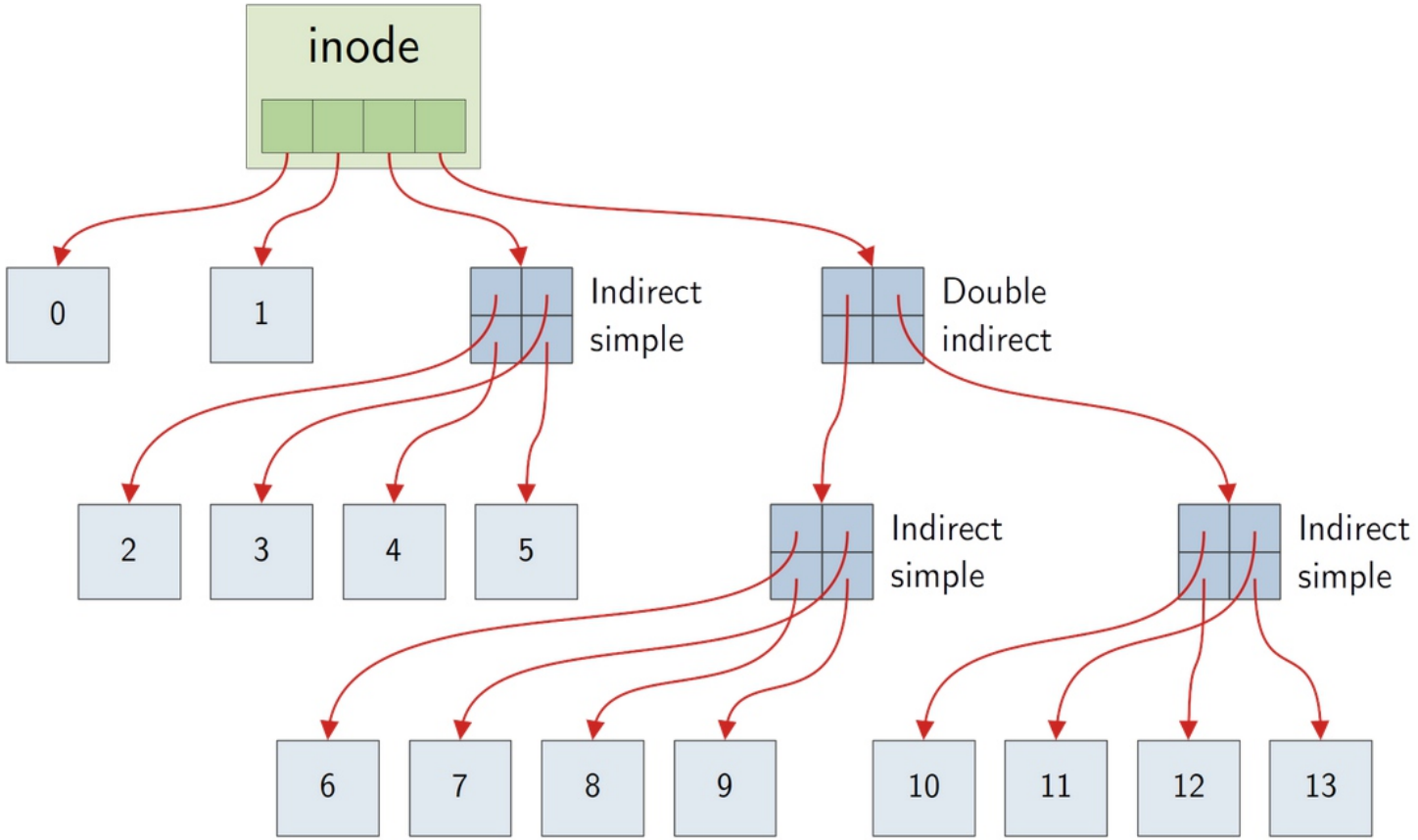
Pourquoi la soustraction entre chaque cas ?

Pour réajuster l'indice dans le bloc lors d'une indirection: on passe d'un indice absolu à un indice relatif au bloc indirect.

Exemple: si 7 blocs sont adressables directement, mais qu'on veut le 8ème, on veut le premier indice du bloc indirect, pas l'indice 8.

Example fonction bmap()

Nombre de blocs adressables directement: 2
Nombre de blocs adressables avec une seule indirection: 4
Nombre de blocs adressables avec une double indirection: 4*4



Example fonction bmap()

Ici les blocs sont indéxés logiquement; en réalité, c'est leur numéro physique qui est présent dans les tableaux de zone.

Example fonction bmap()

```
if (i < 2) {  // cas 1
  return inode.i_zone[i]
}

i = i-2

if (i < 4) {  // cas 2
  indir_bloc = read_block(inode.i_zone[indice_du_bloc_des_indir_simples])
  return indir_bloc[i]
}

i = i-4

if (i < 16) {  // cas 3
  indir_bloc = read_block(inode.i_zone[indice_du_bloc_des_indir_doubles])
  indir2_bloc = read_block(indir_bloc[i/4])
  return indir2_bloc[i % 4]
}
```

Example fonction bmap()

i = 4:

- pas inférieur à 2 → on rejette le cas 1
- 4-2 = 2
- 2 inférieur à 4 → on entre dans le cas 2
- lecture du bloc d'indirection simple
- indice 2 dans bloc d'indirection simple pointe bien vers bloc 4

i = 6:

- pas inférieur à 2 → on rejette le cas 1
- 6-2 = 4
- 4 pas inférieur à 4 → on rejette le cas 2
- 4-4 = 0
- 0 inférieur à 16 → on entre dans le cas 3
- lecture du bloc d'indirection simple d'indice 0/4 = 0
- 0 % 4 (=0) dans bloc d'indirection simple pointe bien vers bloc 6

Example fonction bmap()

i = 9:

- pas inférieur à 2 → on rejette le cas 1
- 9-2 = 7
- 7 pas inférieur à 4 → on rejette le cas 2
- 7-4 = 3
- 3 inférieur à 16 → on entre dans le cas 3
- lecture du bloc d'indirection simple d'indice 3/4 = 0
- indice 3 % 4 (=3) dans bloc d'indirection simple pointe bien vers bloc 9

Example fonction bmap()

i = 10:

- pas inférieur à 2 → on rejette le cas 1
- 10-2 = 8
- 8 pas inférieur à 4 → on rejette le cas 2
- 8-4 = 4
- 4 inférieur à 16 → on entre dans le cas 3
- lecture du bloc d'indirection simple d'indice 4/4 = 1
- indice 4 % 4 (0) dans bloc d'indirection simple pointe bien vers bloc 10

Problème de la résolution du chemin d'accès

Les inodes n'ont pas de noms, juste un numéro.

Comment retrouver le numéro d'un inode à partir de son chemin d'accès dans la chaîne des répertoires du système de fichiers?

Fonction namei()

On désire une fonction `namei()` qui retourne le numéro d'un inode à partir de son chemin d'accès dans le système de fichiers. Cette fonction est utilisée par tous les appels systèmes qui prennent un chemin d'accès en argument.

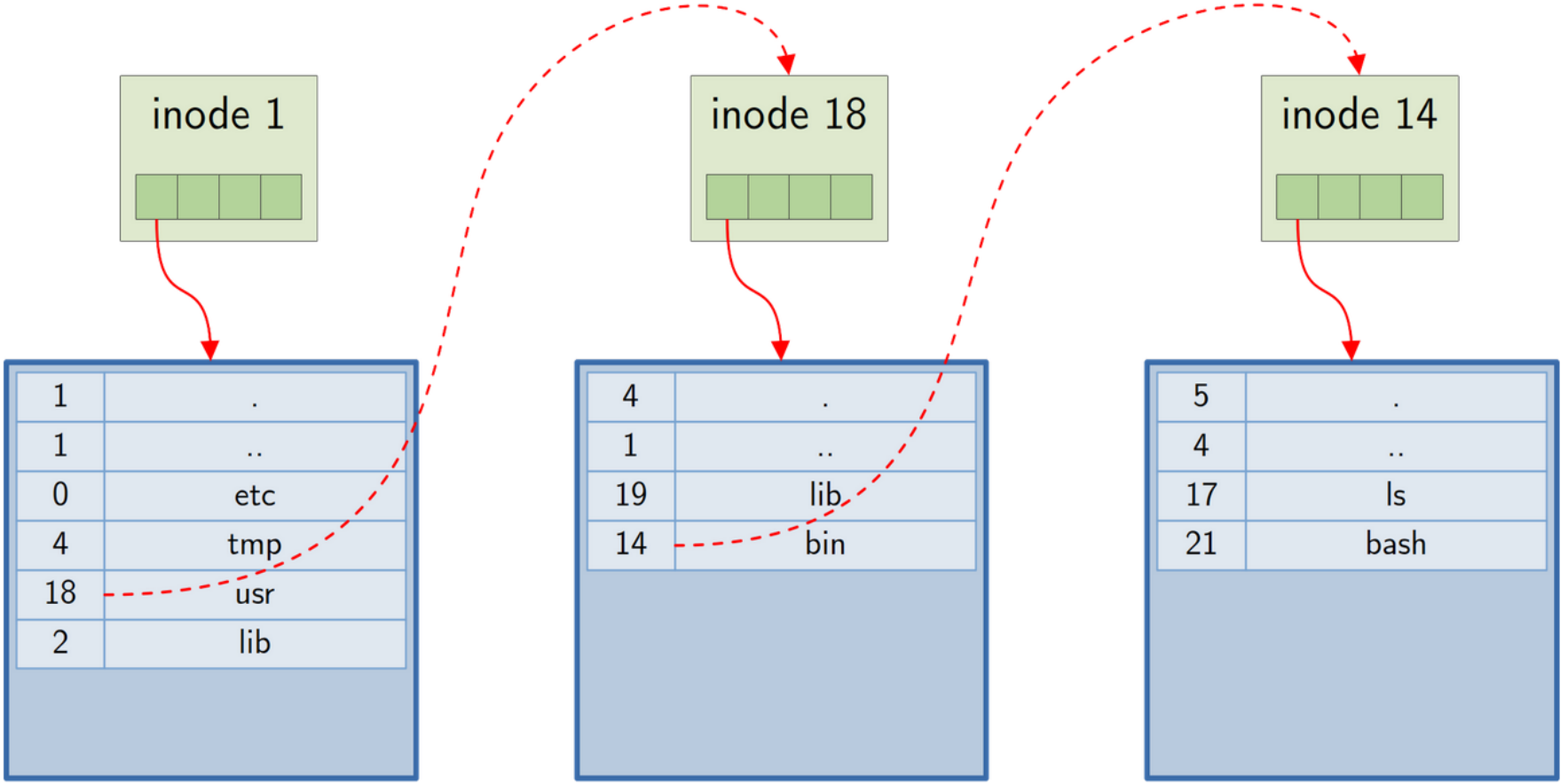
```
// renvoie le numero d'inode du fichier dont le
// chemin d'acces est path
int namei(path);
```

La recherche débute depuis:

- l'inode de la racine, si le chemin d'accès est absolu
- le répertoire courant du processus, si le chemin d'accès est relatif

Example de fonction namei()

Appeler `namei()` sur le chemin "/usr/bin/bash" parcourt le contenu des inodes suivants et renvoie 21:



Algorithme pour la fonction namei()

Pseudocode pour un chemin absolu:

```
int namei(path) {
    inode = ROOT_INODE
    pour chaque composant du path p {
        inode = lookup_entry(inode, p)
    }
    return inode
}
```

Fonction lookup_entry()

La fonction `lookup_entry()` recherche un fichier dans un répertoire et renvoie son inode:

```
// renvoie le numero d'inode correspondant a name
int lookup_entry(dir_inode, name);
```

`lookup_entry()` réalise une recherche linéaire de la chaîne de caractères "name" dans les entrées de répertoire (dir_entry) du répertoire dont l'inode est "dir_inode".

Le contenu du répertoire peut-être réparti sur plusieurs blocs → utiliser `bmap()`

Aucune garantie que la taille du répertoire soit un multiple de la taille des blocs: utiliser le champs taille de l'inode pour limiter la recherche

Algorithme pour la fonction lookup_entry()

```
int lookup_entry(dir_inode, name) {
    pour chaque bloc de dir_inode et dans la limite
    de dir_inode.size lire le prochain dir_entry {
        if (dir_entry.name == name) {
            return dir_entry.inode
        }
    }
    return not_found
}
```

Fonction del_entry()

La fonction `del_entry()` supprime un fichier d'un répertoire.

```
del_entry(dir_inode, name)
```

`del_entry()` réalise une recherche linéaire de la chaîne de caractères "name" dans les entrées de répertoire (dir_entry) du répertoire dont l'inode est "dir_inode".

Une fois l'entrée de répertoire (dir_entry) et son numéro d'inode trouvée:

- le nombre de liens pointant vers l'inode correspondant est décrémenté
- si le nombre de liens tombe à 0 il faut supprimer l'inode (en mettant le bit dans le bitmap d'inode à 0)
- le champ "inode" du dir_entry est mis à 0 et le bloc modifié est écrit sur disque

Tout comme pour `lookup_entry()`, le contenu du répertoire peut-être réparti sur plusieurs blocs

Processus et virtualisation

Processus

Pour vous, qu’est-ce qu’un processus?

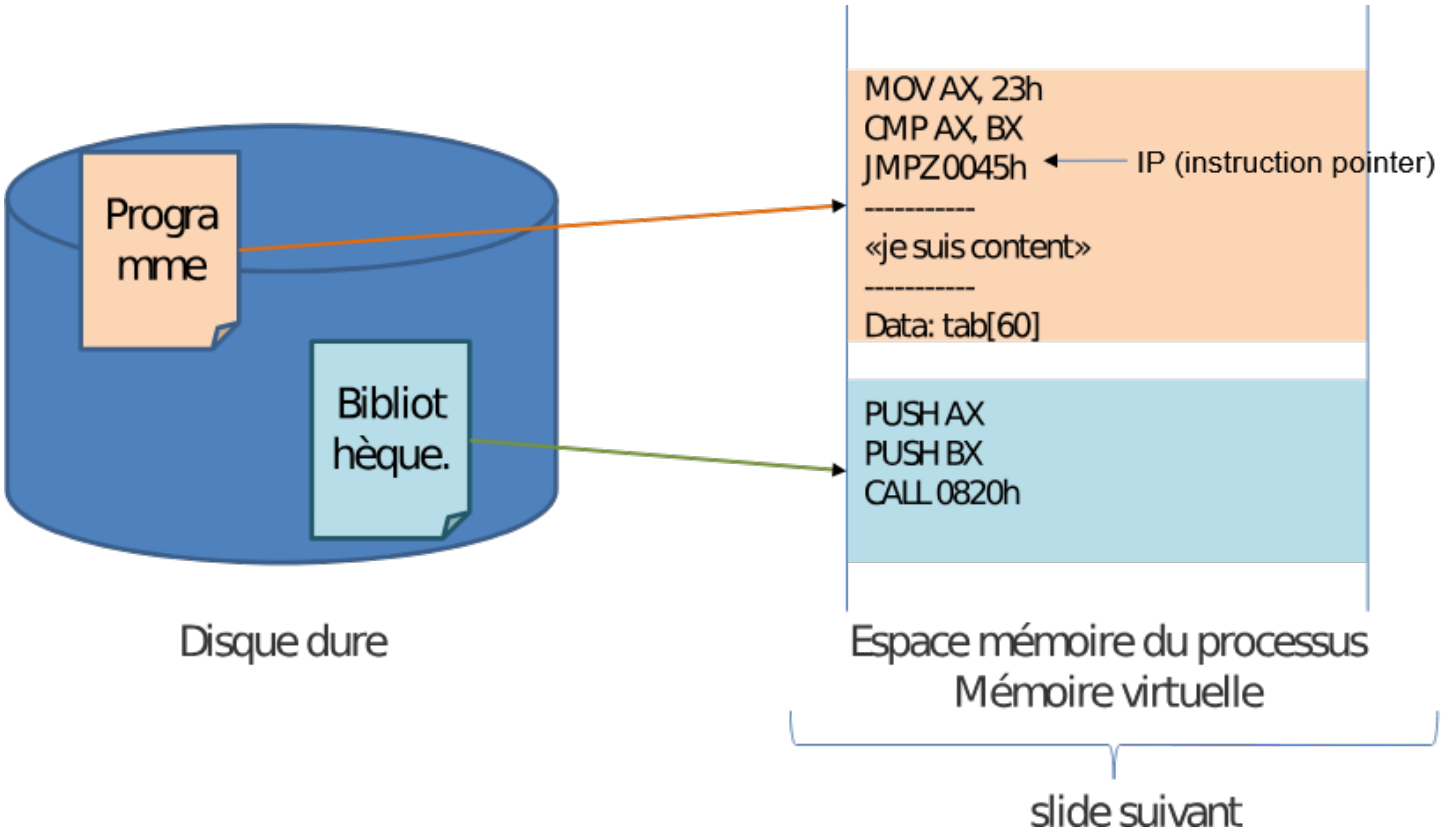
Quelles informations contient-il?

Où réside ces informations en mémoire?

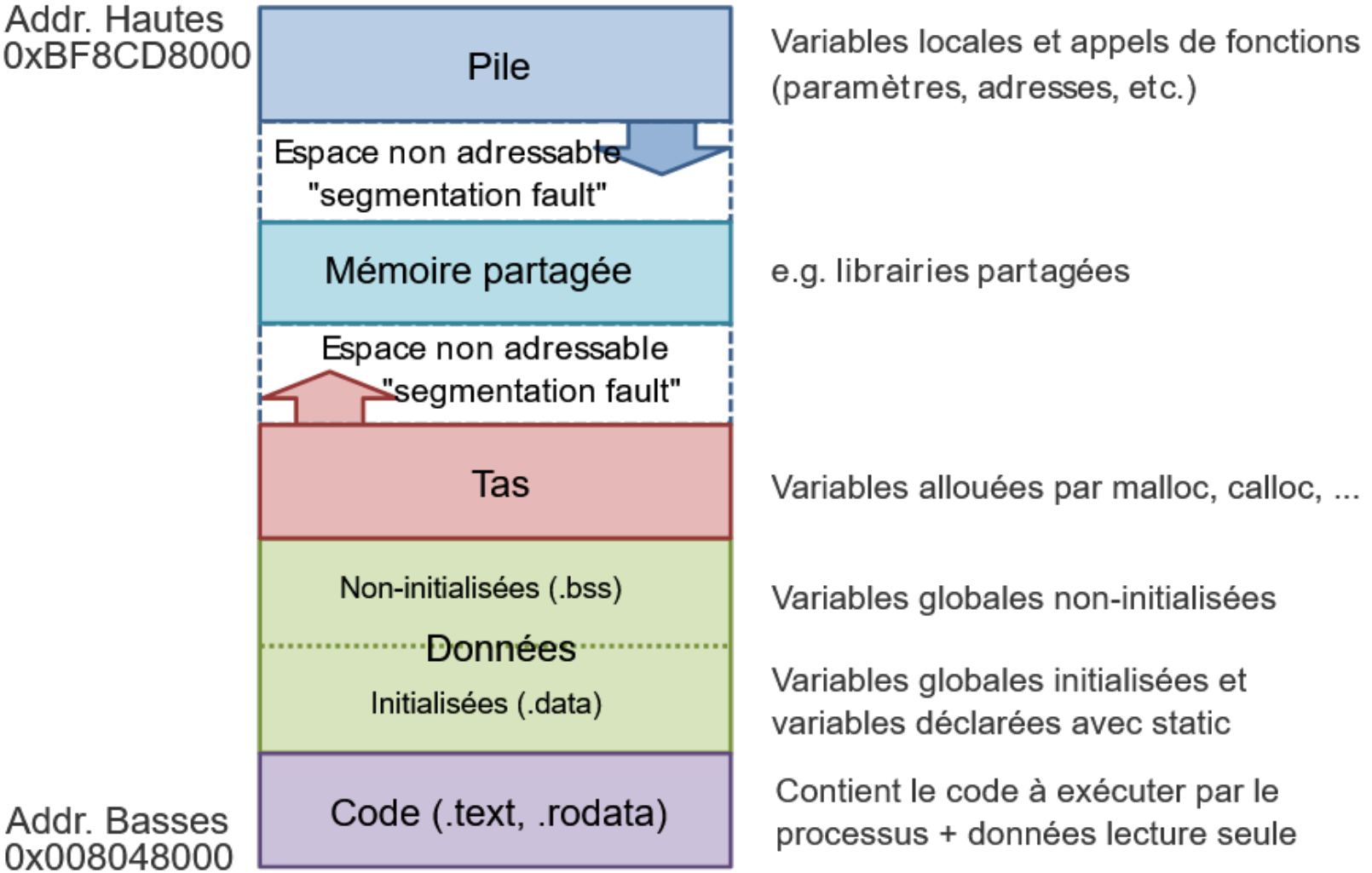
Que permet les systèmes multi-processus ?

Processus

Un processus représente l'exécution courante d'un programme.
Il contient donc toutes les informations nécessaire à l'exécution du programme.



Mémoire virtuelle
Espace d'adressage du processus



Processus

Toute application, programme, script ou service qui s'exécute sur le système est un **processus**. Les segments d'un processus sont appelés **threads**, qui partagent le même espace mémoire.

Chaque processus possède un **PID (Process ID)** unique et un **PPID (Parent Process ID)** : lorsqu'un processus engendre un enfant, le PID du processus est attribué au PPID de l'enfant.
Le processus avec le PID 1 (`init` ou `systemd`) est l'ancêtre de tous les processus. On ne peut pas le tuer; sa mort provoque un kernel panic.

Processus: exercice

Créer un programme en C qui:

- déclare des variables globales
- déclare des variables locales (e.g. dans une fonction)
- utilise la fonction malloc pour allouer de la mémoire
- appelle une fonction autre que la fonction `main()`
- affiche **toutes** les adresses des objects déclarés ci-dessus (y compris les fonctions et les adresses des pointeurs)
- attend une entrée utilisateur ou se met en pause

Processus: solution de l'exercice

```c
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#define MALLOC_SIZE 0x10 //en octets
#define STACK_ALLOC_SIZE 256*1024 //en octets

uint32_t nonInitData; //Variable globale non initialisée
uint32_t initData = 0x12; //Variable globale initialisée
char* ptChar = "Where is this pointed string?"; // ???
char tabChar[] = "Where is this char array?"; // ???

void useStack()
{
  //Augmentation de la pile
  char tab[STACK_ALLOC_SIZE];

  //Pause pour examiner le processus
  printf("Stack size has been increase and the program paused...\n");
  fgetc(stdin);
}

int main()
{
  //Déclaration de variables locales
  uint32_t localVar1, i;
  char *pt_malloc; //pointeur sur la zone allouée

  //Affiche le PID
  printf("My PID = %d\n\n", getpid());

  //Récuperer l'adresse du brk avant et après allocation de mémoire
  pt_malloc = malloc(MALLOC_SIZE);

  //Affichage de l'adresse de la fonction main dans le segment de code
  printf("--- code segment ---\n");
  printf("main function adress: %p\n", main);
  printf("use stack function adress: %p\n", useStack);
  printf("Fixed string adress(ptChar): %p\n", ptChar); //La chaine est constante donc dans .bss

  //Affichage des differentes variables presentes dans le segment de données (d'abord initialisées puis non initialisées)
  printf("--- data segment ---\n");
  printf("initData address: %p\n", &initData);
  printf("Pointer to fixed string address (ptChar adress): %p\n", &ptChar); //Ce pointeur est alloué en donnée globale initialisée
  printf("Character array address (tabChar): %p\n", tabChar); //Le tableau est une variable (avec equivalence de pointeur) -> données initialisées
  printf("--\n");
  printf("nonInitData address: %p\n", &nonInitData);

  //Affichage des données du tas (brk avant et après allocation de mémoire et pointeur sur données allouées)
  printf("--- Heap ---\n");
  printf("Address allocated variable: %p\n", pt_malloc);

  //Affichage des données locales affichées dans la pile
  printf("--- Stack ---\n");
  printf("First Local variable address: %p\n", &localVar1);
  printf("Second Local variable address: %p\n", &i);
  printf("pt_malloc variable address: %p\n", &pt_malloc);
  printf("-------------\n");

  //Pause pour examiner le processus
  fgetc(stdin);

  //Utilisation de la pile et pause
  useStack();

  //Essaie de modifier la chaine de caractères
  ptChar[0] = 'e';

  return EXIT_SUCCESS;
}
```

Processus: solution de l'exercice

```
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#define MALLOC_SIZE 0x10 //en octets
#define STACK_ALLOC_SIZE 256*1024 //en octets

uint32_t nonInitData; //Variable globale non initialisée
uint32_t initData = 0x12; //Variable globale initialisée
char* ptChar = "Where is this pointed string?"; // ???
char tabChar[] = "Where is this char array?"; // ???
```

Processus: solution de l'exercice

```
void useStack()
{
  //Augmentation de la pile
  char tab[STACK_ALLOC_SIZE];

  //Pause pour examiner le processus
  printf("Stack size has been increase and the program
paused...\n");
  fgetc(stdin);
}
```

Processus: solution de l'exercice

```
int main()
{
  //Déclaration de variables locales
  uint32_t localVar1, i;
  char *pt_malloc; //pointeur sur la zone allouée

  //Affiche le PID
  printf("My PID = %d\n\n", getpid());

  //Récuperer l'adresse du brk avant et après allocation de
mémoire
  pt_malloc = malloc(MALLOC_SIZE);
```

Processus: solution de l'exercice

```
  //Affichage de l'adresse de la fonction main dans le
segment de code
  printf("--- code segment ---\n");
  printf("main function adress: %p\n", main);
  printf("use stack function adress: %p\n", useStack);
  printf("Fixed string adress(ptChar): %p\n", ptChar); //La
chaine est constante donc dans .bss
```

Processus: solution de l'exercice

```
  //Affichage des differentes variables presentes dans le
segment de données (d'abord initialisées puis non
initialisées)
  printf("--- data segment ---\n");
  printf("initData address: %p\n", &initData);
  printf("Pointer to fixed string address (ptChar adress):
%p\n", &ptChar); //Ce pointeur est alloué en donnée globale
initialisée
  printf("Character array address (tabChar): %p\n",
tabChar); //Le tableau est une variable (avec equivalence
de pointeur) -> données initialisées
  printf("--\n");
  printf("nonInitData address: %p\n", &nonInitData);
```

Processus: solution de l'exercice

```
  //Affichage des données du tas (brk avant et après
allocation de mémoire et pointeur sur données allouées)
  printf("--- Heap ---\n");
  printf("Address allocated variable: %p\n", pt_malloc);
```

"brk" est le "program break" ou "breakpoint", c'est à dire l'adresse qui marque la fin du segment de données d'un processus.
Il représente la première adresse après la fin du segment de données non initialisées (BSS), donc le début du tas (*heap*).
Il définit essentiellement la limite entre la partie allouée et non allouée de l'espace d'adressage du processus.

Processus: solution de l'exercice

```
//Affichage des données locales affichées dans la pile
printf("--- Stack ---\n");
printf("First Local variable address: %p\n", &localVar1);
printf("Second Local variable address: %p\n", &i);
printf("pt_malloc variable address: %p\n", &pt_malloc);
printf("-------------\n");

//Pause pour examiner le processus
fgetc(stdin);

//Utilisation de la pile et pause
useStack();
```

Processus: solution de l'exercice

```
  //Essaie de modifier la chaine de caractères
  ptChar[0] = 'e';

  return EXIT_SUCCESS;
}
```

Pourquoi l'instruction `ptChar[0] = 'e';` produit un segfault?
Parce que il n'est pas permis de modifier un pointer à une chaîne de caractères, qui est stocké en mémoire read-only!
Il faudrait donc remplacer

```
char* ptChar = "Where is this pointed string?";
```

par

```
char ptChar[] = "Where is this pointed string?";
```

dans la déclaration des variables globales.

Processus: pmap

En utilisant la commande `pmap`:

- observez les différents segments du processus
- comparez les adresses des segments avec les adresses des variables de votre programme
- confirmez la bonne répartition des données dans les segments

Processus: pmap

Champs de `pmap -X <pid>`:

Address: start address of memory map of the process
Kbytes: size of map in kilobytes
RSS: Resident Set Size in kilobytes = the amount of memory that is currently in RAM, and not swapped out
PSS: Proportional Set Size = the count of pages the process has in memory, where each page is divided by the number of processes sharing it. So if a process has 1000 pages all to itself, and 1000 shared with one other process, its PSS will be 1500
Dirty: dirty pages (both shared and private) in kilobytes = memory that has been changed since the process (and the mapping) started
Mode: permissions on map: read, write, execute, shared, private (copy on write)
Mapping: file backing the map, or '[anon]' for allocated memory, or '[stack]' for the program stack
Offset: offset into the file
Device: device name (major:minor)

Signaux

Le noyau utilise des **signaux** pour communiquer avec un processus.

Les signaux sont également utilisés pour la communication entre les processus (*IPC, inter-process communication*); cependant, c'est toujours le noyau qui vérifie si un processus est autorisé à envoyer un signal spécifique à un autre processus.

Signaux

Most common signals		
Signal number	**Signal name**	**Meaning**
1	SIGHUP	Reload your configuration
2	SIGINT	Interrupt execution
3	SIGQUIT	Interrupt execution and produce a core dump. User-initiated
4	SIGILL	Illegal instruction
5	SIGTRAP	Debugger breakpoint
6	SIGABRT	Abort the program due to a critical error
7	SIGBUS	Bus error. Also caused by any general device fault
8	SIGFPE	Floating point exception, or any other fatal arithmetic error
9	SIGKILL	Kill unconditionally (this signal cannot be ignored)
10	SIGUSR1	User-defined signal 1
11	SIGSEGV	Segmentation fault
12	SIGUSR2	User-defined signal 2
13	SIGPIPE	You attempted to write to a pipe without a process connected to the other end
14	SIGALRM	A time interval specified in a call to the alarm function has expired
15	SIGTERM	Terminate gracefully
16	SIGSTKFLT	Stack fault on coprocessor
17	SIGCHLD	Your child is stopped or terminated, please reap it
18	SIGCONT	Continue execution
19	SIGSTOP	Stop execution (this signal cannot be ignored)
20	SIGTSTP	Stop execution

Signaux

loop.sh:

```
#!/bin/bash
echo "My PID is $$"
while true
do
   date
   sleep 3
done
```

Signaux

Lancez `loop.sh` dans un terminal et `top` dans un autre.
Filtrez l'output de `top` (touche o, "COMMAND=loop").
Regardez l'état de ce processus.

Depuis un autre terminal, envoyez des signaux via `kill` à ce processus:
SIGSTOP, SIGCONT, SIGKILL, etc.
(Syntaxe: `kill [-`*`signal`*`|-s `*`signal`*`] `*`pid`*`|`*`name`*)

Examinez l'output des commandes suivantes:

```
ps -ef | grep loop
pstree -p | grep loop
```

Swap

Le **swap** est une zone sur le disque (fichier ou partition) utilisée comme extension de la mémoire physique (RAM).
Lorsqu'il n'y a pas assez de RAM disponible pour un processus, les pages inactives en mémoire sont temporairement transférées de la mémoire vers le disque (**swap out**), pour être ensuite réintégrées dans la mémoire (**swap in**) lorsque les ressources RAM sont à nouveau disponibles. Si la RAM et swap sont presque pleins, le système peut se retrouver engorgé en passant tout son temps à transférer des blocs de mémoire entre la RAM et le swap (**thrashing**).

Sous Linux, on utilise une partition de swap plutôt qu'un fichier de swap.
Un fichier de swap ne peut pas être utilisé pour l'hibernation, pourquoi?
Parce que le système doit d'abord localiser le header du fichier de swap -> le système de fichiers contenant le fichier de swap doit être monté, mais les systèmes de fichiers journalisés (p.ex. ext3 ou ext4) ne peuvent pas être montés pendant le resume.

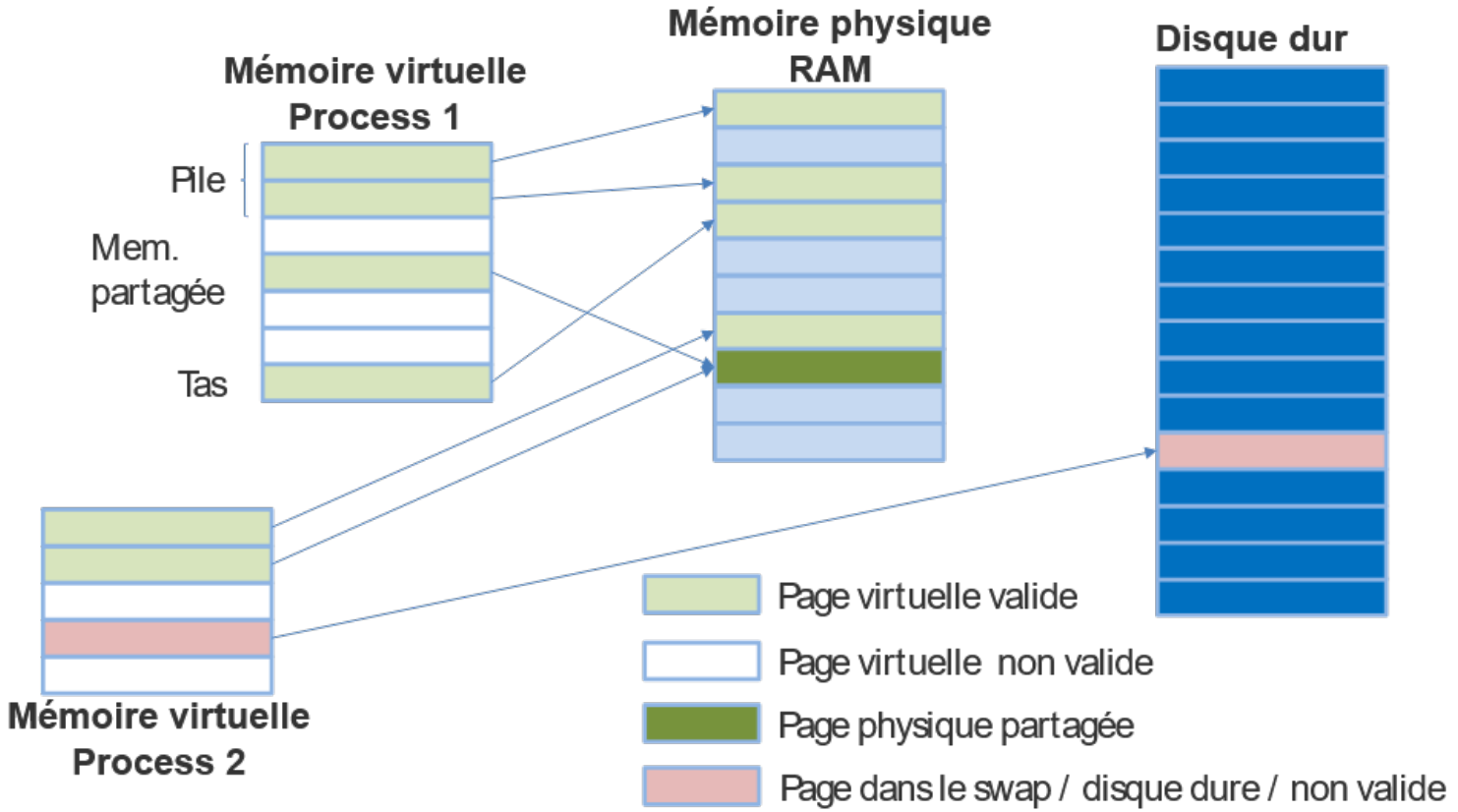
Mémoire virtuelle

Grâce à la mémoire virtuelle on va pouvoir:

- définir un espace d’adressage indépendant pour chaque processus;
- adresser plus de mémoire que la mémoire physique disponible;
- partager facilement des zones de mémoire entre processus;
- adresser le contenu de fichiers comme s’ils étaient en mémoire.

Mémoire virtuelle

L'espace d'adressage est divisé en **pages** (en général de 4Ko).
Une page virtuelle peut être associée à une page de mémoire vive (**page valide**) ou morte (**page invalide**).



Mémoire virtuelle

La conversion entre adresse virtuelle et adresse physique est réalisée par le **MMU (Memory Management Unit)**, hardware.

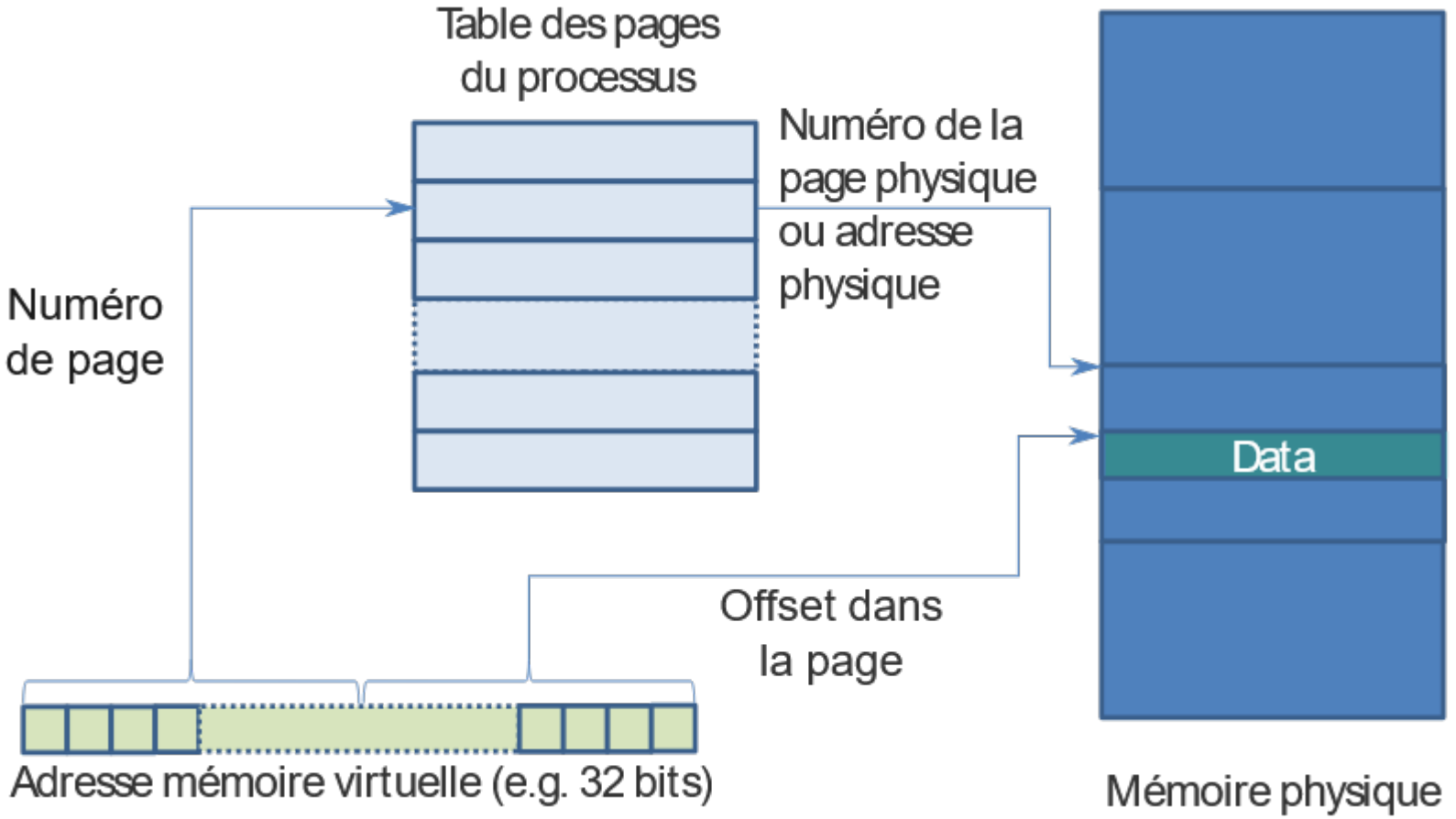


Table des pages (*Page table*)

Une **table des pages** existe pour chaque processus.

Chaque table est maintenue par le système (i.e. Linux, MacOS, Windows, etc…) et utilisée par le MMU.

Quelques informations généralement contenues dans une entrée de la table:

- numéro de page physique;
- taille d'une page;
- permissions d'accès;
- bit «page valide» ou «page présente en RAM»;
- bit «page sale» (i.e. modifiée depuis sa dernière présence sur disque);
- etc.

Défaut de page (*Page fault*)

Un **défaut de page** arrive lorsque le MMU ne peut pas satisfaire une demande de page car elle n'est pas référencée dans la table du processus (bit «page valide» = false).

Il y a alors 3 cas possibles:

- L'accès mémoire est illégal -> le noyaux termine le processus en «segmentation fault» (SIGSEGV);
- La page est présente en mémoire physique, c'est un **défaut de page mineur** -> il suffit de mettre à jour la table du processus pour la faire pointer sur la page en mémoire physique;
- La page n'est pas présente en mémoire physique, c'est un **défaut de page majeur**.

Défaut de page majeur

Pour un défaut de page majeur il faut charger la page manquante:

- on sauvegarde l'état du processus et on le met «en attente»;
- si il n'y a pas de place en mémoire physique on libère une page peu utilisée;
- on charge la page manquante en mémoire depuis le disque;
- on met à jour la table des pages du processus;
- on charge l'état du processus et on repart de l'instruction ayant provoqué la faute de page (cette fois satisfaite).

Une page est libérée en mémoire physique quand soit:

- elle existe déjà sur le disque car elle n'a pas été modifié (i.e. bit «page sale» = 0), dans ce cas il suffit de remplacer cette page physique par la nouvelle
- elle à été modifiée et est mise en swap pour conserver les modifications.

Processus: exercice

En utilisant la commande `pmap -X <pid>` sur le processus précédent, observer et expliquer les champs Size, RSS, PSS et Swap.

Comment ces champs évoluent-t-ils lorsque plusieurs processus identiques sont lancés ?

Structure d'un processus

Un processus est identifié grâce à son PID (Process ID). Il est unique pour chaque processus mais un PID libéré peut être réutilisé.

Chaque processus est décrit par son contexte:

- l'état du processeur qui l'exécute:
  - les registres accessibles au programme
  - l'instruction courante (compteur ordinal)
  - les informations de pagination (tables des pages...)
- son espace mémoire virtuel -> les données et le programme;
- les ressources dont il dispose;
- des informations administratives:
  - PID, utilisateur(s), Session ID, Groupe ID
  - priorités (statique et dynamique)
  - consommation de ressources

Structure d'un processus

Vous pouvez examiner l'état d'exécution d'un processus et les appels système qu'il est en train d'effectuer au moyen de la commande `strace`.

Dans `/proc/`*`pid`* vous trouverez toutes les informations sur le processus *pid*. En particulier, lancez la commande `cat /proc/`*`pid`*`/status`.

Structure d'un processus

Dans le noyaux Linux (3.7.10), un processus est défini par la structure `task_struct` (/usr/src/linux/include/linux/sched.h).

```
struct task_struct {
        /* ... */
        /* PID du processus */
        pid_t pid;

        /* Description de la mémoire virtuelle + table de page */
        struct mm_struct *mm;

        /* Etat du CPU / registres (specifique à la platforme) */
        struct thread_struct thread;

        /* Information sur l'ordonnancement du processus */
        struct sched_info sched_info;

        /* Contient notament la table des descripteur de fichier ainsi
        qu'une liste des descripteurs "close on-exec" */
        struct files_struct *files;

        /* ... (+ de 350 lignes au total) */
};
```

Structure d'un processus

Dans le noyaux Linux (3.7.10), un processus est défini par la structure `task_struct` (dans /usr/src/linux/include/linux/sched.h).

```
/* Dans /usr/src/linux/include/linux/mm_types.h */
struct mm_struct {
        /* ... */
        unsigned long start_code, end_code, start_data, end_data; /* segments de code / données */
        unsigned long start_brk, brk, start_stack; /* segment du tas et de la pile */
        /* ... */
}
```

Création de processus

Lors du démarrage du système, le processus `init` (ou `systemd`) est créé par le noyau. Il est donc le premier processus et porte le PID 1.

Tous les autres processus sont créés par un appel à la fonction `fork()`. Chaque processus a donc un parent (excepté `init`/`systemd`); pour voir l'arbre des processus, lancez la commande `pstree -p -C age`.

```c
#include <unistd.h>
#include <sys/types.h>
pid_t fork(void); // Crée un nouveau processus enfant
pid_t getpid(void); // retourne le PID du processus
pid_t getppid(void); // retourne le PID du parent
```

Création de processus

La fonction `fork()` crée un nouveau processus qui est une réplique du processus parent (e.g. copie de la table des pages, état du processeur, descripteurs de fichier, etc.) et va continuer son exécution à partir du fork.

La fonction `fork()` retourne 0 pour le processus enfant, le PID de l'enfant dans le processus parent, ou -1 en cas d'erreur.

Création de processus

L'implémentation d'un fork peut donc se faire de la manière suivante:

```
#include <unistd.h>
pid_t pid = fork()
if(pid > 0) {
        // Code du parent
}
else if(pid == 0){
        // Code de l'enfant
}
else // Error
```

Le processus enfant n'est pas une réplique exacte du parent (see `man fork`), notamment:

- l'enfant a son propre PID et son PPID est égale au PID du parent;
- pas d'héritage des verrous mémoire et fichiers (mlock, flock).

Création de processus

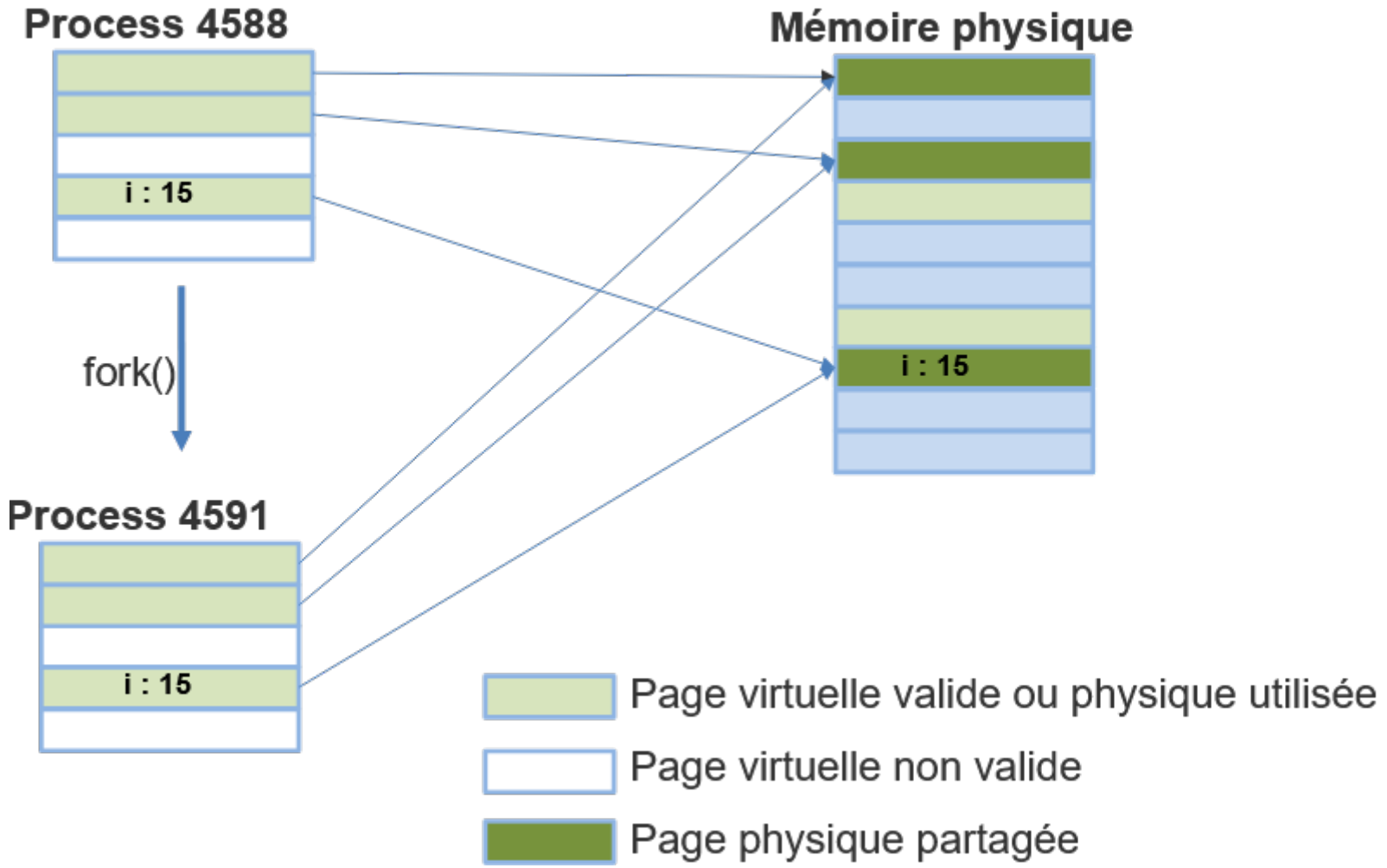
La table des descripteurs de fichier est copiée après un fork.

En conséquence, que se passe-t-il si le processus parent écrit sur le même descripteur que le processus enfant ?

Lorsque le parent et l'enfant écrivent simultanément via le même descripteur de fichier, leurs opérations sont redirigées vers le même fichier ou ressource.
Ils partagent donc le même pointeur de position dans le fichier: l'écriture dans l'un des processus avancera le pointeur utilisé par les deux. Les écritures peuvent donc se mélanger ou se suivre, selon l'ordre dans lequel le kernel les traite.

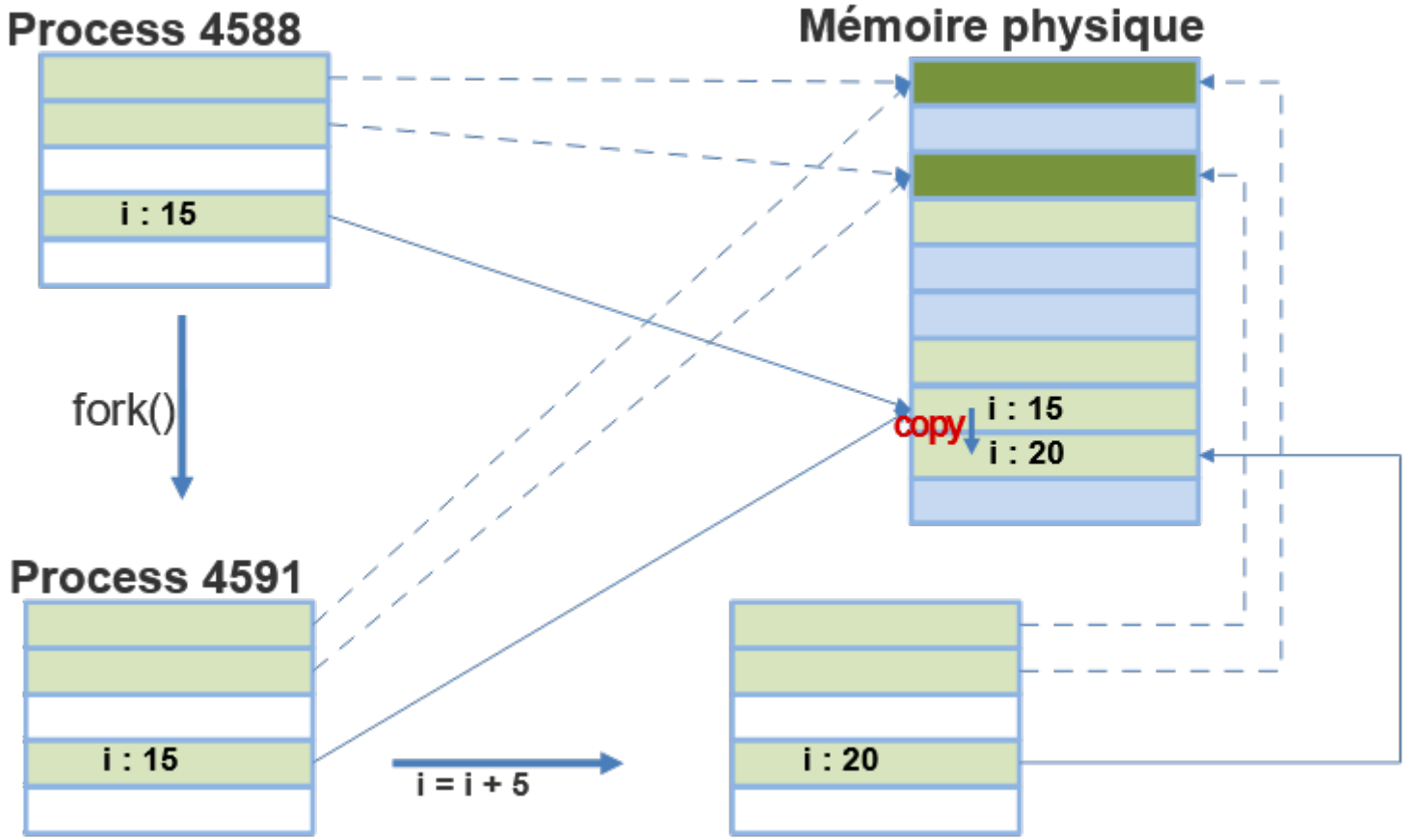
Fork et mémoire virtuelle

Le nouveau processus va donc partager des pages avec son processus parent.



Fork et mémoire virtuelle

Ces pages seront copiées uniquement lors de modifications de la mémoire.



C’est ce que l’on appelle le «**copy on write**».

Fork bomb (en langage C)

Do not try this at home!

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

int main()
{
    while(1)
        fork();
    return 0;
}
```

Fork bomb (en shell script)

Do not try this at home!

```
:(){ :|:& };:
```

ou, en termes plus lisibles,

```
toto() {
    toto | toto &
}
toto
```

Terminaison de processus

Pour terminer un processus à tout moment: `exit(int status);`

Il existe deux constantes souvent utilisées: EXIT_SUCCESS et EXIT_FAILURE.

Avant de terminer le processus la fonction `exit()`:

- ferme les descripteurs de fichiers ouverts (inclus STDIN, STDOUT, STDERR);
- envoie le signal SIGCHLD au parent pour l'informer de la mort de l'enfant;
- tous les enfants du processus deviennent enfant du processus 1 (systemd ou init), on dit alors qu'ils sont orphelins;
- appelle les fonctions enregistrées par `atexit()`.

Autres fonctions pour terminer un processus:

```
void _exit(int status); // appel système direct,  sans appel aux fonction enregistrées avec atexit
void abort(void); // génération d'un core dump
```

Processus zombies

Lorsqu'un processus se termine, le noyau garde certaines informations de la task_struct (pid, statut de terminaison, etc.). On dit alors que le processus est un **zombie**.

Ces information sont conservées en mémoire tant que le parent du processus n'y a pas accédé.

Dans le cas où le parent du processus est mort, c'est le processus avec PID 1 (`init` ou `systemd`) qui va se charger de détruire la task_struct. Il devient donc le nouveau parent du processus orphelin.

Processus zombies

Un processus zombie est donc un processus dont l'exécution a pris fin, mais dont le processus parent, pour une raison quelconque, n'a pas réussi à récupérer les ressources.
Lorsqu'un processus enfant meurt, son statut devient EXIT_ZOMBIE et un SIGCHLD est envoyé au processus parent. Le processus parent doit alors appeler la fonction système `wait()` pour lire le statut de sortie du processus mort et d'autres informations ; jusqu'à ce moment, le processus enfant reste un zombie.

Processus zombies

Les processus zombies ne consomment pas de ressources système et ne posent généralement pas de problème, mais peuvent être le symptôme d'une programmation négligée dans le code du programme parent ou de bugs du système.

Pour éliminer un zombie, on doit terminer son parent en lui envoyant un SIGKILL; le zombie orphelin sera alors récupéré par le processus PID 1.

Il est également possible d'envoyer un SIGCHLD au parent pour lui demander de récupérer l'enfant. Toutefois, cela ne fonctionne pas toujours, car le parent peut ignorer les signaux SIGCHLD, ou ne pas avoir de *signal handler* (gestionnaire de signal) approprié pour eux.

Processus zombies

Lorsqu'un processus effectue un fork, il doit donc prendre soins d'éviter les zombies en appelant une des fonctions suivantes:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Ces fonctions permettent d'attendre la terminaison d'un enfant pour récupérer son statut. Si un enfant est déjà terminé (i.e. est un zombie), ces fonctions retournent immédiatement.

Plusieurs macros permettent de tester le statut de retour, (voir `man wait`) dont:

WIFEXITED(status) : indique si l'enfant s'est terminé normalement;
WCOREDUMP(status) : indique si un core dump de l'enfant a été créé.

Processus zombies: questions

Un processus orphelin peut-il rester un zombie longtemps?

Dans quels cas un processus peut rester un zombie longtemps ?

Processus zombies: exercice

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();
    if (pid == -1) {
        printf("error\n");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        printf("Child: I am %d\n", getpid());
        printf("Child: My parent is %d\n", getppid());
        exit(0);
    }
    else {
        printf("Parent: I am %d and will start sleeping\n", getpid());
        sleep(60);
        printf("Parent: I have finished sleeping and will now exit\n");
    }

    return 0;
}
```

Processus zombies: exercice

Le programme précédent, `zombie.c`, crée un child process zombie.
Examinez l'exécution du programme via `pmap -X <pid>` et via `top` en filtrant l'output (touche o, "COMMAND=zombie").

Ensuite, ajoutez les commandes suivant dans le code du parent pour éviter qu'il crée un child process zombie:

```
// Call wait() on the child process to reap it
int status;
wait(&status);
```

Re-compilez et re-examinez l'exécution du programme.

Exécution de processus

L'execution d'un nouveau programme se fait par les fonctions `exec...()`, dont:

```
#include <unistd.h>
int execve(const char *filename, char *const argv[], char *const envp[]);
```

Cette fonction ne retourne pas de valeur en cas de succès mais elle:

- retourne -1 en cas d'erreur (et met à jour la variable globale `errno`);
- remplace les segments du processus courant par les segments de l'éxécutable `filename` (voir slides sur les fichiers ELF);
- les paramètres `argv` et `envp` sont disponibles dans le main du programme appelé.

Si filename est un script, le shell correspondant est chargé et le fichier exécuté par le shell.
C'est donc cette fonction qui se charge de construire l'espace de mémoire virtuel d'un processus à partir du fichier exécutable.

execve(): exemple

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

int main(int argc, char *argv[], char*env[]) {
    int i;
    for(i=1; i < argc; i++) {
        pid_t pid;
        if((pid = fork()) == 0) {
            char *new_argv[] = {argv[i], NULL};
            if(execve(argv[i], new_argv, env) == -1) {
                perror(argv[i]);
                exit(EXIT_FAILURE);
            }
            printf("This process was succesfully spawned");
            //WILL THE LINE ABOVE BE EXECUTED ?
        }
    }
    return 0;
}
```

execve(): exemple

Le programme commence par une boucle qui parcourt tous les arguments de la ligne de commande, sauf le premier (qui est le nom du programme lui-même).

Pour chaque argument (`argv[i]`), le programme crée un nouveau processus fils en appelant `fork()`.
Si on est dans le processus fils (valeur de retour de fork = 0), le code suivant dans le bloc `if` est exécuté.

Dans chaque processus fils, le programme prépare un tableau de chaînes de caractères contenant uniquement le nom du programme à lancer, puis appelle `execve()` pour essayer d'exécuter ce programme avec son propre nom comme argument, et les variables d'environnement héritées.

execve(): exemple

Si le programme ciblé existe et est exécutable, le processus fils est totalement remplacé par ce nouveau programme : tout son code et sa mémoire sont remplacés, donc plus rien de l'ancien programme n'est exécuté (ni même ce qu'il y a après `execve()`). Le `printf()` ne sera donc jamais executé.

Si cela échoue, le message d'erreur (avec `perror()`) est affiché, puis le processus fils se termine avec un code d'erreur.

Le processus parent (celui où la valeur de retour de `fork()` est différente de 0) continue la boucle pour traiter l'argument suivant.

Types de fichiers exécutables

MS-DOS / Windows

COM	Exécutable très limité, n'est presque plus utilisé
PE (Portable Executable)	Fichiers exécutables: .EXE Librairies partagées: .DLL ActiveX: .OCX

Mac OS X

Mach-O (Mach Object)	Applications: .app Frameworks Librairies partagées: .dylib Objets compilées: .o

Types de fichiers exécutables

Unix/Linux

a.out	Format original d'objets et exécutables Unix, non adapté au librairies partagées
COFF (Common Object File Format)	Ancien format d'objets et exécutables Unix, non adapté au librairies partagées
ELF (Executable and Linkable Format)	Fichiers exécutables: .o Librairies partagées: .so Fichiers core (core dumps) Utilisable sur plusieurs plate-formes

Organisation d'un fichier ELF

Segments:

- permettent à l'OS de préparer le programme pour son exécution (`exec...()`);
- contiennent une ou plusieurs sections;

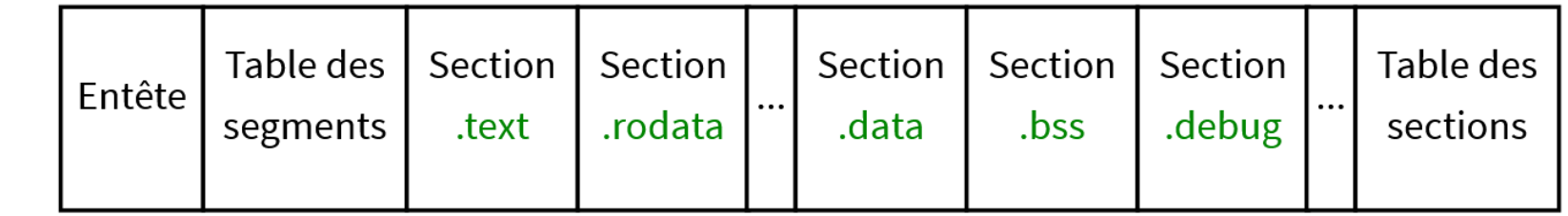
Sections:

- contiennent toutes les informations du programme (code, données), aussi celles pas forcément nécessaires à l'exécution, e.g. débogage (*debugging*);
- sont nécessaires lors de la phase de liage (*linking*);

Entêtes et tables:

- indiquent la position de chaque section;
- indiquent la position de chaque segment;
- indiquent la position de la table des sections et de la table des segments.

Organisation d'un fichier ELF




```c
#include <elf.h>
typedef struct {
//les variables ci-dessous ne sont pas listée dans l'ordre de l'entête
...
uint16_t e_type; /* Executable, bibliothèque, objet, ... */
uint_16 e_machine; /* Intel, HP,...*/
ElfN_Addr e_entry; /* Première instruction à exécuter par le processus */

ElfN_Off e_phoff; /* Offset de départ de la table des segments */
uint16_t e_phentsize; /* Taille d'une entrée dans la table des segments*/
uint16_t e_phnum; /* nombre d'entrées dans la table des segments */

ElfN_Off e_shoff; /* Offset de départ de la table des sections */
uint16_t e_shentsize; /* Taille d'une entrée dans la table des sections*/
uint16_t e_shnum; /* nombre d'entrées dans la table des sections*/
...
} ElfN_Ehdr;
```

Organisation d'un fichier ELF

La table des sections permet de définir les sections dans le fichier. Une section peut contenir des informations de liage, du code, des données.

```
typedef struct {
        ...
        uint32_t sh_name; /*Index spécifiant le nom de la section (.text, .data, etc.)*/
        ElfN_Addr sh_addr; /* Adresse de la section en mémoire virtuelle */
        ElfN_Off sh_offset; /* Offset de la section dans le fichier ELF*/
        uintN_t sh_size; /* Taille de la section */
        ...
} ElfN_Shdr;
```

La table des segments (program header) permet de regrouper les sections en plusieurs segments, qui peuvent être chargés en mémoire virtuelle lors de l'exécution.

```
typedef struct {
        uint32_t p_type; /* if == PT_LOAD -> le segment doit être placé en mémoire */
        ElfN_Off p_offset; /* Offset du segment dans le fichier */
        uintN_t p_filesz; /* Taille du segment dans le fichier*/

        ElfN_Addr p_vaddr; /* Adresse où charger le segment en mémoire virtuelle */
        uint32_t p_memsz; /* Taille du segment en mémoire, si >= p_filesz, complété par des 0 */
        uintN_t p_flags; /* Exec, write, read */
        ...
        } ElfN_Phdr;
```

Organisation d'un fichier ELF

On peut observer le contenu d'un fichier ELF avec la commande `objdump` :

.bss	Section contains uninitialized read-write data.
.comment	Comment section.
.data	Section contains initialized read-write data.
.rodata	Section contains read-only data.
.debug	Section contains debugging information.
.fini	Section contains runtime finalization instructions.
.init	Section contains runtime initialization instructions.
.text	Section contains executable text.
.line	Section contains line # info for symbolic debugging.
.note	Section contains note information.

Une autre commande utile est `readelf` .

Ordonnancement préemptif (*preemptive scheduling*)

Le noyaux se charge de distribuer les processus sur les différents CPU:

- l'ordonnanceur attribue un CPU à un processus, généralement pour une tranche de temps précise appelée **quantum**;
- l'ordonnanceur choisi quel est le nouveau processus qui va être alloué à ce CPU une fois le quantum ou le processus terminé.

Toutefois un processus peut libérer un CPU volontairement avant la fin du quantum si:

- il se met en attente d'une ressource;
- il reçoit un signal de suspension (SIGSTOP, SIGSTP);
- il appelle la fonction `sched_yield()`.

Un quantum = 100 ms (variable RR_TIMESLICE dans le code du kernel Linux):

```
#define RR_TIMESLICE    (100 * HZ / 1000)
```

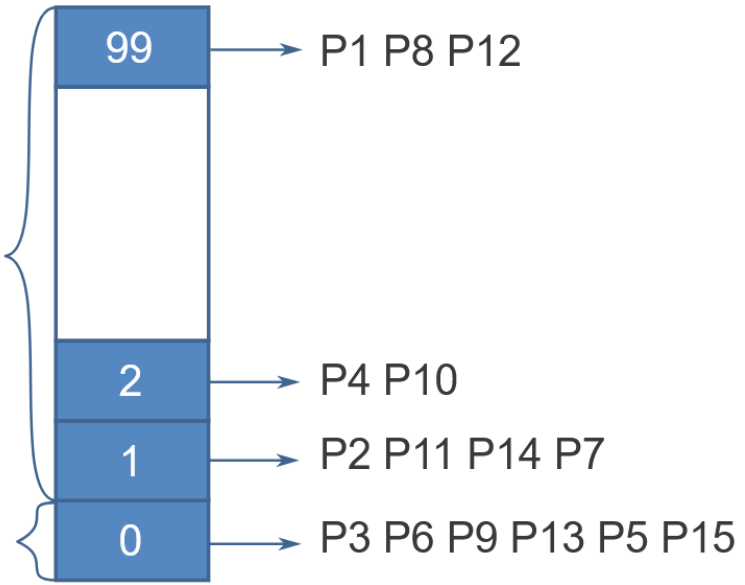
Ordonnancement sous Linux

Implémentation:

- il existe une liste de processus pour différentes priorités statiques [0-99];
- les processus de hautes priorités sont toujours exécutés d'abord.

Priorité statique de 1 à 99:
Processus "temps réel" (TR) qui
sont ordonnancés en FIFO (SCHED_FIFO)
ou en Round Robin (SCHED_RR).
Il faut en général avoir des droits privilégiés.

Priorité statique 0: la plus part
des processus (sinon tous) sont
exécuté dans cette priorité statique.
C.f. slide suivant.



Priorité statique 0

Trois modes d’ordonnancement sont disponibles en priorité statique 0:

- standard (i.e. par défaut – SCHED_OTHER);
- pour processus à lourde charge de calcul (SCHED_BATCH);
- pour processus à très faible priorité (SCHED_IDLE).

Priorité statique 0 - SCHED_OTHER

Cette stratégie permet de s'assurer que chaque processus sera traité après avoir eu un certain nombre de déni de CPU qui dépends de sa priorité et de celle des autres processus.

Le processus de la liste est choisi par rapport à une priorité dynamique = valeur *nice* + nombre de quantum en état prêt sans avoir de processeur à disposition.

La valeur *nice* d'un processus est attribuée par la commande `nice` ou par la fonction C `setpriority()`.

À chaque processus est associé une valeur *niceness* (de -20 à 19; 0 pour tout nouveau processus). Plus *nice* est élevée, plus la priorité est faible.

`nice -n -5 commande` Lance la commande avec niceness = -5

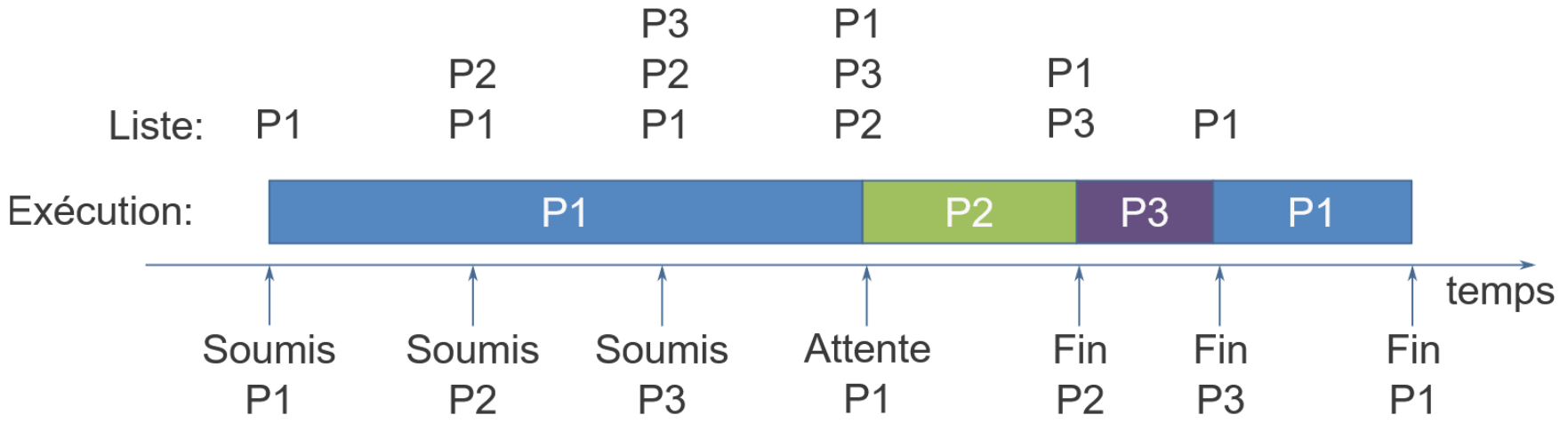
`renice -5 commande` Modifie niceness de la commande en exécution à -5

Les utilisateurs non privilégiés peuvent utiliser une niceness entre 1 et 19.

Ordonnancement Temps Réel
FIFO

Implémentation:

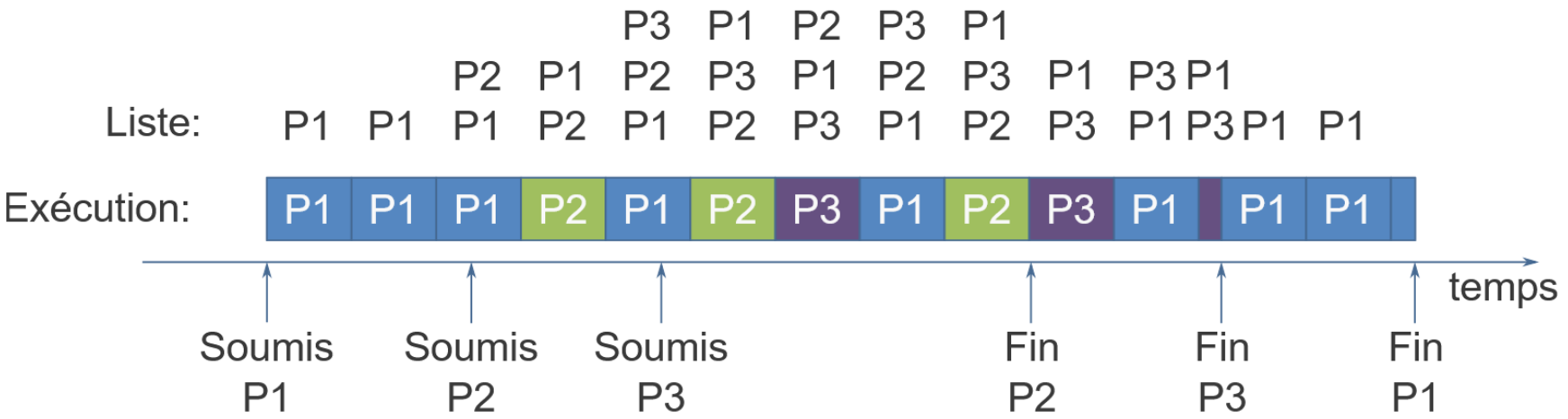
- les processus sont rangés dans une liste lors de leur soumission;
- un nouveau processus est placé en queue de liste;
- chaque processus est exécuté sans interruption (i.e. non préemptif, sauf cas de mise en attente).



Ordonnancement Temps Réel
Round Robin

Implémentation:

- les processus sont placés en queue de liste lors de leur soumission (comme dans le scheduling FIFO);
- chaque processus est exécuté uniquement pour un quantum de temps (préemptif), puis est replacé en fin de liste.



Contrôle d'ordonnancement

L'ordonnancement d'un processus peut être contrôlé par les fonctions et structures suivantes pour la priorité statique:

```
#include <sched.h>
int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);
int sched_getscheduler(pid_t pid);

struct sched_param {
    ...
    int sched_priority; /* prioritée statique */
    ...
};
```

policy peut prendre les valeurs: SCHED_OTHER, SCHED_BATCH, SCHED_IDLE, SCHED_FIFO, SCHED_RR.

La priorité dynamique peut être contrôlée en utilisant:

```
int setpriority(int which, int who, int prio); /* priorité dynamique, nice */
int getpriority(int which, int who);
```

Commutation de contexte (*context switching*)

Lorsque qu'un processus doit en remplacer un autre sur un CPU, une **commutation de contexte** a lieu:

- suspension de l'exécution du processus et sauvegarde de son contexte;
- rétablissement de l'état du CPU à l'état sauvegardé lors de la suspension du processus qui reprend son exécution;
- mise en exécution du nouveau processus.

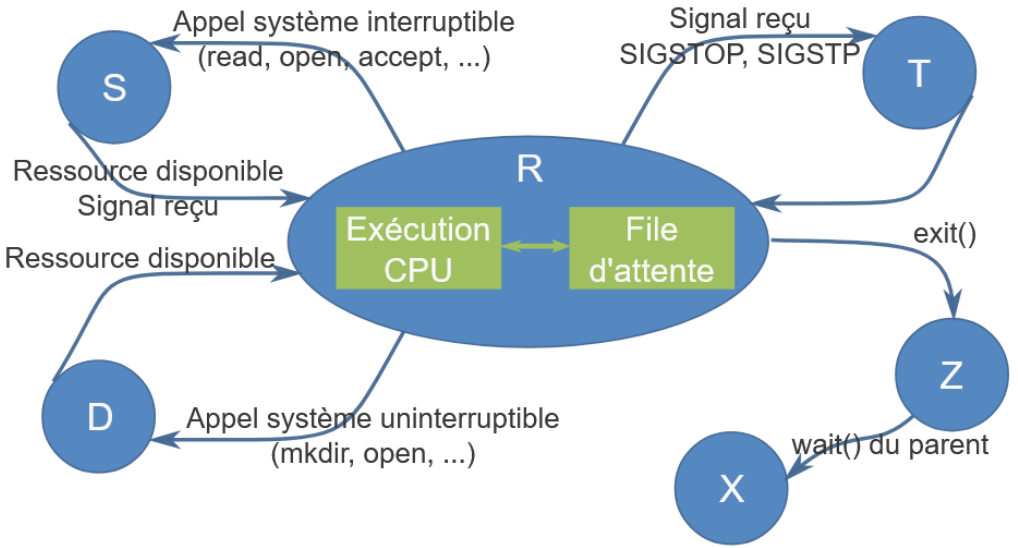
Au passage il faut:

- écrire sur le disque les pages modifiées (pages sales);
- mettre à jour les informations du noyau pour tenir compte du changement de processus actif (par exemple re-calcul de la priorité des processus).

Commutation de contexte (*context switching*)

Un changement de contexte est coûteux, d'où l'avantage des **threads**.
Les threads d'un même processus partagent la mémoire virtuelle et la plupart des ressources du programme parent. Ainsi, lors d'un changement de contexte entre threads, il n'est pas nécessaire de recharger tout l'espace mémoire ou de sauvegarder/écrire autant d'informations qu'avec un processus entier.
Les threads minimisent donc le coût des changements de contexte et facilitent le partage des ressources, ce qui améliore l'efficacité globale du système.

États d'un processus



ps	Constante noyau	Description de l'état
R	TASK_RUNNING	En exécution ou prêt à être exécuté (i.e. dans une file d'attente)
S	TASK_INTERRUPTIBLE	En attente d'une ressource (Sleep) et interruptible par un signal
D	TASK_UNINTERRUPTIBLE	En attente d'une ressource mais ne peut pas être interrompu
T	TASK_STOPPED	Suspendu (sToppé)
Z	EXIT_ZOMBIE	Zombie en attente d'un wait de la part du parent.
X	EXIT_DEAD	Terminé, mort, ne devrais jamais être observé.

États d'un processus: exercice

Lancez la commande `sleep 20` et mettez ce processus dans l'état Stopped en envoyant un SIGTSTP (CTRL+Z).
Lancez `top` et examinez l'output.
Vous pouvez filtrer par champ via la touche O et en insérant p.ex. "PID=5678".

Colonne "Status" dans l'output:

- D = uninterruptible sleep
- I = idle
- R = running
- S = sleeping
- T = stopped by job control signal
- t = stopped by debugger during trace
- Z = zombie

Communications inter-processus (*IPC*)

Pipes et *FIFO*

```
$ ls -lh /dev | more
```

Le noyau crée un canal de communication anonyme (***pipe*** en anglais, **tube** en français) entre les processus `ls` et `more`

Très utile dans le monde UNIX/Linux, p.ex.
`cut -d: -f7 /etc/passwd | sort | uniq`
donne la liste des shells de login des utilisateurs d'un système.

Il est également possible de créer des **tubes nommées** (***named pipes*** ou ***FIFO***).

Pipes (tubes anonymes)

On peut créer un canal de communication anonyme en utilisant:

```
int pipe(int fildes[2]);
```

`fildes[0]` est un descripteur de fichier représentant la sortie du tube/pipe (i.e. on peu lire sur ce descripteur).
`fildes[1]` est un descripteur de fichier représentant l'entrée du tube/pipe (i.e. on peu écrire sur ce descripteur).

Retourne -1 en cas d'erreur (voir `errno`).

Pas d'accès aléatoire possible.

Pipes (tubes anonymes)

Les tubes et FIFO:

- Permettent une communication à haute vitesse entre deux processus sur la même machine.
- Ont deux extrémités: une ouverte en lecture et une ouverte en écriture.
- Sont unidirectionnels: l'information ne transite que dans un sens.
- Sont bloquants:
  - L'ouverture d'une extrémité bloque jusqu'à l'ouverture de l'autre extrémité
    -> L’écriture bloque tant que la lecture n’est pas prête (et inversement)
    -> Permet d'établir des rendez-vous
  - Possibilité de deadlock en cas d'écriture dans la pipe (sans lecture) de plus de données que le pipe buffer peut en contenir !

FIFOs (tubes nommées)

On peut créer un FIFO avec la commande shell:

```
mkfifo [OPTION]... NOM...
```

où **`NOM`** est le nom du FIFO à créer.
Parmi les options on peut passer les permissions du FIFO par l'option `-m` *`MODE`*.

Dans un terminal:

```
mkfifo -m 0640 /tmp/fifo1
ls -lh /dev > /tmp/fifo1
```

Dans un autre terminal:

```
less < /tmp/fifo1
```

FIFOs (tubes nommées)

On peut créer un FIFO avec l'appel système:

```
int mkfifo(const char *pathname, mode_t mode);
```

`pathname` est le nom du fichier à créer

`mode` représente les permissions (modifiées via mode & ~umask)

Un FIFO peut être ouvert en lecture/écriture comme n'importe quel fichier (open/read) mais il faut veiller à respecter la direction du FIFO.

Pas d'accès aléatoire possible.

Exemple: producer.c

```
/* Inspiré d'un exemple de J. Menu */

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <sys/fcntl.h>
#include <sys/stat.h>
#include <unistd.h> // pour STDOUT_FILENO

#define BUF_SIZE 64

int createFIFO( char *name ) {
  int fifo = -1;
  int result = mkfifo( name, 0600 );
  if( result < 0 ) {
    perror( "Cannot create the fifo" );
    exit(EXIT_FAILURE);
  }
  printf( "Created fifo %s\n", name );
  fifo = open( name, O_WRONLY );
  if( fifo < 0 ) {
    perror( "Cannot open the fifo" );
    exit(EXIT_FAILURE);
  }
  printf( "The fifo %s is open\n", name );
  return fifo;
}

void stdin2fifo( int fifo ) {
  char buffer[BUF_SIZE];
  ssize_t nread, nwrit;
  while (1) {
    nread = read(STDIN_FILENO, buffer, BUF_SIZE);
    if( nread < 0 ) {
      perror( "Reading STDIN failed ");
      exit(EXIT_FAILURE);
    }
    if( nread == 0 ) {
      printf( "Received End-Of-File \n" );
      return;
    }
    nwrit = write( fifo, buffer, nread );
    if( nwrit != nread ) {
      perror( "Writing to fifo failed" );
      exit(EXIT_FAILURE);
    }
  }
}

int main (int argc, char **argv) {
  if (argc != 2) {
    printf( "Usage: %s <fifo_name> \n", argv[0] );
    exit(EXIT_FAILURE);
  }
  int fifo = createFIFO( argv[1] );
  stdin2fifo( fifo );
  close(fifo);
  exit(EXIT_SUCCESS);
}
```

Exemple: consumer.c

```
/* Inspiré d'un exemple de J. Menu */

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <sys/fcntl.h>
#include <unistd.h>    // pour STDOUT_FILENO

#define BUF_SIZE 64

int openFIFO( char *name ) {
  int fifo = open( name, O_RDONLY );
  if( fifo < 0 ) {
    perror( "Cannot open the fifo" );
    exit(EXIT_FAILURE);
  }
  printf( "The fifo %s is open\n", name );
  return fifo;
}

void fifo2stdout( int fifo ) {
  char buffer[BUF_SIZE];
  ssize_t nread, nwrit;
  while(1) {
    nread = read( fifo, buffer, BUF_SIZE );
    if( nread < 0 ) {
      perror( "Reading fifo failed ");
      exit(EXIT_FAILURE);
    }
    if( nread == 0 ) {
      printf( "Received End-Of-File\n" );
      return;
    }
    nwrit = write( STDOUT_FILENO, buffer, nread );
    if( nwrit != nread ) {
      perror( "Writing to STDOUT failed" );
      exit(EXIT_FAILURE);
    }
  }
}

void disposeFIFO( int fifo, char *name ) {
  int ret = close( fifo );
  if( ret < 0 ) {
    perror( "Cannot close fifo." );
  }
  ret = unlink( name );
  if( ret < 0 ) {
    perror( "Cannot remove fifo." );
  }
}

int main (int argc, char **argv ) {
  if (argc != 2) {
    printf( "Usage: %s <fifo_name> \n", argv[0] );
    exit(EXIT_FAILURE);
  }
  char *name = argv[1];
  int fifo = openFIFO( name );
  fifo2stdout( fifo );
  disposeFIFO( fifo, name );
  exit(EXIT_SUCCESS);
}
```

Fichier en mémoire partagée (*memory-mapped file*)

Nous avons vu que certaines pages de la mémoire virtuelle:

- ne sont pas présentes en mémoire physique mais résident sur des systèmes de fichiers (swap / fichiers exécutables)
- deviennent disponibles au fur et à mesure des fautes de pages
- sont partagées entre plusieurs processus (e.g. librairies partagées)
- peuvent être partagées uniquement jusqu'à leur modification (copy-on-write)

Nous allons voir un appel système qui permet d'associer un segment de mémoire virtuelle à un segment de fichier. Cet appel système est par exemple utilisé pour charger les librairies partagées.

File mapping

File mapping = Associer le segment (une partie) d'un fichier à un nouveau segment de mémoire partagée.

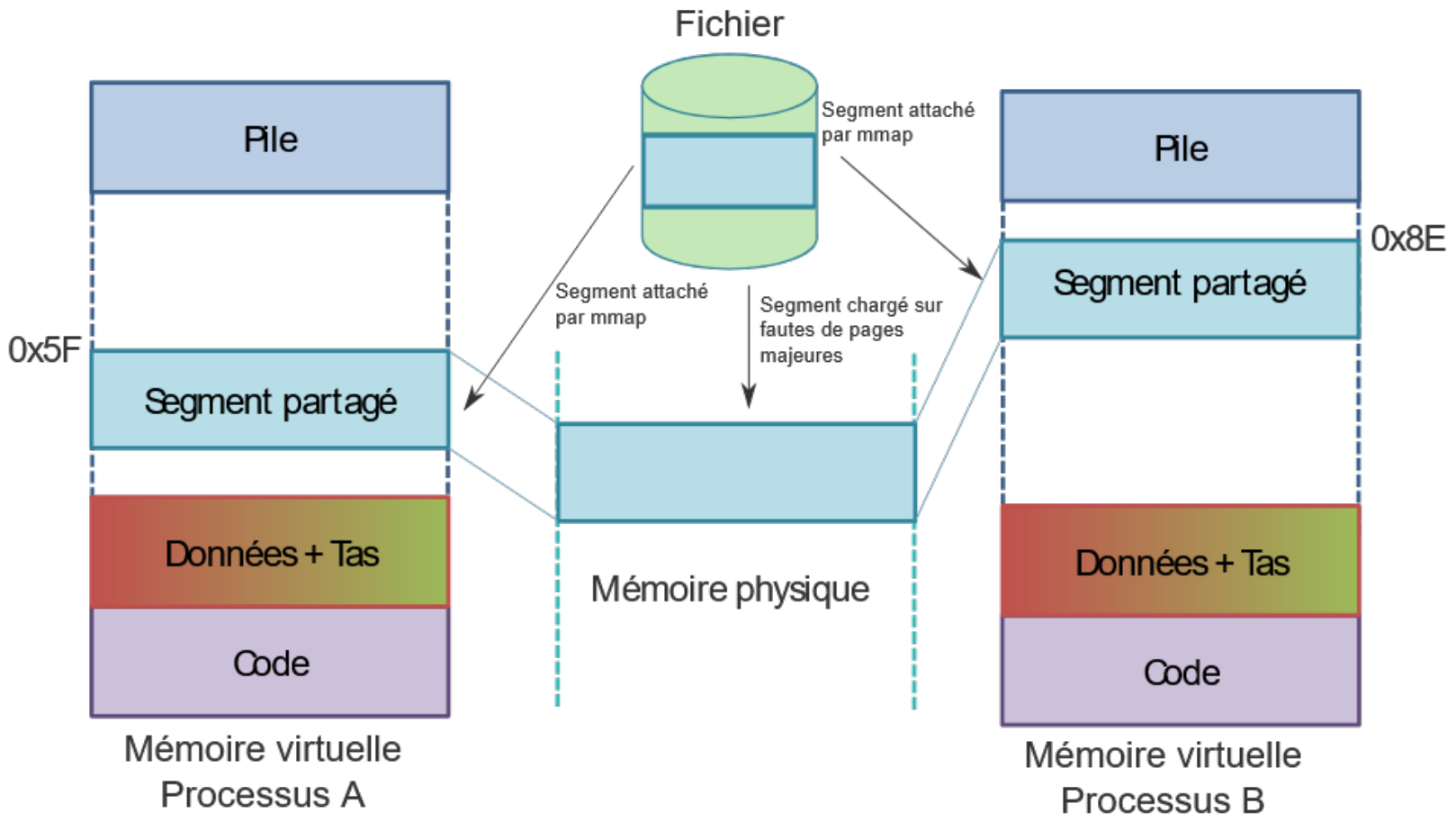
Cela permet de:

- partager des pages (données, instructions) entre plusieurs processus
- accéder aux données directement en mémoire (i.e. par pointeurs) plutôt que dans un fichier (i.e. par curseur)

Un tel fichier ne sera pas chargé intégralement en mémoire mais page par page au fur et a mesure des fautes de page du processus.

File mapping

Deux processus peuvent partager un segment en y associant des espaces d'adressage virtuel différents:



File mapping

Pour associer un fichier à un espace de la mémoire virtuelle du processus on:

1) ouvre le fichier en lecture et/ou écriture pour obtenir un descripteur de fichier **`fd`**:

```
int open(const char *pathname, int flags);
```

2) associe le descripteur de fichier à une zone de la mémoire virtuelle:

```
void *mmap(void *addr, size_t length, int prot, int flags,
int fd, off_t offset);
```

3) désassocie la mémoire partagée:

```
int munmap(void *addr, size_t length);
```

4) ferme le fichier:

```
int close(int fd);
```

File mapping

On ouvre un fichier avec l'appel système suivant:

```
int open(const char *pathname, int flags);
```

- **`pathname`** est le nom du fichier;
- **`flags`** est un champ de bit indiquant le mode d'accès au fichier (O_RDONLY, O_WRONLY, O_RDWR);
- retourne un entier représentant le fichier (descripteur de fichier), soit -1 en cas d'erreur (cf. errno).

On ferme un fichier avec l'appel système suivant:

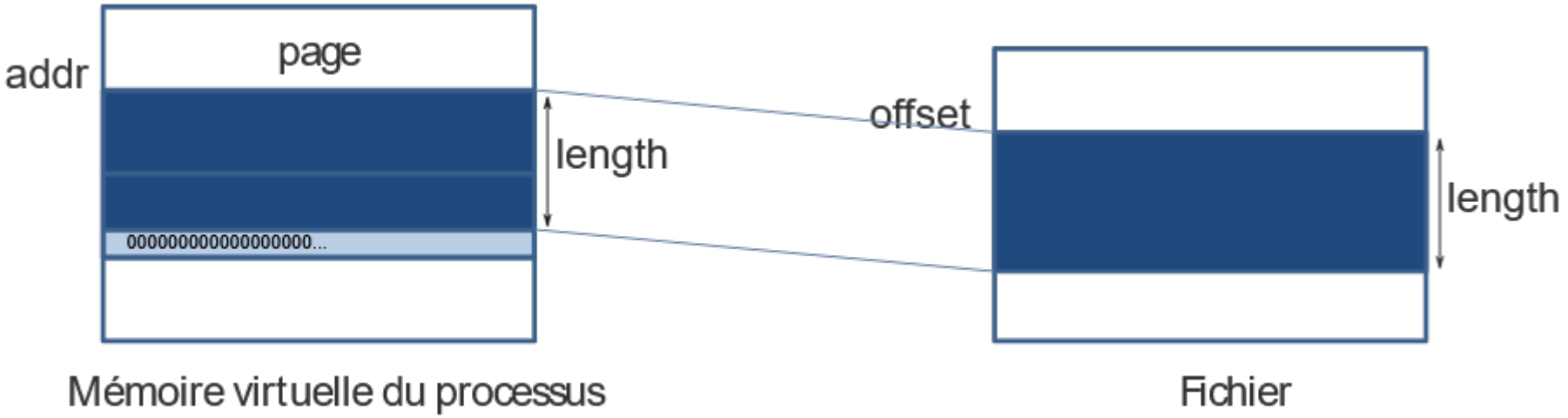
```
int close(int fd);
```

- **`fd`** est l'entier représentant le fichier (descripteur de fichier);
- retourne 0 en cas de succès, -1 en cas d'erreur (cf. errno).

mmap()

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

- **`fd`**: entier représentant le fichier (*file descriptor* ou descripteur de fichier);
- **`addr`**: adresse d'un début de page (ajustée automatiquement), si NULL alors l'adresse est choisie automatiquement;
- **`offset`**: début du mapping dans le fichier, doit être multiple de la taille d'une page
- **`length`**: taille du mapping dans le fichier, complété par des zéros pour remplir une page en mémoire
- retourne l'adresse virtuelle correspondant au début du segment



mmap()

```c
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

`prot` est un champs de bit définissant la protection des pages partagées:

- PROT_READ / PROT_WRITE / PROT_EXEC autorise respectivement la lecture, l'écriture et l'exécution;
- PROT_NONE ne donne aucun droit, est utilisé pour réserver des pages;
- les droits doivent correspondre au mode d'ouverture du fichier.

`flags` est un champs de bit utilisé pour les options suivantes:

- MAP_SHARED: la zone est partagée entre les processus et les fichiers; toute modification sera reportée aux autres processus et dans le fichier;
- MAP_PRIVATE: copy-on-write, si un processus modifie le contenu il crée sa propre copie des pages et le fichier ne sera pas modifié;
- MAP_ANONYMOUS: pas d'association avec un fichier, la mémoire est initialisée à 0 (**`fd`** et **`offset`** sont ignorés) et partageable uniquement avec ses enfants.

Fichier en mémoire partagée: exercice

L'objectif de cet exercice est de créer un processus qui:

- charge le segment contenant le code d'un autre exécutable dans sa mémoire virtuelle;
- exécute ce code à partir du point d'entrée mentionné dans l'exécutable.

On simule ainsi l'exécution d'un code externe un peu comme cela est fait pour une librairie partagée.

Fichier en mémoire partagée: exercice

Le programme à charger est fournit en assembleur. Il faut le compiler avec `nasm` (voir Makefile).

```
; simple.asm
; this program just returns 42
; you can use the command readelf to see its content and determine
; the main function entry point (i.e. _start function) address
BITS 64
GLOBAL _start
SECTION .text
    jmp     $       ; jump to current address -> infinite loop
_start:
    mov     eax, 1  ; systemcall number 1 -> exit
    mov     ebx, 42 ; parameter of system call -> exit with code 42
    int     0x80    ; interrupt executing the system call with number
                    ; stored in eax
```

Fichier en mémoire partagée: exercice

```
all: simple exec

simple: simple.o
    ld simple.o -o simple

simple.o: simple.asm
    nasm -f elf64 simple.asm

exec: exec.c
    gcc -std=gnu11 -Wall exec.c -o exec

clean:
    rm simple.o simple exec
```

Fichier en mémoire partagée: exercice

Il faut donc:

- compiler le programme **`simple`**
- exécuter ce programme et confirmer son status de retour (42)
- utiliser la commande **`readelf`** pour identifier le segment à charger et le point d'entrée du programme
- implementer un programme en C qui va:
  - définir les adresses mesurées à l'étape précédente avec des **`#define`**
  - charger le bon segment avec **`mmap`**
  - définir une fonction pointant sur le point d'entrée dans le mémoire virtuelle
  - exécuter cette fonction
- tester que votre programme C retourne bien 42. (Note: `echo $?` vous permet de voir l'exit code de la dernière commande lancée.)

Pour aller plus loin: votre programme C devra rechercher les informations du segment et du point d'entrée directement dans le fichier executable.

Fichier en mémoire partagée: solution

```
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

//Information gathered from readelf on the simple executable
//This information can be different on your system. This should be changed accordingly
#define SEG_OFFSET 0x1000
#define SEG_SIZE 0x0e
#define ENTRY_PT 0x1002

void onErr(const char *title) {
    perror(title);
    exit(EXIT_FAILURE);
}

int main() {

    //Open executable file
    int fd;
    if((fd = open("./simple", O_RDONLY)) == -1)
        onErr("open");

    //Map file in memory
    void *segment;
    if((segment = mmap(NULL, SEG_SIZE, PROT_EXEC | PROT_READ, MAP_PRIVATE, fd, SEG_OFFSET)) == MAP_FAILED)
        onErr("mmap");

    getchar();

    //Define a pointer to the _start function
    void (*ptFunc)(void) = segment + (ENTRY_PT - SEG_OFFSET);
    ptFunc();

    munmap(segment, SEG_SIZE);

    close(fd);

}
```

Fichier en mémoire partagée: solution

Le programme en C devrait marcher correctement sans besoin de changer les valeurs données dans le cours:

```
#define SEG_OFFSET 0x1000
#define SEG_SIZE 0x0e
#define ENTRY_PT 0x1002
```

Les deux premières valeurs sont obtenues depuis l'output de
`readelf -a simple`
en regardant dans le segment 01 qui correspond à la section `.text`.

Fichier en mémoire partagée: solution

```
#readelf -a simple
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x401002
  Start of program headers:          64 (bytes into file)
  Start of section headers:          4152 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         3
  Size of section headers:           64 (bytes)
  Number of section headers:         4
  Section header string table index: 3
```

Fichier en mémoire partagée: solution

```
Section Headers:
  [Nr] Name              Type            Address           Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .note.gnu.bu[...] NOTE            00000000004000e8  000000e8
       0000000000000024  0000000000000000   A       0     0     4
  [ 2] .text             PROGBITS        0000000000401000  00001000
       000000000000000e  0000000000000000  AX       0     0     16
  [ 3] .shstrtab         STRTAB          0000000000000000  0000100e
       0000000000000024  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)

There are no section groups in this file.
```

Fichier en mémoire partagée: solution

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x000000000000010c 0x000000000000010c  R      0x1000
  LOAD           0x0000000000001000 0x0000000000401000 0x0000000000401000
                 0x000000000000000e 0x000000000000000e  R E    0x1000
  NOTE           0x00000000000000e8 0x00000000004000e8 0x00000000004000e8
                 0x0000000000000024 0x0000000000000024  R      0x4

 Section to Segment mapping:
  Segment Sections...
   00     .note.gnu.build-id
   01     .text
   02     .note.gnu.build-id
```

Fichier en mémoire partagée: solution

Le programme en C devrait marcher correctement sans besoin de changer les valeurs données dans le cours:

```
#define SEG_OFFSET 0x1000
#define SEG_SIZE 0x0e
#define ENTRY_PT 0x1002
```

Pour la troisième valeur, on calcule:

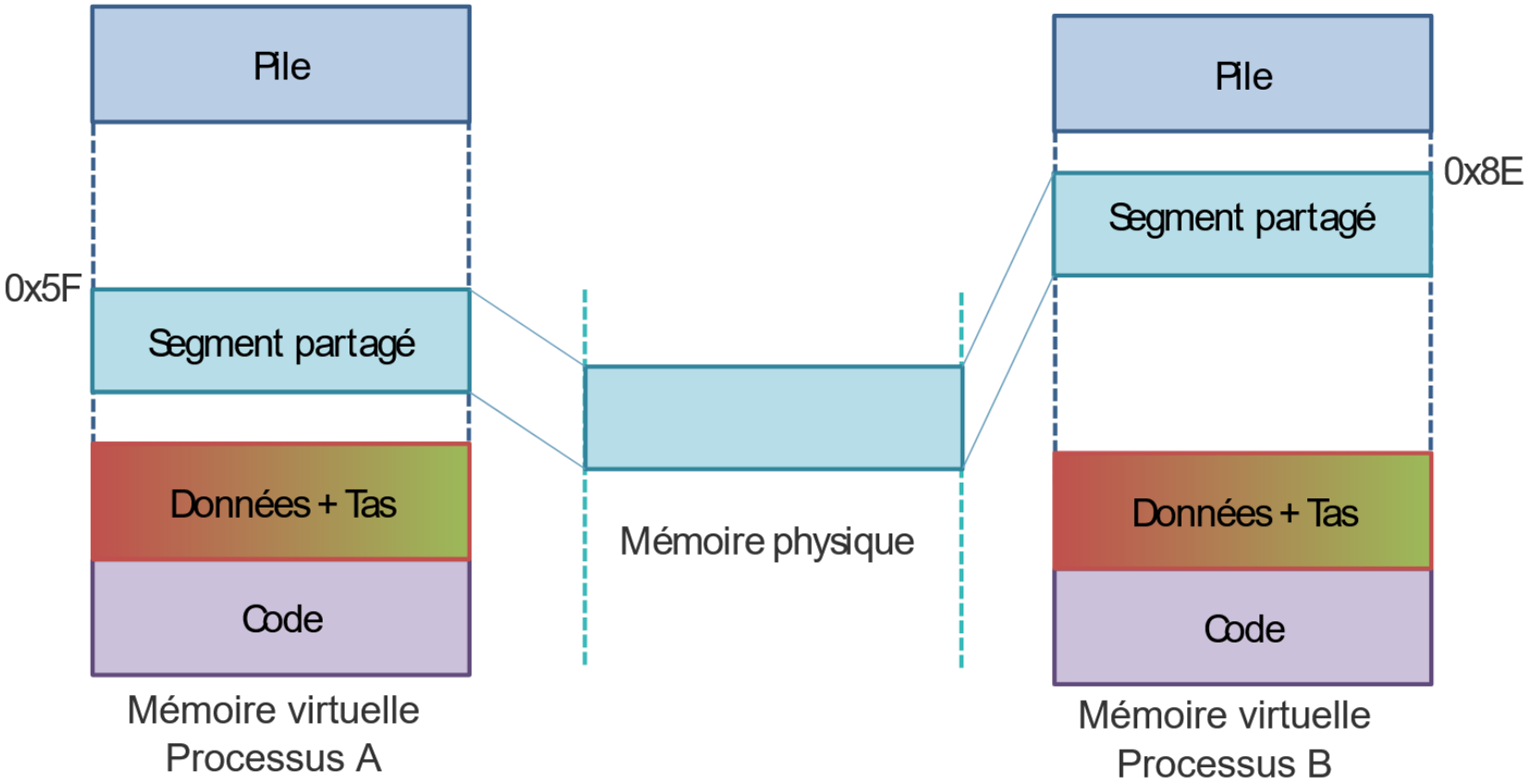
entry point = entry point address - segment virtual address + segment offset

entry point = 0x401002 - 0x401000 + 0x1000

entry point = 0x1002

Mémoires partagées sans fichier

Deux processus peuvent partager un segment en y associant des espaces d'adressage virtuel différents:



Mémoires partagées sans fichier

Un segment partagé:

- est alloué par le noyau sur requête d’un processus;
- peut être intégré dans les espaces d’adressage d’autres processus qui se l’attachent à des adresses potentiellement différentes;
- contient forcément un nombre entier de pages (ce qui est vrai pour tous les segments).

Types de mémoire partagée

Historiquement il existe deux moyens UNIX / Linux pour créer des mémoires partagées entre deux processus sans relation parent / enfant:

XSI Inter Process Communication (IPC)

- est hérité de SystemV;
- est basé sur un système de clef et d'identificateurs;
- n'est pas limité au mémoires partagées mais est aussi utilisé pour créer des files de messages et des sémaphores.

POSIX shared memory objects 

- est basé sur la fonction `mmap()`;
- utilise des descripteurs de fichier;
- est utilisé par Linux pour gérer les mémoires partagées.

Mémoires partagées sans fichier

Pour créer un nouvel objet "mémoire partagée" ou pour ouvrir un objet existant on utilise:

```
#include <sys/mman.h>
#include <sys/stat.h> // For mode constants
#include <fcntl.h>    // For O_* constants
int shm_open(const char *name, int oflag, mode_t mode);
```

Retourne un descripteur de fichier représentant la mémoire partagée.
`name` n'est pas un nom de fichier standard mais doit commencer par "/", on retrouvera la mémoire partagée dans `/dev/shm/`**`name`**.
`oflag` et **`mode`** fonctionnent comme pour `open()`, notamment O_RDONLY, O_RDWR, O_CREAT, O_EXCL.

Attention à ne pas écraser une mémoire partagée créée par un autre utilisateur !

Mémoires partagées sans fichier

Le descripteur obtenu par `shm_open()` crée un objet de mémoire partagée POSIX mais avant sont utilisation il faut lui donner une taille. Cela est effectué par l'appel à:

```
#include <unistd.h>
int ftruncate(int fd, off_t length); // length est la taille en octets
```

`fd` est un descripteur de fichier (ici une mémoire partagée mais fonctionne aussi sur les fichiers)
`length` est la nouvelle taille de la mémoire

Mémoires partagées sans fichier

On utilise les fonctions habituelles pour effectuer un mapping de la mémoire dans l'espace virtuel du processus:

```
#include <sys/mman>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t len);
```

Pour supprimer la référence à la mémoire partagée on utilise:

```
int shm_unlink(const char *name);
```

Comme pour l'appel à `unlink()`, uniquement la référence est supprimée (i.e. `/dev/shm/`**`name`**). La mémoire ne sera effectivement détruite que si elle est désassociée par `munmap()`.

Mémoires partagées sans fichier: exercice

Le programme dans les slides suivantes montre un exemple de construction de mémoire partagée:

```
shm.h
creator.c
helper.c
Makefile
```

Ce programme souffre d'un défaut important. Lequel ?

shm.h

```
#ifndef _SHM_H
#define _SHM_H

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

//La mémoire partagée contiendra deux valeur:
//- un indicateur stipulant si un autre processus est près pour l'opération
//- un nombre qui sera incrémenté par les deux processus conjointement
typedef struct {
    char isReady;
    long int number;
} sharedMemory;

#define READY 0
#define NUM_INCREMENTS 10000000


void OnError(const char *str)
{
    perror(str);
    exit(EXIT_FAILURE);
}

#endif
```

creator.c

```
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include "shm.h"

int main(int argc, char* argv[])
{
    int fd, i;
    sharedMemory *shm;

    //Vérifier les entrées
    if(argc != 2)
    {
        printf("Usage: shm-create /sharedMemoryName\n\n");
        exit(0);
    }

    //Créer une mémoire  partagée en lecture / écriture
    if( (fd = shm_open(argv[1], O_RDWR | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR)) == -1)
        OnError("shm_open");

    //Taille de la mémoire = indicator + number
    if( ftruncate(fd, sizeof(sharedMemory)) == -1)
        OnError("ftruncate");

    //File mapping (Les parametres doivent correspondres au mode d'ouverture de l'objet POSIX)
    shm = mmap(NULL, sizeof(sharedMemory), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if(shm == MAP_FAILED)
        OnError("mmap");
    printf("Adr memoire partagée:%p\n", shm);

    //Initialization memoire a zero
    shm->number = 0;

    //On attend l'autre processus pour travailler
    shm->isReady = !READY;
    printf("Ok j'attends un processus qui peut m'aider...\n");
    while(!(shm->isReady == READY));

    //On peut desassocier l'objet (retirer l'inode) car l'objet ne sera supprimé que lors du munmap
    shm_unlink(argv[1]);

    //Ok on travaille
    for(i=0;i<NUM_INCREMENTS;i++)
        shm->number = shm->number + 1;

    //Afficher le résultat
    printf("Pour moi %d, le total est %ld\n", getpid(), shm->number);

    //Unmap
    if(munmap(shm, sizeof(sharedMemory)) == -1)
        OnError("munmap");

    return 0;
}
```

helper.c

```
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include "shm.h"

int main(int argc, char* argv[])
{
    int fd, i;
    sharedMemory *shm;

    //Vérifier les entrées
    if(argc != 2)
    {
        printf("Usage: shm-help /sharedMemoryName\n\n");
        exit(0);
    }

    //Ouvrir une mémoire  partagée en lecture / écriture
    if( (fd = shm_open(argv[1], O_RDWR, S_IRUSR | S_IWUSR)) == -1)
        OnError("shm_open");

    //File mapping (Les parametres doivent correspondres au mode d'ouverture de l'objet POSIX)
    shm = mmap(NULL, sizeof(sharedMemory), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if(shm == MAP_FAILED)
        OnError("mmap");
    printf("Adr memoire partagée:%p\n", shm);

    //On signal que l'on est prêt
    shm->isReady = READY;

    //Ok on travaille
    for(i=0;i<NUM_INCREMENTS;i++)
        shm->number = shm->number + 1;

    //Afficher le résultat
    printf("Pour moi %d, le total est %ld\n", getpid(), shm->number);

    //Unmap
    if(munmap(shm, sizeof(sharedMemory)) == -1)
        OnError("munmap");

    return 0;
}
```

Makefile

```
CC=gcc
CC_OPTS= -Wall -g
LN_OPTS= -lrt

all: shm-create shm-help

shm-create: creator.c shm.h
	$(CC) $(CC_OPTS) creator.c -o shm-create $(LN_OPTS)

shm-help: helper.c shm.h
	$(CC) $(CC_OPTS) helper.c -o shm-help $(LN_OPTS)
```

Mémoires partagées sans fichier: solution

Le défaut: ce programme n'utilise aucun mécanisme de synchronisation pour protéger l'accès concurrent à la variable partagée `number`, ce qui provoque des **conditions de course (*race conditions*)** entre les deux processus.

En fait, cette opération n'est pas **atomique**:

```
shm->number = shm->number + 1;
```

et aucun verrou (mutex, sémaphore, etc.) n'est utilisé pour garantir que l'accès (lecture + modification + écriture) à la variable est exclusif à un processus à un instant donné.
Cette opération pourrait donc être interrompue et reprise dans l'autre processus entre la lecture et la réécriture, conduisant à des incréments perdus. Le résultat final peut varier à chaque exécution.

Conditions de course (*race conditions*)

Lorsque deux processus coopèrent (e.g. partagent des données) il faut faire extrêmement attention aux conditions de course:

- les deux processus peuvent être vus comme concurrents sur l'accès aux données;
- si il n'y a pas de contrôle d'accès sur ces données, il peut y avoir conflit dans leur utilisation.

Cela est partiellement dû au fait qu'un processus peut être suspendu par l'ordonnanceur au milieu d'une opération, laissant les données partagées dans un état intermédiaire.

Seule les opérations dites atomiques garantissent d'être exécutée en une fois: le processeur n'est jamais alloué à un autre processus pendant leur exécution.

Conditions de course (*race conditions*)

Afin de régler les problèmes de concurrence, il faut utiliser des mécanismes de coordination tel que:

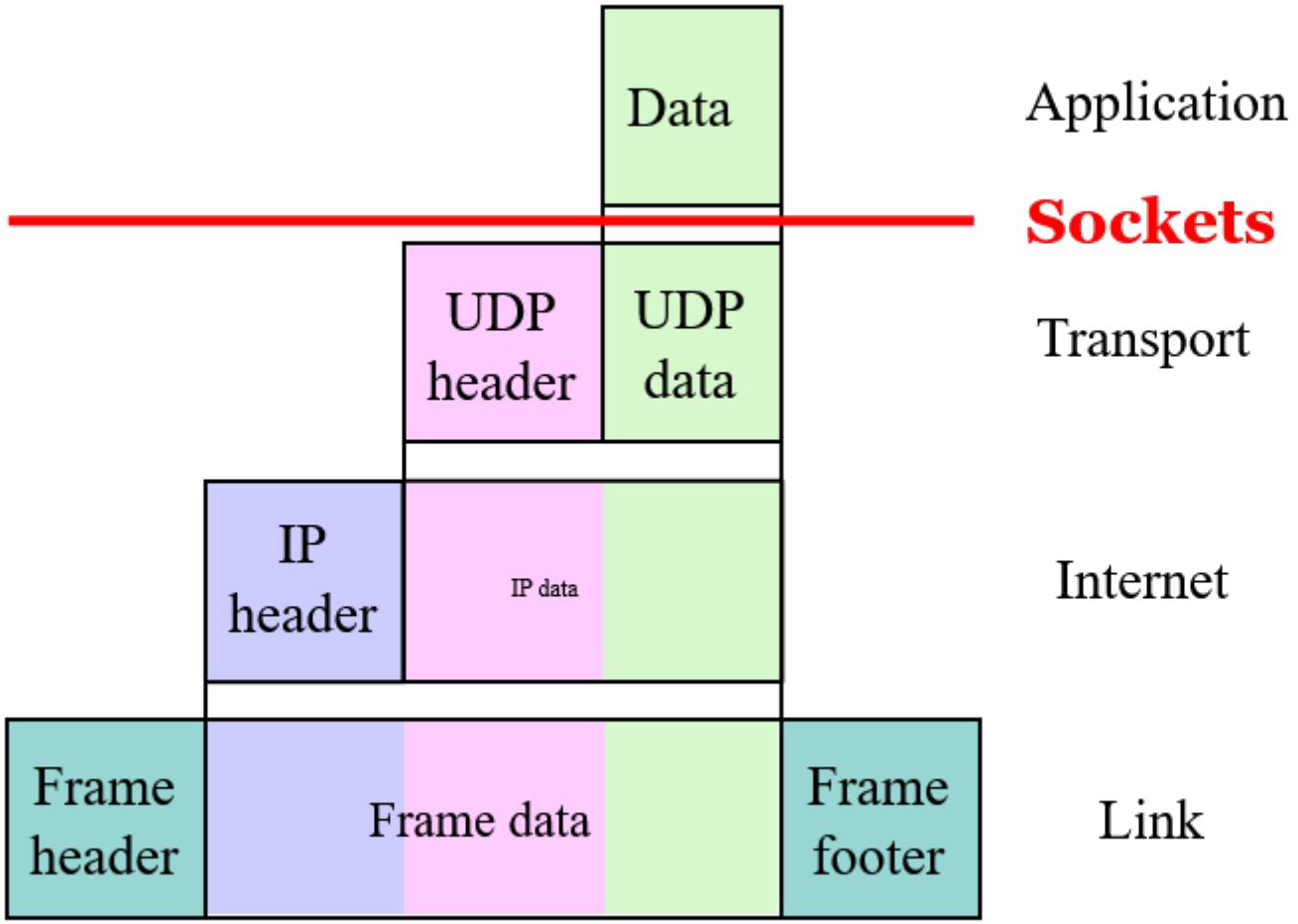
- des variables communes;
- des signaux;
- des mécanismes dédiés:
  - la mémoire partagée étant représentée par un descripteur de fichier, il est possible d'utiliser les lock comme pour les fichiers;
  - il existe d'autre mécanismes comme les sémaphores ou les mutex:

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);

pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);
```

Sockets:
les communications réseaux

Architecture réseau



Sockets

Abstraction d'un canal de communication à travers le réseau

Descripteur de fichier:

- Lecture avec `read()`
- Écriture avec `write()`
- Pas d'accès aléatoire possible

Modèle client-serveur

Domaine d'adressage

Un socket peut être lié à une adresse.

Il existe deux domaines d'adressages:

internet domain	communication réseau (AF_INET)
unix domain	communication locale (AF_UNIX)

Types de sockets

Il existe plusieurs types de sockets dont:

par flot	avec connexion, p.ex. TCP (SOCK_STREAM)
par datagramme	sans connexion, p.ex. UDP (SOCK_DGRAM)
brut	sans protocole de transport (SOCK_RAW)

Nous couvrirons surtout le premier type.

Adressage Internet

L'adresse est composée d'une adresse IP et d'un numéro de port (16 bits).

Structures IPv4 (voir `man 7 ip`):

```
struct sockaddr_in {
    sa_family_t     sin_family; // famille: AF_INET
    in_port_t       sin_port;   // port: big-endian
    struct in_addr  sin_addr;   // adresse internet
};

struct in_addr {
    uint32_t        s_addr;     // adresse ip: big-endian
}
```

Adressage Internet

La fonction suivante permet d'obtenir une adresse IP valide:

```
int inet_pton(int af, const char *src, void *dst);
```

`af`	Famille d'adresse soit AF_INET, soit AF_INET6
`src`	Représentation de l'adresse (par exemple 192.168.1.1)
`dst`	Pointeur vers une structure in_addr ou in6_addr à initialiser

Retourne:

1	succès
0	adresse non valide
-1	famille non valide

Adressage Internet

La fonction suivante permet d'obtenir la représentation d'une adresse IP:

```
const char *inet_ntop(int af, const void *src, char *dst,
socklen_t size);
```

`af`	Famille d'adresse, soit AF_INET ou AF_INET6
`src`	Pointeur vers une structure in_addr ou in6_addr initialisée
`dst`	Pointeur vers un buffer (pour obtenir la représentation)
`size`	Taille du buffer

Retourne NULL en cas d'erreur (cf `errno`)

Adressage Internet

On peut convertir un entier en un numéro de port valide grâce à:

```
uint16_t htons(uint16_t hostshort);
```
ports codés sur 16 bits

```
uint32_t htonl(uint32_t hostlong);
```
ports codés sur 32 bits

Le résultat est dans le bon byte-order (Big-Endian).

Clients TCP

Le fonctionnement d'un client TCP est le suivant:

1) Crée un socket (`socket`)
2) Connecte un socket à un serveur (`connect`)
3) Lecture/écriture à partir du socket (`read/write`)
4) Ferme le socket (`close`)

Clients TCP

Pour créer un socket, on utilise l'appel système:

```
int socket(int domain, int type, int protocol);
```

`domain`	famille d'adresse (AF_INET[1], AF_INET6[1], AF_UNIX, …)
`type`	type de communication (SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, …)
`protocol`	protocole de transport, entrer 0 pour UDP et TCP

Retourne soit un descripteur de fichier, soit -1 (voir `errno`).

[1] ici PF_INET devrais être utilisé mais AF_INET est toléré et souvent utilisé à la place (voir `man`)

Clients TCP

L'appel système suivant permet d'initier une connexion:

```
int connect(int sockfd, const struct sockaddr *addr,
socklen_t addrlen);
```

`sockfd`	Descripteur de fichier du socket
`addr`	Pointeur vers une adresse
`addrlen`	Longueur de la structure

Retourne 0 en cas de succès, -1 sinon (voir `errno`).

Client TCP: exemple

```
struct sockaddr_in address;
memset(&address, 0, sizeof(address));
inet_pton(AF_INET, "192.168.1.1", &(address.sin_addr));
address.sin_family = AF_INET;
address.sin_port = htons(8080);

int sock = socket(AF_INET, SOCK_STREAM, 0);

connect(sock, (struct sockaddr *) &address, sizeof(address));

read(sock, ...)
write(sock, ...)
```

Serveurs TCP

Le fonctionnement d'un serveur TCP est le suivant:

1) Crée un socket serveur (`socket`)
2) Attache le socket serveur à une adresse (`bind`)
3) Configure le socket comme socket d'écoute (`listen`)

4) Accepte une connexion et obtient un socket client (`accept`)
5) Lecture/écriture à partir du socket client (`read/write`)
6) Ferme le socket client (`close`)

Répète les étapes de 4 à 6 si nécessaire

7) Ferme le socket serveur (`close`)

Serveurs TCP

On utilise l'appel système `socket()` pour créer un socket serveur, exactement comme pour le client.

On lie un socket à une adresse (interface locale) avec l'appel système:

```
int bind(int sockfd, const struct sockaddr *addr,
socklen_t addrlen);
```

`sockfd`	Descripteur du socket
`addr`	Pointeur vers l'adresse à lier
`addrlen`	Taille de la structure d'adresse

Retourne 0 en cas de succès, -1 sinon (voir `errno`)

Serveurs TCP

L'appel système `listen()` permet de marquer un socket comme étant passif, c'est à dire un socket permettant d'accepter des connections:

```
int listen(int sockfd, int backlog);
```

`sockfd`	Descripteur du socket
`backlog`	Taille maximum de la queue de connections en attente

Retourne 0 en cas de succès, -1 sinon (voir `errno`)

Serveurs TCP

L'appel système suivant permet d'accepter une connection:

```
int accept(int sockfd, struct sockaddr *addr,
socklen_t *addrlen);
```

`sockfd`	Descripteur du socket (doit être passif)
`addr`	Structure garnie avec les informations du client
`addrlen`	Longueur de la structure

Retourne un nouveau descripteur de fichier permettant de communiquer avec le client en cas de succès, et -1 sinon (cf. `errno`)

Bloque jusqu'à la prochaine connexion entrante ou en extrait une de la queue.

Serveurs TCP

L'utilisation de INADDR_ANY permet de se lier à toutes les interfaces réseaux de la machine.

On utilise la fonction `htonl()` pour obtenir une adresse numérique valide:

```
address.sin_addr.s_addr = htonl(INADDR_ANY)
```

Serveur TCP: exemple

```
struct sockaddr_in address;
memset(&address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_addr.s_addr = htonl(INADDR_ANY);
address.sin_port = htons(8080);

int sock = socket(AF_INET, SOCK_STREAM, 0);
bind( sock, (struct sockaddr *) &address, sizeof(address) );
listen(sock, 5);

while(1) {
    struct sockaddr_in clientAddress;
    unsigned int clientLength = sizeof(clientAddress);
    int clientSock = accept(serverSock,
                            (struct sockaddr *) &clientAddress,
                            &clientLength);
    /* Lectures/ecritures sur clientSock (read/write)... */
    close(clientSock);
}
```

Serveur de fichier (avec client)

Code commun: file_transmission.c
file_transmission.h

Code du serveur: file_server.c

Code du client: file_client.c

file_transmission.h

```
#ifndef FILE_TRANSMISSION_H
#define FILE_TRANSMISSION_H

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

#define BUFF_SIZE 32

/* Affiche le message d'erreur et termine l'exécution */
static inline void die(char *issue) {
  perror(issue);
  exit(EXIT_FAILURE);
}

/* Copie les données d'un descripteur de fichier vers un autre.
   Retourne un nombre négatif en cas d'erreur */
int copy(int from, int to);


#endif /* FILE_TRANSMISSION_H */
```

file_transmission.c

```
#include "file_transmission.h"
#include <errno.h>
#include <unistd.h>

int copy(int from, int to) {
  char buf[BUFF_SIZE];
  ssize_t nread;

  while( nread = read(from, buf, sizeof buf), nread > 0 ) {
    char *out_ptr = buf;
    ssize_t nwritten;
    do {
      nwritten = write(to, out_ptr, nread);
      if (nwritten >= 0) {
        nread -= nwritten;
        out_ptr += nwritten;
      } else if (errno != EINTR) {
        return -1;
      }

    } while (nread > 0);
  }

  return nread;
}
```

file_server.c (1/3)

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "file_transmission.h"

#define MAX_FULLPATH 1024
#define MAX_NAME 255
#define MAX_PENDING 256

void prepare_address( struct sockaddr_in *address,  int port ) {
  size_t addrSize = sizeof( address );
  memset(address, 0, addrSize);
  address->sin_family = AF_INET;
  address->sin_addr.s_addr = htonl(INADDR_ANY);
  address->sin_port = htons(port);
}

int makeSocket( int port ) {
  struct sockaddr_in address;
  int sock = socket(AF_INET, SOCK_STREAM, 0);
  if( sock < 0 ) {
    die("Failed to create socket");
  }
  prepare_address( &address, port );
  if( bind( sock,
            (struct sockaddr *) &address,
            sizeof(address)
            ) < 0  )
    {
      die("Failed to bind the server socket");
    }
  if (listen(sock, MAX_PENDING) < 0) {
    die("Failed to listen on server socket");
  }
  return sock;
}
```

file_server.c (2/3)

```
void handleClient( int clientSock, const char *path ) {
  char fullPath[MAX_FULLPATH];
  int pathLen = strlen( path );
  int file;
  int nRead;
  printf( path );
  strncpy( fullPath, path, MAX_FULLPATH );    /* TODO: Proteger le slash */
  nRead = read( clientSock, (fullPath+pathLen), MAX_NAME );
  if( nRead <= 0 ) {
    die( "WTF? 1" );
  }
  printf( "Requested file: %s\n", fullPath );
  file = open( fullPath, O_RDONLY, 0 );
  if( copy( file, clientSock ) < 0 ) {
    perror( "Failed to send the file" );
  }
  close( file );
  close( clientSock );
}

void run( int serverSock, const char *path ) {
  while( 1 ) {
    struct sockaddr_in clientAddress;
    unsigned int clientLength = sizeof(clientAddress);
    int clientSock;
    printf( "Waiting for incoming connections\n");
    clientSock =
      accept(serverSock, (struct sockaddr *) &clientAddress, &clientLength );
    if( clientSock < 0 ) {
      die("Failed to accept client connection");
    }
    printf( "Client connected: %s\n", inet_ntoa(clientAddress.sin_addr));
    handleClient(clientSock,path);
  }
}
```

file_server.c (3/3)

```
int main( int argc, char **argv ) {
  int servSock;
  char *path;
  int port;

  if (argc != 3) {
    fprintf(stderr, "USAGE: %s <port> <path>\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  port = atoi(argv[1]);
  path = argv[2];

  servSock = makeSocket( port );

  printf( "Server running on port %d at dir '%s'\n", port, path );

  run( servSock, path );

  close(servSock);

  return EXIT_SUCCESS;
}
```

file_client.c (1/2)

```
#include "file_transmission.h"
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void prepare_address( struct sockaddr_in *address, const char *host, int port ) {
  size_t addrSize = sizeof( address );
  memset(address, 0, addrSize);
  address->sin_family = AF_INET;
  inet_pton( AF_INET, (char*) address, &(address->sin_addr) );
  address->sin_port = htons(port);
}

int makeSocket( const char *host, int port ) {
  struct sockaddr_in address;
  int sock = socket(AF_INET, SOCK_STREAM, 0);
  if( sock < 0 ) {
    die("Failed to create socket");
  }
  prepare_address( &address, host, port );
  if( connect(sock, (struct sockaddr *) &address, sizeof(address)) < 0) {
    die("Failed to connect with server");
  }
  return sock;
}
```

file_client.c (2/2)

```
int main(int argc, char *argv[]) {
  int sock,file;
  char *host;
  int port;
  char *filename;
  int filenameLength;

  if (argc != 4) {
    fprintf(stderr, "USAGE: %s <host> <port> <filename>\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  host = argv[1];
  port = atoi(argv[2]);
  filename = argv[3];
  filenameLength = strlen( filename );

  /* Open the file */
  if( (file = open(filename,O_WRONLY|O_CREAT|O_TRUNC,0600)) < 0 ) {
    die("Failed to create the local file");
  }

  sock = makeSocket( host, port );

  /* Send the filename */
  if( write(sock,filename,filenameLength) != filenameLength ) {
    die( "Cannot send the filename to retrieve" );
  }

  /* Receive the file */
  if( copy( sock, file ) < 0 ) {
    die( "Cannot receive the file" );
  }

  close(sock);
  close(file);


  exit(EXIT_SUCCESS);
}
```

Serveur de fichier (avec client): problèmes

Dans l'exemple précédent, il faudrait:

- Avertir le client qui demande un fichier qui n'existe pas
- Permettre au client d'obtenir la liste des fichiers disponibles

→ Nécessité de définir un protocole

L'exemple précédent contient une faille de sécurité importante: laquelle?

Serveur de fichier (avec client): problèmes

La faille de sécurité importante consiste en une vulnérabilité à l'**attaque par traversée de répertoires (*path traversal attack*)**, qui permet à un client de demander et de télécharger des fichiers arbitraires sur le serveur, même en dehors du dossier prévu.

Un client malveillant peut demander un fichier p.ex. `../../../etc/passwd` accédant ainsi à des fichiers sensibles du système, extérieurs au dossier autorisé.

Implémentation des sockets

Les sockets sont implémentés au niveau de l'OS, par presque tous les OS.

La grande majorité des langages de programmation ont une librairie permettant d'utiliser les sockets.

Les sockets permettent la communication entre processus écrits dans des langages différents.

Les sockets permettent la communication entre OS différents.

Implémentation des sockets: Python

```
#CLIENT
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("www.mcmillan-inc.com", 80))

#SERVEUR
serversocket = \
   socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind((socket.gethostname(), 80))
serversocket.listen(5)
```

Implémentation des sockets: Java

```
try {
    serverSocket = new ServerSocket(4444);
}
catch (IOException e) {
    System.out.println("Could not listen on port: 4444");
    System.exit(-1);
}
Socket clientSocket = null;
try {
    clientSocket = serverSocket.accept();
}
catch (IOException e) {
    System.out.println("Accept failed: 4444");
    System.exit(-1);
}
```

recv() / send()

On peut utiliser les appels systèmes suivant à la place de `read()` et de `write()` lorsqu'on utilise des sockets:

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t send(int sockfd, const void *buf, size_t len,
int flags);
```

Le paramètre `flags` permet de passer des paramètres supplémentaires pour contrôler finement la transmission.

`recv(sockfd, buf, len, 0)` est équivalent à
`read(sockfd, buf, len)`

`send(sockfd, buf, len, 0)` est équivalent à
`write(sockfd, buf, len)`

sendfile()

Sous Linux, on peut remplacer une paire read/write ou send/recv par un appel à `sendfile()` (non-POSIX) qui permet de rester dans l'espace du noyau:

```
ssize_t sendfile(int out_fd, int in_fd, off_t *offset,
size_t count)
```

`out_fd`	descripteur ouvert en écriture (devait être un socket jusqu'à Linux 2.6.33)
`in_fd`	descripteur ouvert en lecture (ne peut pas être un socket)
`offset`	l'offset en lecture (peut être NULL)
`count`	taille à envoyer

UDP

UDP n'a pas de mécanisme d'établissement de liaison (*handshake*), il est **orienté transaction** et pas **orienté connexion**.

Différences avec TCP:
Le client n'a pas besoin d'établir une connexion.
Le serveur n'a pas besoin d'écouter et d'accepter une connexion.

Comme on n'établit pas de connexion, on utilisera les appels systèmes suivants pour effectuer des lectures/écritures:

`sendto()` pour envoyer des données vers une adresse

`recvfrom()` pour recevoir des données depuis une adresse

UNIX socket

Un socket Unix permet d'établir une communication locale entre deux processus au moyen d'un inode.
Il faut passer le domaine AF_UNIX à l'appel système socket.
L'adressage est différent, mais tout le reste est identique aux sockets internet.

Adresse Unix (`man 7 unix`):

```
#define UNIX_PATH_MAX 108
struct sockaddr_un {
    sa_family_t sun_family;          /* Family: AF_UNIX*/
    char sun_path[UNIX_PATH_MAX];  /* Path on filesystem */
};
```

Domain name resolution

La fonction suivante permet de résoudre le nom de domaine d'un service:

```
int getaddrinfo(const char *node,
                const char *service,
                const struct *hints,
                struct addrinfo **res);
```

```
struct addrinfo {
    int              ai_flags;
    int              ai_family;
    int              ai_socktype;
    int              ai_protocol;
    socklen_t        ai_addrlen;
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next; // This is a linked list
};
```

Modules kernel

Périphérique (*device*)

Un **périphérique (*device*)** est **un matériel connecté à une unité centrale**.

Exemples de périphériques: une souris, un terminal (clavier + écran), un disque dur, un lecteur DVD, un capteur de température, un affichage LED, ...

Sous Linux les périphériques sont souvent associés à des **fichiers de périphériques (*device files*)** qui permettent de communiquer avec le matériel.

Exemples de device files:

`/dev/sda` (disque dur)
`/dev/video0` (webcam)
`/dev/ttyS0` (port série)
...

Périphérique virtuel (*virtual device*)

Un **périphérique virtuel (*virtual device*)** est **un fichier de périphérique qui n'est pas associé à un hardware**. Cela permet par exemple de faire de l'émulation de périphérique.

Exemples de périphériques virtuels:

`/dev/null`	émule un trou noir à octets
`/dev/zero`	génère des zéros
`/dev/urandom`	génère des nombres aléatoires
`/dev/tty0`, `/dev/tty1`, ...	consoles virtuelles
`/dev/pts/0`, `/dev/pts/1`, ...	pseudo-terminaux
`/dev/loop1`, `/dev/loop2`, ...	fichiers émulant disques durs ou *loop devices*

Nous implémenterons un périphérique virtuel en TP.

Types de périphérique

Les périphériques peuvent être de deux types:

Character devices:

Le noyau communique avec ces périphériques octet par octet: on peu lire et écrire le nombre d'octets que l'on souhaite.
Ce sont les périphériques les plus communs (ports séries, terminaux, souris, webcam, ...)

Block devices:

On ne peut lire des données que par blocs (souvent des blocs de 4K).
C'est par exemple le cas des disques dures ou des CD-ROM qui nécessitent de transiter des quantités d'information importantes rapidement.

Modules et drivers

Un **module** est **un code compilé qui est ajouté au noyau alors que ce dernier est en exécution** (drivers, protocoles réseaux, systèmes de fichiers, transformations cryptographiques, etc.)

Un ***driver*** **(ou pilote)** est **une couche logicielle qui gère un périphérique**.
On distingue les drivers:

- intégrés au noyau: il s'agit de drivers qui sont compilés en même temps que le noyau (e.g. terminaux);
- implémentés sous forme de module, ce qui corresponds à la plupart des drivers aujourd'hui.

Linux étant un **système monolithique**, les drivers sont exécutés dans l'espace noyau (*kernel space*). Il existe des systèmes d'exploitations de type **micro-noyau** où les drivers sont exécutés dans l'espace utilisateur (c.f. Minix).

Drivers

Le but d'un driver est de gérer les I/O d'un périphérique. Il devra donc être capable de répondre aux demandes des programmes de l'espace utilisateur. Pour cela il doit pouvoir répondre aux appels systèmes suivant:

`open()`	initialiser un nouveau canal de communication permettant d'accéder au périphérique;
`read()`	lire des données du périphérique;
`write()`	envoyer des données au périphérique;
`close()`	fermer le canal de communication vers le périphérique, uniquement s'il n'y a plus de descripteur de fichier référençant ce canal;
`lseek()`	se déplacer dans le périphérique si cela est possible (rarement pour un périphérique physique);

etc.

modprobe

La commande `modprobe` permet de gérer les modules (voir aussi les fichiers de configuration `/etc/modules-load.d/`).

`modprobe module [param1=value1 param2=value2]`
Insère un module (avec ses paramètres) dans le noyau, si le module est présent dans `/lib/modules/linux-version/`. Insère aussi les dépendances

`insmod /path/to/my/module.ko [param1=value1 param2=value2]`
Insère un module d'un autre dossier (sans ses dépendances) dans le noyau

`modprobe -r module` `rmmod module`	Retire un module du noyau
`lsmod`	Liste les module chargés par le noyau
`modinfo module`	Obtient des informations sur un module

Identification des périphériques

Le noyau identifie chaque périphérique grâce à une paire de nombres (majeur, mineur):

le **nombre majeur** indique le driver qui permettra d'exploiter le périphérique;
le **nombre mineur** indique à quel périphérique du driver l'on fait référence.

```
$ ls /dev/
brw-rw----   1 root   disk         8,   0 Mar 25 15:03 sda
brw-rw----   1 root   disk         8,   1 Mar 25 15:03 sda1
brw-rw----   1 root   disk         8,   2 Mar 25 15:03 sda2
brw-rw----+  1 root   optical     11,   0 Mar 25 15:03 sr0
```

Ici, `/dev/sr0` est un DVD géré par un autre driver.

Identification des périphériques

En C c'est le type `dev_t`, accompagné de macros, qui permet de gérer ces identifiants:

```
#include <linux/types.h>
#include <linux/kdev_t.h>
dev_t dev = MKDEV(int major, int minor) // crée des identifiants de device
MAJOR(dev_t dev) // obtient le major number d'un device
MINOR(dev_t dev) // obtient le minor number d'un device
```

mknod

La commande `mknod` permet de créer un fichier de périphérique associé aux identifiants (majeur, mineur):

```
mknod [-m mode] filename type major minor
```

`filename`	le nom du fichier à créer (e.g. `/dev/mydevice`)
`type`	type de périphérique (i.e. 'b' pour block, 'c' pour character)
`major`, `minor`	identifiants du périphérique

Kernel sources

Les modules sont liés aux sources du noyau. Il est donc nécessaire d'avoir les sources pour pouvoir compiler un module.
On peut:

- télécharger les sources et les compiler (https://www.kernel.org/);
- utiliser le package manager pour télécharger les sources (généralement dans `/usr/lib/modules/linux-version`)

Kernel makefile

La compilation de modules fait référence à des makefile du noyau. Le makefile d'un module a donc un format particulier:

```
LINUX_VER=$(shell uname -r)

# Votre code est dans my-module.c
# Indique au noyau que un fichier objet doit être compilé pour ce module
obj-m := my-module.o
# Voir module-objs pour des modules à plusieurs fichiers

build:
    # Fait l'hypothèse que les sources du noyau sont dans /lib/modules/...
    # -C execute le makefile du noyau dans le dossier spécifié
    # M=path indique au noyau le dossier du module (pour obtenir les obj-m)
    # puis le noyau execute la target modules qui compile les objets dans obj-m
    make -C /lib/modules/$(LINUX_VER)/build M=$(PWD) modules
```

Comme les modules résident dans le noyau, aucune fonction de la GLib (bibliothèque libre pour le langage C) n'est disponible.
Donc pas de `malloc()`, `calloc()`, `printf()`, ...

Heureusement les programmeurs du noyau ont rendu disponibles d'autres fonctions:
Il existe `printk()`, `kmalloc()`, ...

En fait beaucoup d'autres fonctions sont (re)-implémentées comme `strncpy()`.
Toutefois leur comportement peut varier de celui de la norme ANSI.

Une documentation de l'API du noyaux linux est disponible:
https://www.kernel.org/doc/htmldocs/kernel-api/

printk()

Très utilisé pour le debugging, `printk()` permet d'envoyer des messages au système de log du noyau suivant un format similaire à `printf()`:

```
printk(KERN_ERR "Error (%d): %s", number, string);
```

KERN_ERR est le niveau de log et peut prendre d'autre valeurs (KERN_EMERG, KERN_ALERT, KERN_CRIT, KERN_ERR, KERN_WARNING, KERN_INFO, KERN_DEBUG, ...)

Ces messages

- apparaissent sur la console (pas sur les fenêtres des pseudo-terminaux!) si le niveau de log est suffisant;
- sont visualisables par la commande `dmesg`.

kmalloc()

```
#include <linux/slab.h>
void * kmalloc(size_t size, gfp_t flags)
void kfree(void *ptr)
```

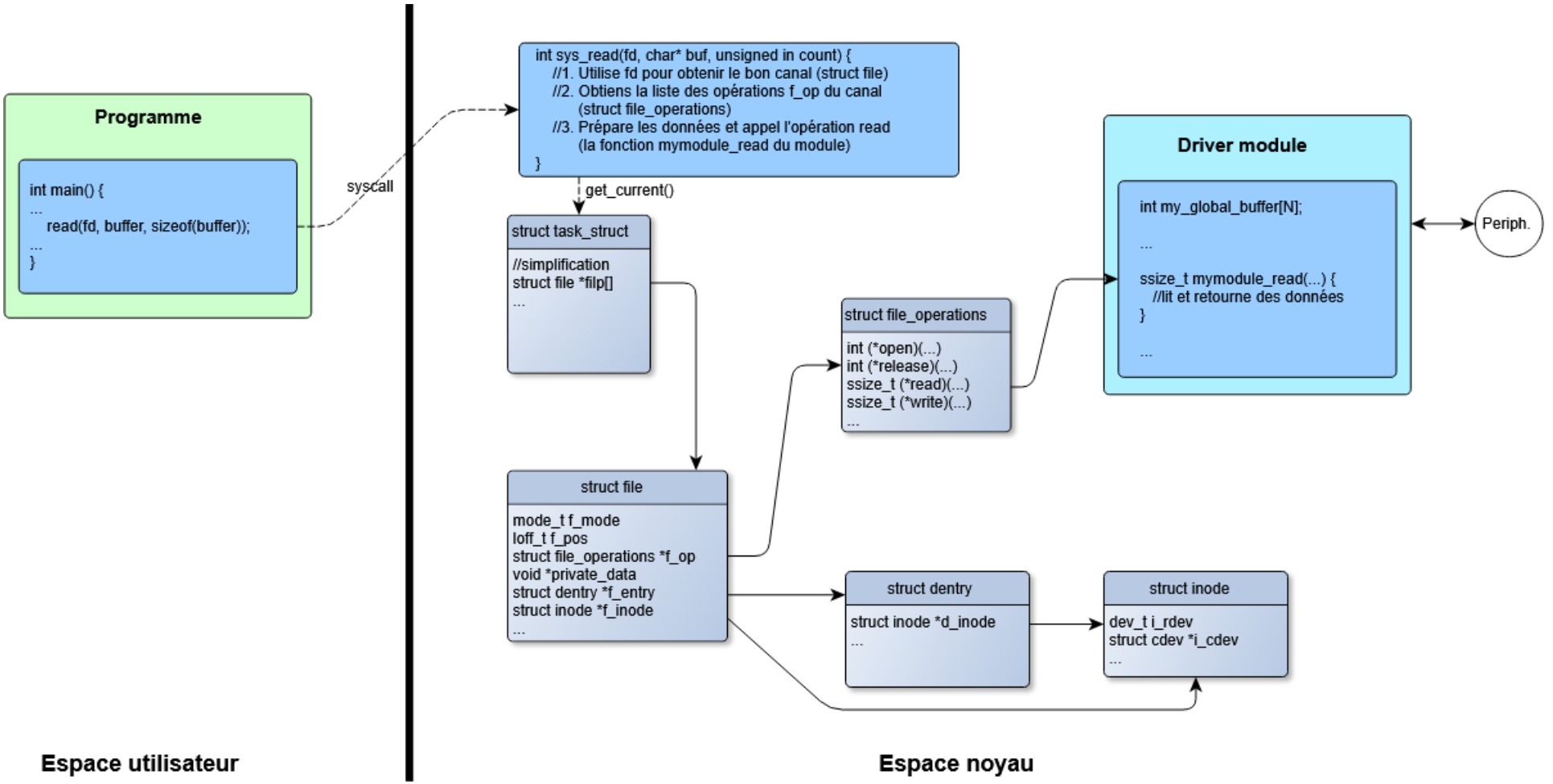
`size` taille en octet de la mémoire à allouer

`flags` type de mémoire à allouer

Retourne un pointeur `ptr` sur la mémoire allouée; le libérer avec `kfree()`

Dans la majorité des cas le type de mémoire utilisé est GFP_KERNEL, cet appel implique que le processus courant peut-être mis en attente de la ressource mémoire (voir GFP_NOWAIT, GFP_ATOMIC pour des alternatives).

Vue générale d'un module



Déclaration et configuration d'un périphérique (écriture d'un module)

Informations sur le module

Tout module va inclure les fichiers d'entête suivants:

```
#include <linux/module.h>
#include <linux/init.h>
```

De plus des macros permettent de donner des informations sur le module:

```
MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("Guillaume Chanel <Guillaume.Chanel@unige.ch>");
MODULE_DESCRIPTION("This is a wonderful module");
MODULE_VERSION("1.0");
// Voir également MODULE_ALIAS, MODULE_DEVICE_TABLE
```

La macro MODULE_LICENSE est particulièrement importante et peut prendre plusieurs valeurs. Sans cette macro le module est considéré comme non-opensource et on dit que le noyau est "contaminé" (*tainted*).

Initialisation et terminaison du module

Notre module devra implémenter deux fonctions qui seront appelées lorsque le module est chargé et retiré du noyau:

- l'utilisation de `modprobe` ou `insmod` appellera une fonction d'initialisation;
- l'utilisation de `modprobe -r` ou `rmmod` appellera une fonction de terminaison.

Il ne faut pas confondre ces fonctions avec l'ouverture (`open()`) et la fermeture (`close()` / `release()`) d'un périphérique.

Initialisation du module

Une fonction doit être dédiée à l'initialisation du module:

```
static int __init my_wonderful_init(void) {
    ...
}
module_init(my_wonderful_init)
```

Cette fonction doit préparer les périphériques, ce qui veux dire:

- **inscrire des périphériques**, c'est-à-dire réserver des identifiants pour les périphériques (majeur / mineurs);
- initialiser les données nécessaires au bon fonctionnement du périphérique;
- ajouter le périphérique au système d'exploitation.

Lorsqu'un périphérique est ajouté au système, il peut alors être utilisé immédiatement, même si la fonction d'initialisation n'est pas terminée. Il faut donc que le périphérique sois prêt à fonctionner avant de l'ajouter au noyau.

Initialisation du module

Une fonction doit être dédiée à l'initialisation du module:

```
static int __init my_wonderful_init(void) {
    ...
}
module_init(my_wonderful_init)
```

`__init` est un mot clef du noyau qui s'assure que la fonction sera libérée après la phase d'initialisation. Cela permet d'économiser l'espace mémoire du noyau.

`module_init()` est une macro du noyau qui permet de lui indiquer où se trouve la fonction d'initialisation.

En cas d'erreur dans la fonction d'initialisation il faut:

- penser à nettoyer ce qui a été fait (voir terminaison du module);
- retourner le code d'erreur donné par la fonction fautive.

Terminaison du module

Une fonction doit être dédiée à la terminaison du module:

```
static void __exit my_wonderful_finale(void) {
    ...
}
module_exit(my_wonderful_finale)
```

Cette fonction doit:

- retirer les périphériques déjà ajoutés au système;
- éventuellement libérer des données allouées;
- **désinscrire des périphériques**, c'est-à-dire libérer les identifiants (majeur / mineurs).

En cas d'erreur lors de l'initialisation il faut aussi effectuer ces trois étapes pour tout périphérique enregistré.

Inscription des périphériques

Pour inscrire un périphérique caractère:

```
#include <linux/fs.h>
// Pour obtenir un numéro majeur en spécifiant le premier mineur et le nombre de périphériques
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);
// Pour spécifier les numéros majeur, mineur et le nombre de périphériques
int register_chrdev_region(dev_t dev, unsigned int count, char *name);
```

`dev` Numéro de device retourné par la fonction (majeur + premier mineur); on peut aussi le spécifier avec `register_chrdev_region()`

`firstminor` Premier numéro mineur référençant notre premier périphérique, généralement 0

`count` Nombre de périphériques que l'on souhaite inscrire; il déterminera le plus haut nombre mineur de nos périphériques

`name` Nom de notre module / ensemble de périphériques

Retourne 0 en cas de succès, un code d'erreur sinon.

Inscription des périphériques

Rappel: on peu gérer le type `dev_t` avec des macros:

```
// référence notre ième périphérique
MKDEV(MAJOR(dev), MINOR(dev) + i)
```

Une fois inscrit, on peut retrouver le numéro majeur de nos périphériques dans `/proc/devices`:

```
$ cat /proc/devices
Character devices:
...
137 mydevice
```

Désinscription des périphériques

Pour désinscrire un périphérique lors de la terminaison d'un module ou d'un erreur on utilise:

```
void unregister_chrdev_region(dev_t first,
unsigned int count);
```

`first`	Identifiant du premier périphérique
`count`	Nombre de périphériques

Définition d'un périphérique

En général, on définie une structure globale qui va contenir les données globales des nos périphériques.

```
struct mydevice {
    char *string_data; /* ou tout autre donnée que l'on souhaite */
    int int_data;      /* ou tout autre donnée que l'on souhaite */
    struct cdev mycdev;
}
```

Cette structure contient une structure `cdev` qui représente notre périphérique caractère dans le noyau.

```
struct cdev {
    ...
    struct file_operations ops;
    ...
}
```

C'est la structure `file_operations` qui va nous permettre de définir les actions à réaliser lorsque nous recevons des appels système.

Initialisation d'un périphérique

Avant d'ajouter le périphérique, il faut l'initialiser:

```
#include <linux/cdev.h>
void cdev_init(struct cdev *cdev,
struct file_operations *fops);
```

`cdev`	Représente notre périphérique caractère dans le noyau
`fops`	Doit être initialisée correctement (voir section suivante "répondre aux appels système")

Ajout d'un périphérique

Pour ajouter un périphérique on utilise:

```
int cdev_add(struct cdev *dev, dev_t num,
unsigned int count);
```

`dev`	Structure initialisée avec `cdev_init`
`num`	Numéro du périphérique
`count`	Généralement 1

Retourne 0 en cas de succès, un code d'erreur sinon.

`count` est utile dans les cas où un périphérique correspond à plusieurs numéros de périphériques.

Suppression d'un périphérique

Pour supprimer un périphérique on utilise:

```
void cdev_del(struct cdev *dev);
```

`dev` Structure initialisée avec `cdev_init`

Paramètres des modules

La commande `modprobe` peut passer des paramètres de configuration à notre module:

```
$ modprobe module [param1=value1 param2=value2]
```

Dans notre module ces paramètres correspondent à des variables globales qui sont rendues disponibles par des macros:

```
static int param1 = 0;
static char *param2 = "default value";
module_param(param1, int, S_IRUGO);
module_param(param2, charp, S_IRUGO);
```

Paramètres des modules

```
module_param(name, type, perms);
```

`name` Nom de la variable

`type` Type de la variable: peut être `bool`, `charp`, `int`, `long`, `short`, `uint`, `ulong`, `ushort`

`perms` Chaque paramètre du module apparaît comme une entrée dans le système de fichier spécial `sysfs`. Les droits de ce fichier sont définis par le paramètre `perms`:

S_IRUGO Read permission for user / group / others

S_IWUSR Write permission for user root

etc. (les valeur habituelles fonctionnent)

Répondre aux appels système

file

```
#include <linux/fs.h>
struct file {
    mode_t f_mode;
    loff_t f_pos;
    unsigned int f_flags;
    struct file_operations *f_op;
    void *private_data;
    struct dentry *f_dentry;
}
```

La structure `file` représente un **canal de communication / un fichier ouvert** (peut-être un fichier de périphérique).

Cette structure est crée lors d'un appel `open()` et supprimée lorsque tous les descripteurs de fichiers ont été fermés (dernier `close()`).
Il s'agit donc d'une **entrée de la table des canaux**, comme vu précédemment.

file

```
#include <linux/fs.h>
struct file {
    mode_t f_mode;
    loff_t f_pos;
    unsigned int f_flags;
    struct file_operations *f_op;
    void *private_data;
    struct dentry *f_dentry;
}
```

Quelques champs importants:

`f_pos`	Position actuelle dans le fichier
`f_op`	Pointeur vers la liste des fonctions permettant de manipuler le fichier (lire, écrire, etc.)
`private_data`	Pointeur pour stocker des données dépendant du type de canal (e.g. un pointeur sur notre device `struct mydevice`)

file_operations

C'est la structure `file_operations` qui détermine comment notre module réagit aux appels systèmes. Cette structure est généralement déclarée de manière globale dans le module.

```
#include <linux/module.h>
#include <linux/fs.h>
struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .open = my_open,
    .release = my_release,
    .read = my_read,
    .write = my_write,
    .llseek = my_lseek,
    ...
};
```

La macro THIS_MODULE retourne un pointeur sur notre module.

file_operations

Chaque élément de la structure est un pointeur sur une fonction de notre module (sauf `owner`):

```
int my_open(struct inode *i, struct file *filp) {
    // Actions à réaliser pour ouvrir notre périphérique
}

ssize_t my_read(struct file *filp, char __user *buffer, size_t,
loff_t *offset) {
    // Actions à réaliser pour lire notre périphérique
}
```

open

```
int (*open)(struct inode *inode, struct file *filp);
```

`open` est appelée à chaque ouverture du périphérique.
Son but est de préparer le driver à traiter les opérations futures:

- retourner un erreur si le driver n'est pas prêt;
- créer les données privées `filp->private_data` (si nécéssaire).

On peut mettre l'entrée `open` de la structure `f_ops` à NULL. Dans ce cas le noyau crée simplement la structure `file` et notre driver n'est pas notifié d'une ouverture.

Exemples de préparation de device:

- initialiser un buffer de communication;
- envoyer une commande d'initialisation au hardware pour le premier `open`.

release

```
int (*release) (struct inode *inode, struct file *filp);
```

`release` est appelée uniquement lorsque il n'y a plus de descripteurs de fichiers pointant sur `filp` (un `release` pour un `open`).

Toute mémoire initialisée par `open` doit être libérée par `release`.

On peut mettre l'entrée `release` de la structure `f_ops` à NULL. Dans ce cas le noyau libère simplement la structure `file` et notre driver n'est pas notifié d'une fermeture.

read

```
ssize_t (*read) (struct file *filp, char __user *buff,
size_t len, loff_t *f_pos);
```

`read` est appelé à chaque lecture du périphérique (voir l'appel système correspondant).

`filp`	Fichier concerné par la lecture
`buff`, `len`	Buffer à remplir avec les données lues et la quantité de données demandées; **si on le souhaite on peu lire moins de données que demandé**
`f_pos`	Position courante dans le fichier; **il faut mettre à jour cette position en fonction du nombre de bytes effectivement lus**

Doit retourner la quantité de bytes effectivement lus.

read

```
ssize_t (*read) (struct file *filp, char __user *buff,
size_t len, loff_t *f_pos);
```

Comme indiqué par la macro `__user`, **le pointeur `buff` est une adresse de l'espace mémoire du processus; il faut donc convertir cette adresse dans l'espace d'adressage du noyau.** Pour cela on utilise:

```
#include <asm/uaccess.h>
unsigned long copy_to_user(void __user *to, const void *from,
unsigned long count);
```

`to`	Adresse du processus vers laquelle copier les données
`from`	Données de l'espace noyau à copier
`count`	Quantité de bytes à copier

Retourne la quantité de bytes qu'il reste à copier; si elle est différente de 0 alors un problème à été rencontré.

write

```
ssize_t (*write) (struct file *filp, const char __user *buff,
size_t len, loff_t *f_pos);
```

`write` est appelé à chaque écriture du périphérique (voir l'appel système correspondant).

`filp`	Fichier concerné par l'écriture
`buff`, `len`	Buffer et quantité de données à écrire; **si on le souhaite on peu écrire moins de données que demandé**
`f_pos`	Position courante dans le fichier; **il faut mettre à jour cette position en fonction du nombre de bytes effectivement écrits**

Doit retourner la quantité de bytes effectivement écrits.

write

```
ssize_t (*write) (struct file *filp, const char __user *buff,
size_t len, loff_t *f_pos);
```

Comme indiqué par la macro `__user`, **le pointeur `buff` est une adresse de l'espace mémoire du processus; il faut donc convertir cette adresse dans l'espace d'adressage du noyau.** Pour cela on utilise:

```
#include <asm/uaccess.h>
unsigned long copy_from_user(void *to, const void __user *from,
unsigned long count);
```

`to`	Adresse noyau vers laquelle copier les données
`from`	Données à copier depuis l'espace du processus
`count`	Quantité de bytes à copier

Retourne la quantité de bytes qu'il reste à copier; si elle est différente de 0 alors un problème à été rencontré.

lseek

```
loff_t (*llseek) (struct file *filp, loff_t off, int whence);
```

`filp`	Fichier concerné par l'écriture
`off`	Offset demandé (pas l'offset actuel comme pour `read` et `write`)
`whence`	L'offset doit être interprété relativement à ce paramètre (voir l'appel système `lseek`)

Retourne le nouvel offset dans le fichier.

La fonction `llseek` doit modifier `filp->f_pos` comme demandé. La fonction doit aussi collaborer avec les fonctions `read` et `write`.

On peut mettre l'entrée `llseek` de la structure `f_ops` à NULL. Dans ce cas, le noyau effectue les changements lui même (pas toujours souhaité).

lseek

Si l'on souhaite ne pas implémenter d'accès aléatoire (parce que le périphérique ne supporte pas cette fonctionnalité) il faut utiliser la fonction suivante **lors de l'ouverture du fichier**:

```
int nonseekable_open(struct inode *inode, struct file *filp);
```

Il faut aussi faire pointer l'opération vers la fonction pré-définie `no_llseek`:

```
struct file_operations my_fops = {
    .owner = THIS_MODULE,
    ...
    .llseek = no_llseek,
    ...
};
```