# EPU - Informatique ROB4
# Informatique Système
## Gestion des E/S, accès aux fichiers

**Sovannara Hak**
hak@isir.upmc.fr

Université Pierre et Marie Curie
Institut des Systèmes Intelligents et de Robotique (CNRS UMR 7222)

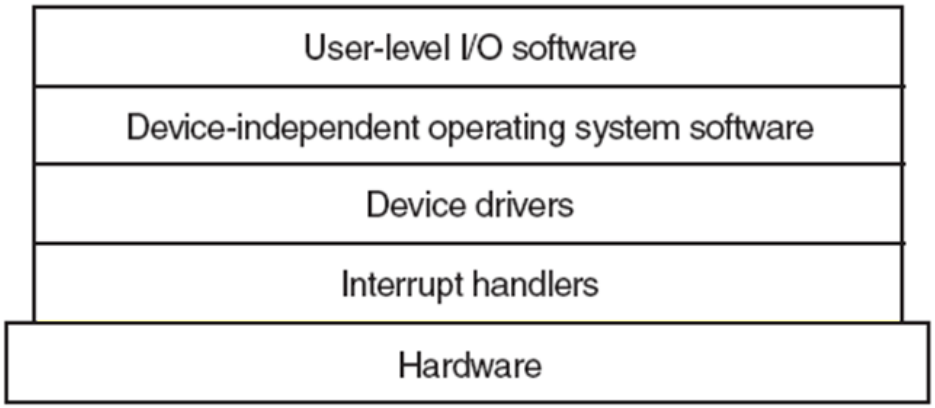
2014-2015


# Plan de ce cours

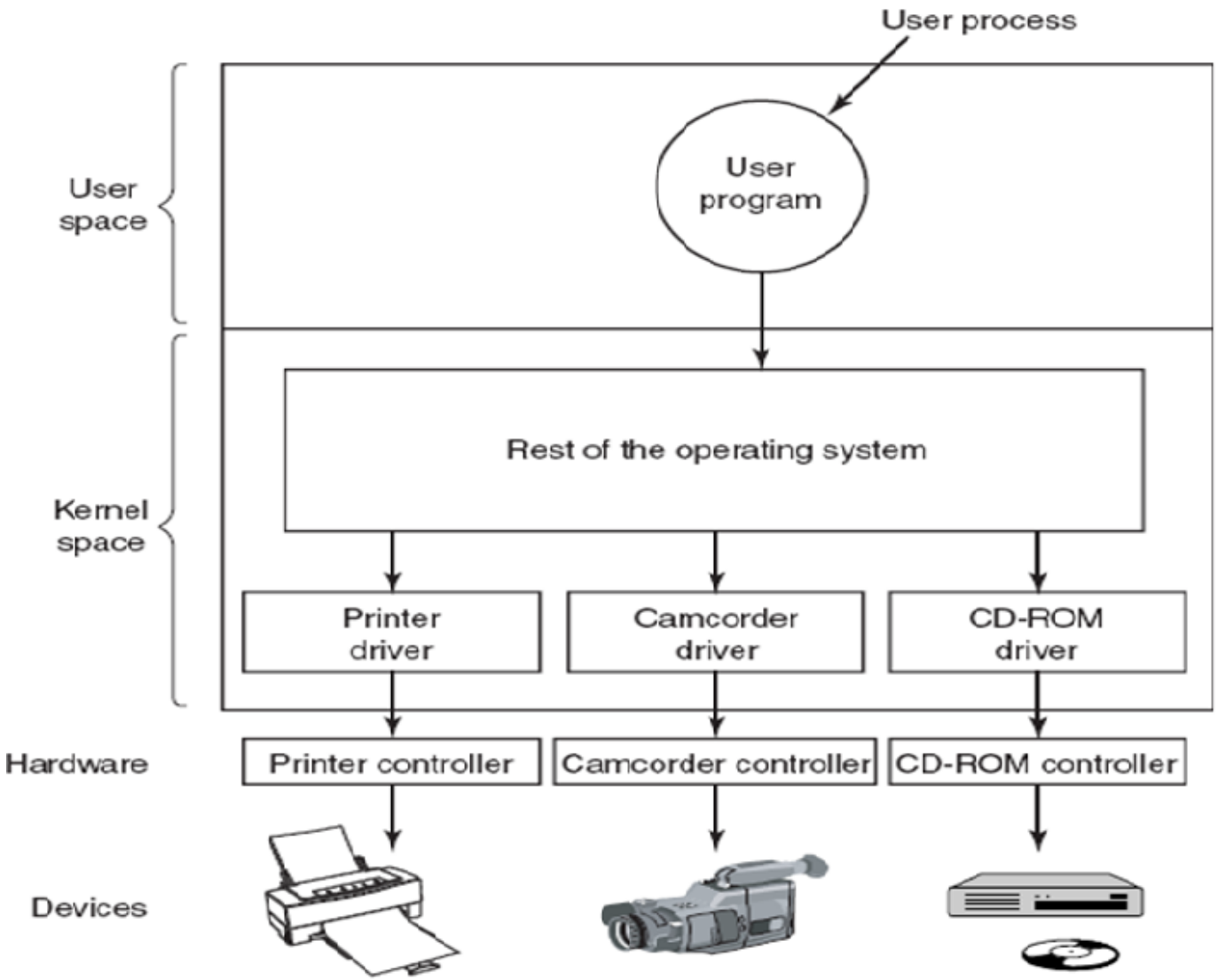
1. Gestion des E/S
   - E/S et OS
   - Quelques mots sur les protocoles d'entrées/sorties
   - Quelques mots sur le spooling
   - Gestion de fichiers
   - Accès direct vs séquentiel
   - Les fichiers spéciaux
2. Streams
   - Streams & stdio.h
   - Un mot sur la gestion des codes d'erreur
3. Accès aux fichiers
   - Accès direct
   - Accès séquentiel
   - Manipulation du mode d'ouverture d'un fichier avec open()
   - Rappel sur les opérateurs binaires en C
4. Exercices

# Rappel du plan

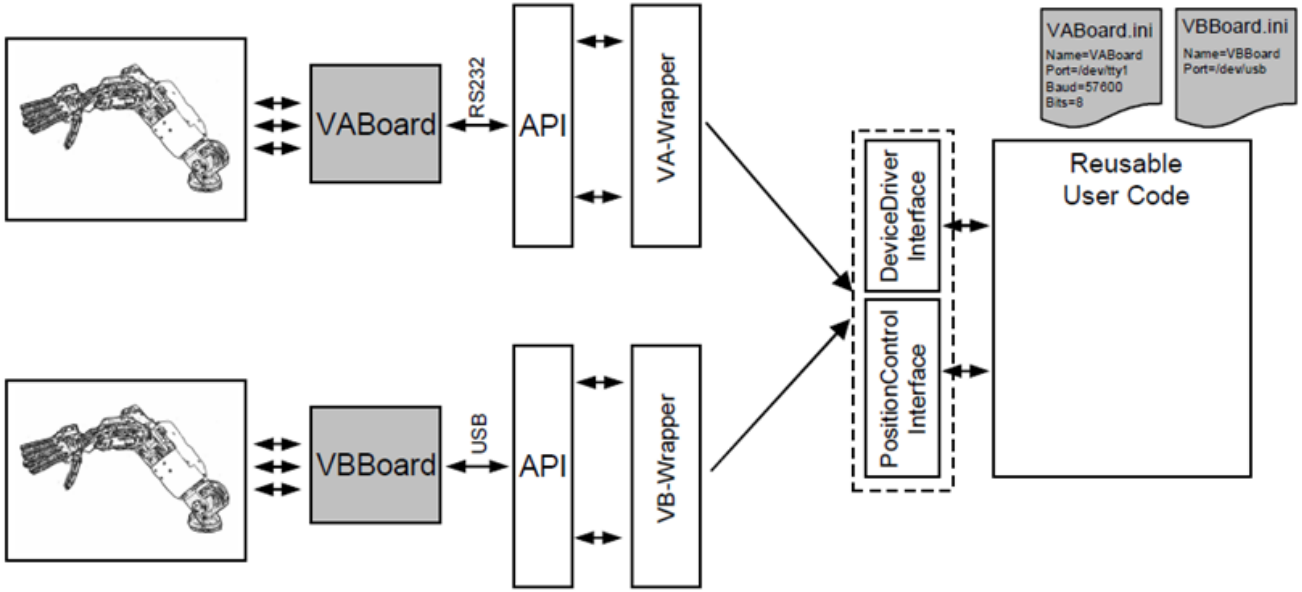
## ↦ Programme utilisant des E/S




1. L'accès aux périphériques dans un programme doit être indépendant des périphériques
   - Exemple : Lecture de fichier sur un cd, disque dur...
2. OS va cacher à l'utilisateur les contrôles bas niveau du périphérique en plusieurs niveau :
   - Gestion des interruptions (reveil du driver quand le périphérique est prêt...)
   - Driver (traduire les demandes R/W en commandes périphérique...)
   - Interface qui va rendre les périphériques uniformes (montage...)

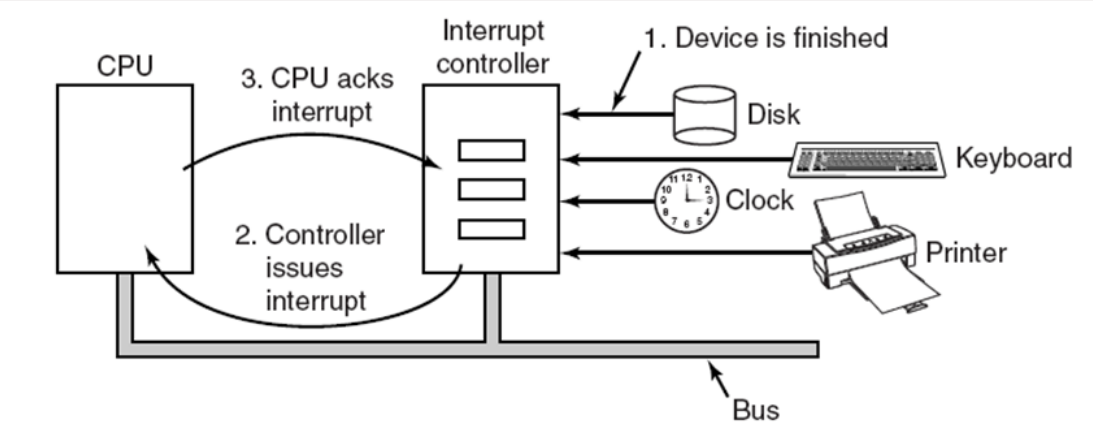

User process
User space
User program
Kernel space
Rest of the operating system
Printer driver
Camcorder driver
CD-ROM driver
Hardware
Printer controller
Camcorder controller
CD-ROM controller
Devices

## Analogue aux problèmes en robotique



## Exemple sous Linux où *tout* est fichier

- Gestion des interruptions : OS doit "cacher" ces opérations au programme de l'utilisateur.




- OS donne accès au matériel via des fichiers spéciaux dans /dev
  - /dev/hda1 : disque dur IDE *a* partiion 1
  - /dev/sdb3 : disque dur SCSI, SATA *b* partition 3
  - /dev/input/mouse0 : souris
  - /dev/mem : mémoire physique
  - ...
- Programme utilisateur manipule ces fichiers

# Rappel du plan

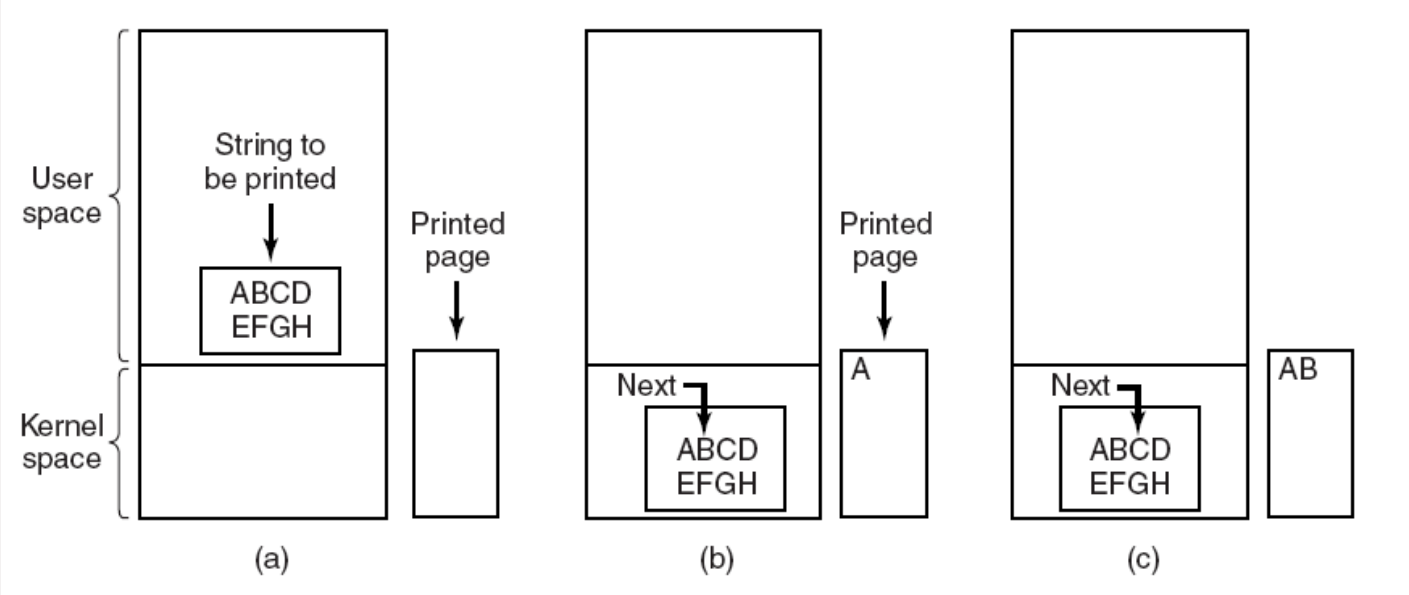
- On distingue deux grandes familles de protocoles d'E/S :
  - **Série** : transmission d'informations bit à bit sur un nombre limité de fils (RS232, USB, I2C, CAN ...).
  - **Parallèle** : transmission d'informations en // (mot par mot) sur un nombre dédié (PCI, SCSI, Bus processeur ...).
- Un protocole de communication est notamment caractérisé par son débit, son (a)synchronisme, sa directionnalité (simplex, half duplex, full duplex).
- On n'entre pas dans les (nombreux) détails dans le cadre de ce cours mais il faut tout de même savoir...
- ...que la transmission d'informations (notamment à des débits élevés) peut être la source d'erreur notamment liées à des perturbations électro-magnétiques.
- Il est important de pouvoir détecter ces erreurs (bit de parité, somme de contrôle (checksum)), voire de les corriger (code de Hamming).

## Principe des bits de parité

- On ajoute un bit dit de parité au message ;
- Convention de parité paire : le bit ajouté est tel que le message contient un nombre pair de 1 ;
- Convention de parité impaire : le bit ajouté est tel que le message contient un nombre impair de 1 ;
- Permet une détection simple des erreurs de transmission ;
- La somme de contrôle généralise ce principe.

## Gestion d'accès à un périphérique par des programmes

- Exemple : un process qui imprime une chaîne de caractères




- Que se passe-t-il si un processus bloque le périphérique ?
- Que se passe-t-il si d'autre processus ont besoin du périphérique ?

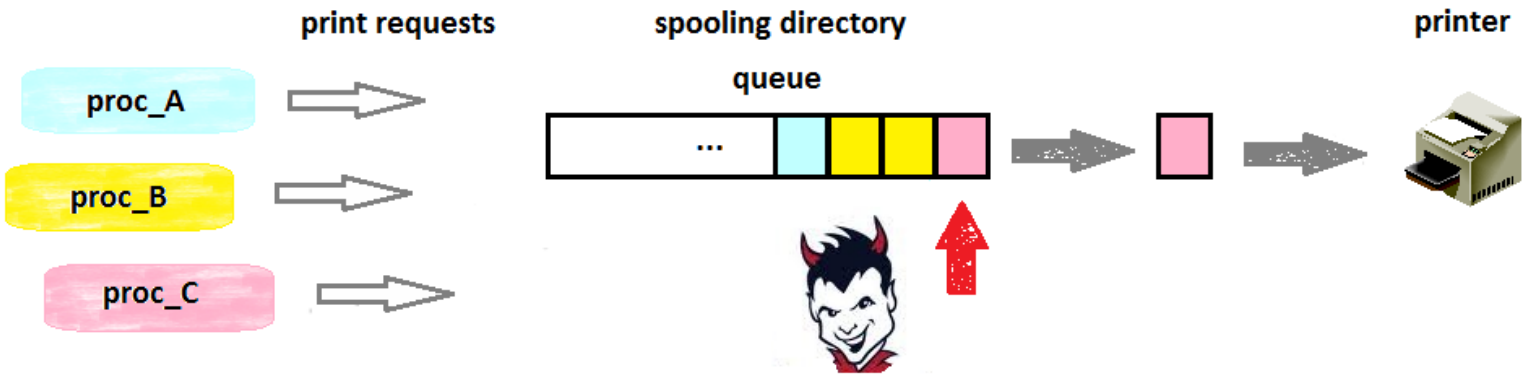
# Rappel du plan

- Les programmes n'accèdent pas directement aux périphériques
- Un processus intermédiaire se met entre les processus utilisateur et le périphérique
  - pour empêcher un accès direct par l'utilisateur
  - pour empêcher de monopoliser une ressource
  - famine
- Exemple : imprimante ne va pas être directement accessible aux processus utilisateur

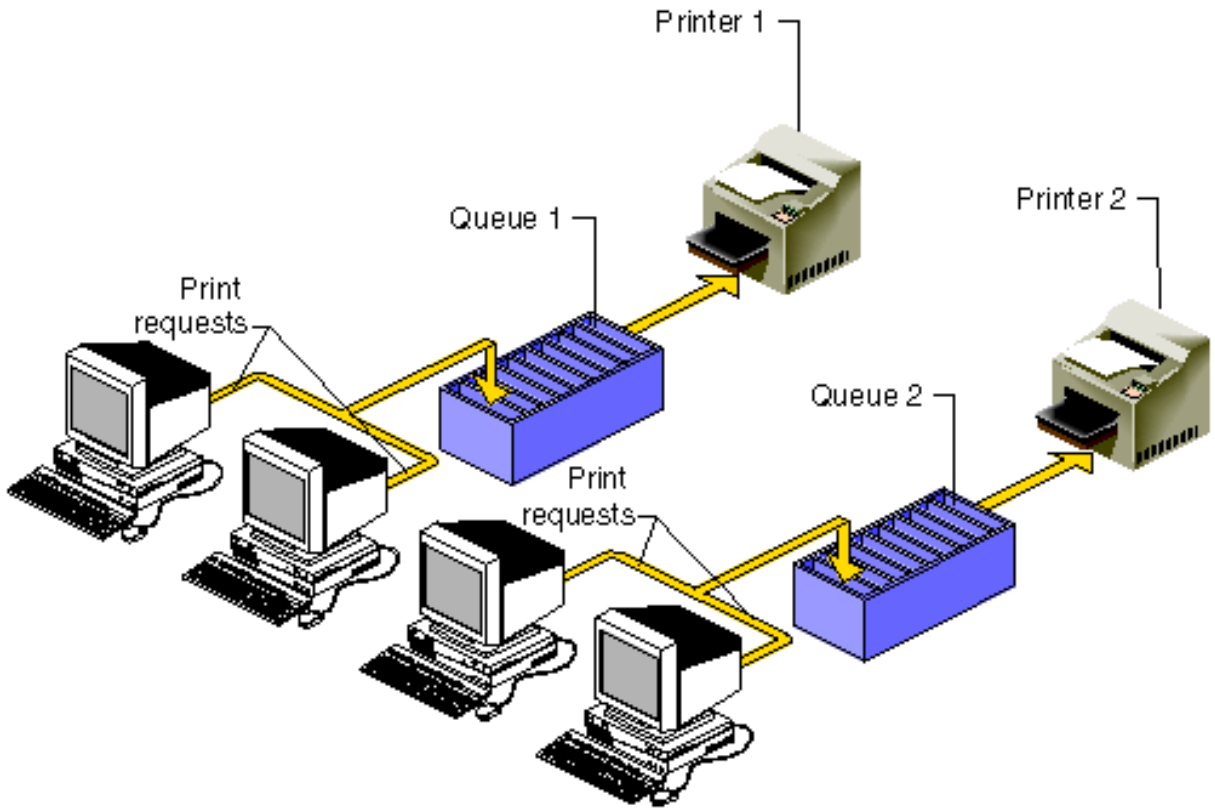
$\mapsto$ **spooling & demon**

## Spooling & demon

- Une file d'attente est créée (**le spool**)
- Les processus qui veulent utiliser l'imprimante mettent leurs fichier dans la file
- Un **processus démon** est le seul à pouvoir accéder au fichier de l'imprimante
- Le démon gère l'impression, en prenant les fichiers en attente et en les envoyant à l'imprimante

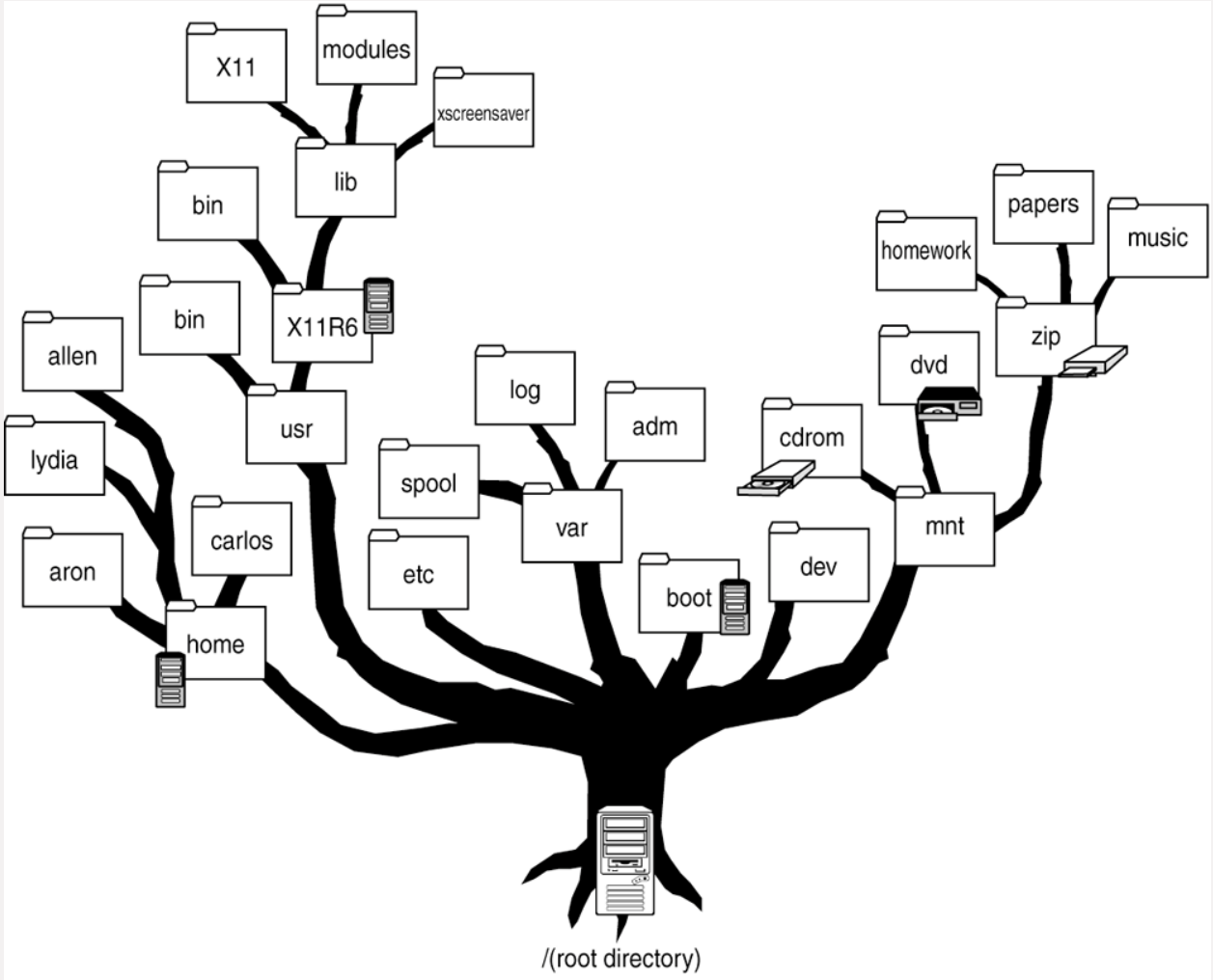


- Spooling est aussi utilisé pour les transferts de fichier par réseau (envoi d'e-mails, imprimante partagée, ..)



# Rappel du plan

## Sous Linux où *tout* est fichier



- L'OS possède un **système de gestion de fichiers**.
- **Rôle :** Assurer la conservation des données sur un support de masse non volatile (ex : disque dur).
- **Elément de base :** le fichier, unité de stockage indépendante des propriétés physiques des supports de conservation.

## Fichier Logique vs Fichier physique

**Fichier logique :** correspond à la vue qu'a l'utilisateur de la conservation de ses données.
**Fichier physique :** représente le fichier tel qu'il est alloué physiquement sur le support de masse.

- L'OS assure le lien et la correspondance entre ces deux niveaux de représentation en utilisant une structure de **répertoire**.

## Structure d'un fichier

Contient des informations de gestion de fichiers.
Pour chaque fichier :

- le nom logique du fichier (celui connu par l'utilisateur) et son type (éventuellement)
- l'adresse physique du fichier (adresse des blocs allouées au fichier sur le support de masse)
- la taille en octets ou en block du fichier
- la date de création du fichier, le nom du propriétaire
- les droits et protections (rwx, uog) du fichier
- la fonction `stat()` fournit retourne ces informations dans une structure de type **struct** stat pour un fichier donné.

- Manipulation des répertoires en C : `mkdir()`, `rmdir()`, `chdir()`, `getcwd()`, `opendir()`, `readdir()`, `closedir()` ...

## stat() - get file status

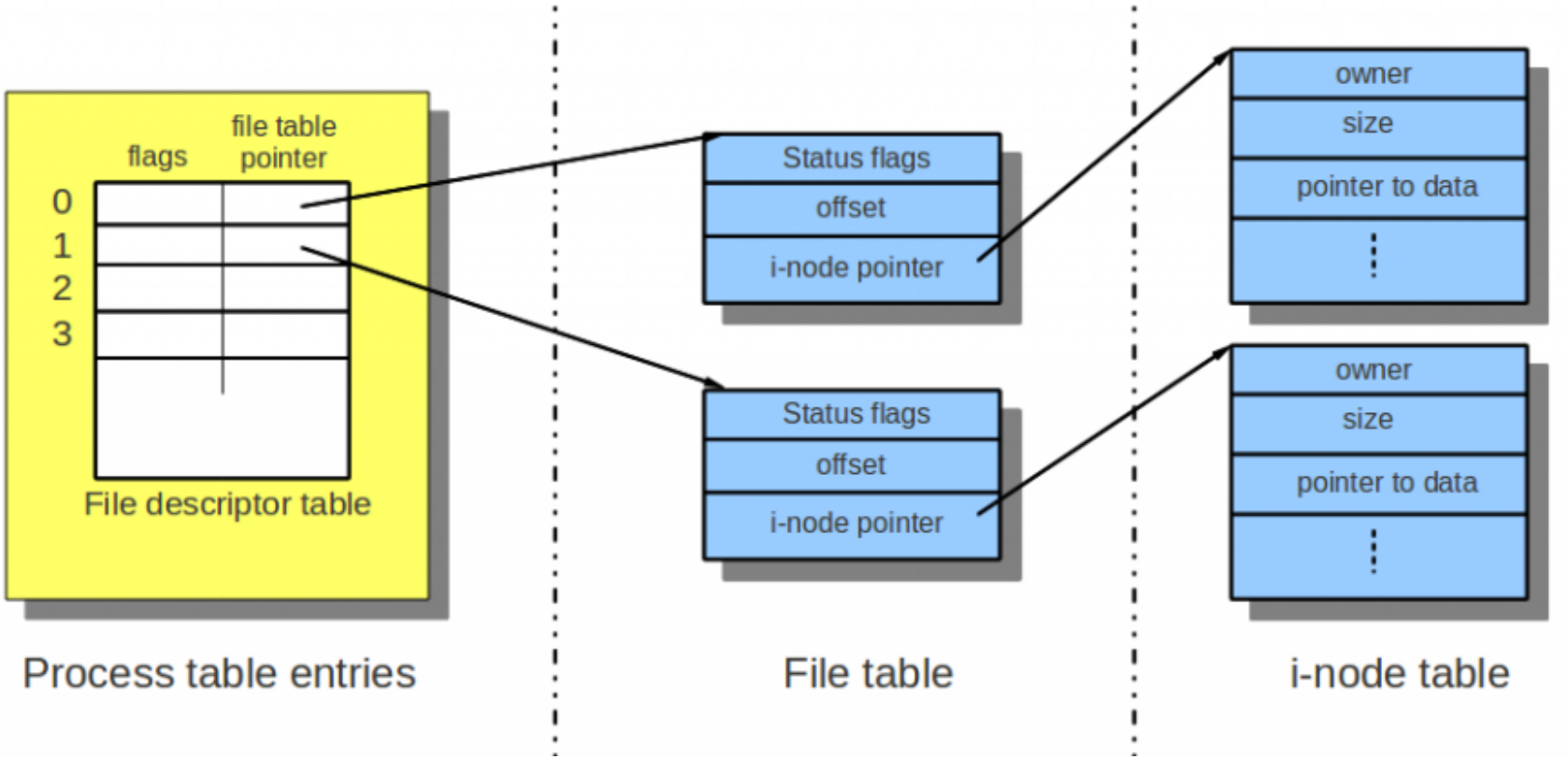
```
int stat(const char *path, struct stat *buf);
```

Include files :

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

Stat structure :

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    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 ID (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 blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```


flags
file table pointer
0
1
2
3
File descriptor table
Process table entries
Status flags
offset
i-node pointer
Status flags
offset
i-node pointer
File table
owner
size
pointer to data
owner
size
pointer to data
i-node table

- Un fichier logique est un **type de données standard** défini dans la plupart des langages de programmation.
- **Opérations associées :** création, ouverture, fermeture, destruction.
- Les opérations de création et d'ouverture effectuent un lien entre le fichier logique et le fichier physique correspondant.
- Les opérations de fermeture et destruction rompent ce lien.
- Un fichier logique correspond à un certain nombre d'enregistrements (données) dont la structure est propre au programme qui les manipule.
- Les enregistrements d'un fichier logique sont accessibles au travers d'opérations de lecture et d'écriture appelés fonctions d'accès.
- L'accès à un fichier peut être **direct ou séquentiel**.

# Rappel du plan

## Accès séquentiel

- les enregistrements sont traités dans l'ordre où ils se trouvent dans le fichier (octet par octet) ;
- une opération de lecture délivre l'enregistrement courant et se postionne sur le suivant ;
- une opération d'écriture place le nouvel enregistrement en fin de fichier ;
- Mode d'accès simple, pas forcément pratique, fichier accessible en lecture seule ou en écriture seule.

## Accès direct

- Accès bufferisé : le fichier est chargé en totalité en mémoire cache ;
- tous les enregistrements sont accessibles quelque soit leur position dans le fichier ;
- l'accès direct à un enregistrement se fait en spécifiant sa position relative par rapport au début du fichier ;
- Mode d'accès plus complexe, souvent plus pratique, fichier accessible en lecture seule, en écriture seule ou en lecture et en écriture simultanément.

# Rappel du plan

- L'OS gère les entrées/sorties au travers de **fichiers spéciaux**.

## Fichiers standards vs fichiers spéciaux

**Fichiers standards :** l'ensemble des fichiers directement manipulés et structurés par l'utilisateur.
**Fichiers spéciaux :** fichiers associés aux périphériques, possèdent une structure interne liée au système et doivent être accédés de manière spéciale.

- Un même appel système de type `write()` peut être utilisé pour écrire dans un fichier standard ou pour lancer une impression en écrivant dans le fichier spécial `/dev/lp0` attaché au pilote de l'imprimante.
- Les fichiers spéciaux sont de deux types : **bloc** ou **caractère**.

## Fichiers de type bloc

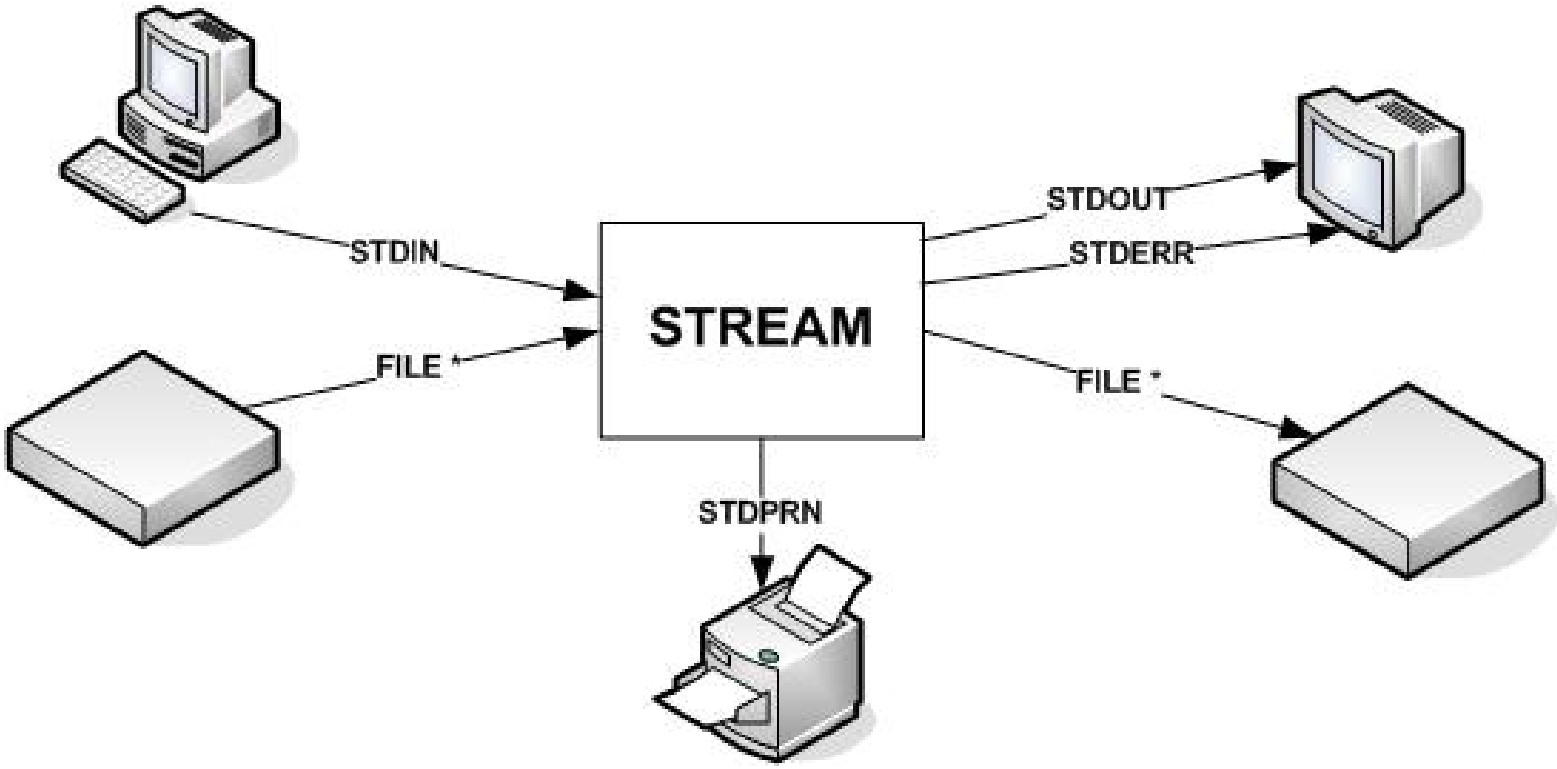
- Correspondent aux périphériques structurés en blocs : disque dur, CDROM.
- Permettent un accès direct.

## Fichiers de type caractère

- Correspondent aux périphériques sans structure : terminaux (clavier, écran), imprimante/scanner, carte son, souris, joystick, tubes (cf. semaine prochaine)...
- Permettent un accès octet par octet (séquentiel).

- Fonctions associées aux fichiers spéciaux et aux périphériques : `mknod()` (cf. cours sur les tubes), `ioctl()`, `select()` (pas traitées dans ce cours).

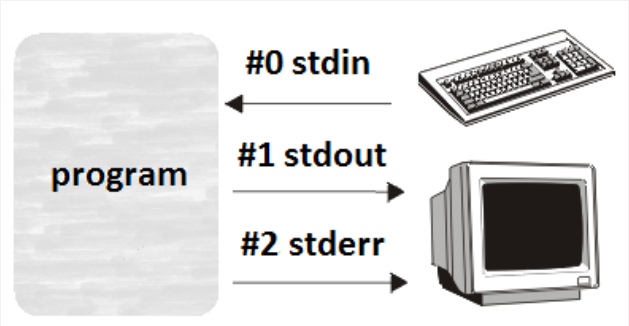
# Rappel du plan


STDIN
FILE *
STREAM
STDOUT
STDERR
FILE *
STDPRN

## Streams

- **stream** = un fichier ou un périphérique physique
- moyen flexible et portable de lire/écrire des données
- streams sont toujours manipulé à travers des *pointeurs*, comme des fichiers

## Streams prédéfini sous UNIX




- `stdin` : Clavier par défaut, descripteur de fichier 0
- `stdout` : Console par défaut, descripteur de fichier 1
- `stderr` : Console par défaut, descripteur de fichier 2

E/S formattées

- `int printf(char *format, ...)` : imprime vers `stdout`
- `int scanf(char *format, ...)` : lit de `stdin`
- `int fprintf(FILE *stream, char *format, args..)` : imprime vers `stream` (e.g. `stderr`)

## Redirection en bash

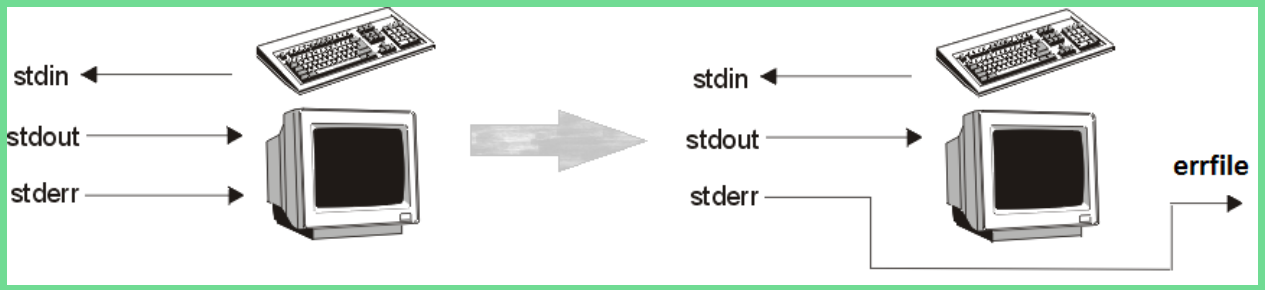
- N > Cible : redirection du file descriptor N (1 par défaut) vers Cible (écrasement de Cible si déjà existant)
  ```
  mon_programme > mon_fichier
  ```
- N >> Cible : redirection du file descriptor N (1 par défaut) à la fin de Cible (concaténation)
- &N : Référence vers le file descriptor N (pour désambiguiser avec les fichiers N)
- N < Source : Le file descriptor N utilise Source comme source de donnée
- commande << TAG : lecture de stdin jusqu'à TAG
- Redirection de `stdout` d'un programme vers le `stdin` d'un autre : | (**pipe**)
  ```
  program1 | program2
  ```

**Exemple :**

Using `fprintf` on preopened `stdprn` stream.

```
#include <stdio.h>
void main( )
{
        printf("This␣goes␣to␣stdout\n");
        fprintf(stderr,"This␣goes␣to␣stderr\n");
}
```

## TEST




```c
#include <stdio.h>
void main( )
{
        printf("This goes to stdout\n");
        fprintf(stderr,"This goes to stderr\n");
}
```

- Comment rediriger `stdout` vers `errfile` ?
- Comment rediriger `stderr` vers `errfile` ?
- Comment rediriger `stderr` vers `stdout` lors de l'exécution de `myProgram` ?

# Rappel du plan

- Lors de l'appel à une fonction standard du C, un bilan de l'exécution de la fonction est passé à la variable entière "spéciale" errno
- A chaque valeur possible de errno correspond un type d'erreur représenté par une constante symbolique et dont la liste peut être trouvé dans errno.h .
- La valeur numérique de errno n'est pas très informative en tant que telle mais des fonctions comme strerror() ou perror() permettent d'obtenir un message lié au type d'erreur, lisible par un être humain.

## Utilisation de perror()

```
$ gcc -o test test_error.c
$ ./test
Ouverture: No such file or directory
```

```c
#include <stdio.h>

int main(){
    FILE *pF;
    pF = fopen("./test.txt","r");
    if(!pF)
        perror("Ouverture");
    return 0;
}
```

# Rappel du plan

# Manipulation de fichiers en mode direct (bufferisé) (`stdio.h`)

## Ouverture

- FILE *fopen(**const char** *path, **const char** *mode);
  - fopen() retourne un pointeur sur le flux associé au fichier ouvert.
  - Une variable de type FILE contient l'ensemble des informations nécessaires à la gestion des accès en lecture/écriture à un fichier. Parmi ces informations :
    - l'adresse du fichier physique associé ;
    - la position courante en lecture ou en écriture dans le fichier
    - l'adresse du tampon (buffer) associé au fichier (les accès à un fichier sont groupés pour être moins fréquents : lorsqu'un processus demande à écrire dans un fichier, les données à écrire sont mises en mémoire centrale dans un tampon jusqu'à ce que le tampon soit plein ou bien que le fichier soit fermé à la demande de l'utilisateur ; les données sont alors recopiées dans le fichier. De même pour des accès en lecture.)
  - path est un pointeur vers la chaine contenant le chemin du fichier dans l'arborescence ;
  - mode est un pointeur sur une chaine de caractère indiquant le mode d'ouverture du fichier : lecture, écriture, ajout ...
  - si le fichier est créé, ses droits par défaut son du type 666 à moins qu'un appel à umask() n'est modifié le masque de création de fichier.

## Réassociation

- FILE *freopen(**const char** *path, **const char** *mode, FILE *stream);
  - (ré)ouvre un fichier et l'associe au flux stream - le flux original s'il existe est fermé

## Fermeture

- **int** fclose (FILE *fp);
  - permet de fermer le fichier associé au flux pointé par fp

## Lecture et écriture en mode texte

fprintf (), fputc (), fputs () / fscanf (), fgetc (), fgets ()

## Lecture

- size_t fread(**void** *ptr, size_t size, size_t nmemb, FILE *stream);
  - lit nmemb éléments de données, chacun d'eux représentant size octets de long, depuis le flux pointé par stream , et les stocke à l'emplacement pointé par ptr .
  - renvoie le nombre d'éléments correctement lus (et non pas le nombre d'octets). Si une erreur se produit, ou si la fin du fichier est atteinte en lecture, le nombre renvoyé est plus petit que nmemb et peut même être nul.

## Écriture

- size_t fwrite(**const void** *ptr, size_t size, size_t nmemb,FILE *stream);
  - écrit nmemb éléments de données, chacun d'eux représentant size octet de long, dans le flux pointé par stream , après les avoir récupérés depuis l'emplacement pointé par ptr .
  - renvoie le nombre d'éléments correctement écrits (et non pas le nombre d'octets). Si une erreur se produit le nombre renvoyé est plus petit que nmemb et peut même être nul.

## Fonctions liées

- fseek () : déplace l'indicateur de position du flux à l'endroit indiqué (en octets).
- rewind() : déplace l'indicateur de position du flux au début du fichier.
- fflush () : force l'écriture des données du tampon.

# Rappel du plan

## Manipulation de fichiers en mode séquentiel ( sys/types.h, sys/stat.h, fcntl.h, unistd.h )

### Ouverture

- **int** open(**const char** *pathname, **int** flags , mode_t mode);
  - open() retourne un descripteur de fichier. Ce descripteur est associé à une entrée dans la table des fichiers ouverts du système ;
  - pathname est un pointeur sur une chaine de caractère contenant le chemin du fichier dans l'arborescence ;
  - flags est une combinaison d'options permettant de spécifier le mode (écriture, lecture, ajout) d'ouverture du fichier ;
  - mode spécifie les droits associés au fichier si celui est créé par open() .

### Fermeture

- **int** close ( **int** fd );
  - qui permet de fermer le fichier associé au descripteur fd

## Lecture

- ssize_t read(int fd, void *buf, size_t count);
  - lit jusqu'à count octets depuis le descripteur de fichier fd dans le tampon pointé par buf .
  - renvoie -1 s'il échoue, auquel cas errno contient le code d'erreur, et la position de la tête de lecture est indéfinie. Sinon, renvoie le nombre d'octets lus (0 en fin de fichier), et avance la tête de lecture de ce nombre. Le fait que le nombre renvoyé soit plus petit que le nombre demandé n'est pas une erreur. Ceci se produit à la fin du fichier notamment.

## Écriture

- ssize_t write(int fd, const void *buf, size_t count);
  - lit au maximum count octets dans la zone mémoire pointée par buf , et les écrit dans le fichier référencé par le descripteur fd .
  - renvoie le nombre d'octets écrits (0 signifiant aucune écriture), ou -1 s'il échoue, auquel cas errno contient le code d'erreur. Le nombre d'octets écrits peut être inférieur à count par exemple si la place disponible sur le périphérique est insuffisante.

## Fonction liée

- off_t lseek(int fd, off_t offset , int whence);
  - déplace la tête de lecture/écriture à la position offset (en octets) dans le fichier associé au descripteur fd

# Rappel du plan

- Le mode d'ouverture du fichier est spécifié par les constantes symboliques O_RDONLY, O_WRONLY ou O_RDWR.
- De plus, zéro ou plus d'attributs de création de fichier et d'attributs d'état de fichier peuvent être spécifiés dans flags avec un OU binaire : O_CREAT, O_APPEND....
- Si O_CREAT est spécifié, les droits sont spécifiés par des constantes symboliques :
  - ...
  - S_IRWXU (00700) L'utilisateur (propriétaire du fichier) a les autorisations de lecture, écriture, exécution.
  - S_IRUSR (00400) L'utilisateur a l'autorisation de lecture.
  - ...
  - S_IWGRP (00020) Le groupe a l'autorisation d'écriture.
  - ...
  - S_IXOTH (00001) Tout le monde a l'autorisation d'exécution.

**Ouverture d'un fichier en lecture/écriture avec création le cas échéant et droits 750**

```
int fd = open("/tmp/test.data",O_RDWR | O_CREAT,S_IRWXU | S_IRGRP | S_IXGRP);
```

# Rappel du plan

- Les opérateurs binaires en C permettent la manipulation bit à bit de variables.
- Ils sont aussi très utiles pour la manipulation des constantes symboliques utilisées pour spécifier des options lors de l'appel de fonctions (cf. `open()` ). On parle alors de **masque binaire**.
- L'opérateur OU (noté `|` ) : permet notamment la mise à 1 d'un bit dans un mot binaire.

**Mise à un 1 du 4ème bit d'un octet**

$b01001110 \,|\, b00010000 = b01011110$

- L'opérateur NON (noté `!` ) : permet d'obtenir la négation d'un mot binaire.

**Négation du mot $0xFF002275$**

$!(0xFF002275) = 0x00FFDD8A$

- L'opérateur ET (noté `&` ) : permet notamment la mise à 0 d'un bit dans un mot binaire.

**Mise à un 1 du 4ème bit d'un octet**

$b01011110 \,\&\, !(b00010000) = b01001110$

- Les opérateurs de décalage (notés `>>` et `<<` ) : permettent le décalage à droite ou à gauche des bits d'un mot binaire.

Operateurs binaires

| A | B | A&B | A \| B |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Example 1 : Quel est le résultat de $73 >> 3$ ?

Example 2 : Trouver la valeur du $i$-e bit d'une certaine séquence, par exemple the 5e bit de $x = 11011001$.

## Exercice 1

Écrire un programme qui ouvre un fichier de texte, lit le contenu, et écrit dans un autre fichier le même texte.

Utiliser : `fopen, fclose, fread, fwrite`

Code :

## Exercice 2

Écrire un programme qui lit des lignes sur stdin et les recopie sur stdout. Le programme doit supporter les usages suivants :

```
$ ./fileTransfer
Bonjour <-- écrit par moi
Bonjour <-- écrit par le programme
..

$ ./fileTransfer < input.txt
- Bonjour je m'appelle Bob.
- Bonjour Bob.
Fin du fichier

$ ./fileTransfer < input.txt > output.txt
Fin du fichier
```

Utiliser : `fgets`, `fputs`, `ferror`, `feof`, `clearerr`

```
char *fgets(char *s, int size, FILE *stream);
 return s on success, and NULL on error or when end of file occurs
 while no characters have been read.

int fputs(const char *s, FILE *stream);
 return a nonnegative number on success, or EOF on error.

int ferror(FILE *stream);
 tests the error indicator for the stream pointed to by stream,
 returning nonzero if it is set. The error indicator can only
 be reset by the clearerr() function.

int feof(FILE *stream);
 tests the end-of-file indicator for the stream pointed to by stream,
 returning nonzero if it is set.  The end-of-file indicator
 can only be cleared by the function clearerr().

void clearerr(FILE *stream);
```

Code :

## Exercice 3

Écrire un programme qui ouvre un fichier de texte, lit le contenu ligne par ligne, et écrit dans un autre fichier le même texte avec les lignes dans l'ordre inverse.

# Questions ?