EPU - Informatique ROB4
Informatique Système
Pointeurs de fonction + E/S

Miranda Coninx
Presented by
Ludovic Saint-Bauzel
ludovic.saint-bauzel@sorbonne-universite.fr

Sorbonne Université
Institut des Systèmes Intelligents et de Robotique (CNRS UMR 7222)

2023-2024

SORBONNE UNIVERSITÉ

ISIR
INSTITUT DES SYSTÈMES INTELLIGENTS ET DE ROBOTIQUE

Plan de ce cours

Définition

- Un programme en cours d'exécution (processus) occupe de l'espace dans la mémoire centrale.
- Le code de l'exécutable et les variables utilisées sont placés dans la mémoire et constituent une partie de ce que nous avons appelé le processus.
- Ainsi chaque instruction ou fonction du code exécutable est stockée en mémoire et possède donc une adresse.
- Un pointeur de fonction représentent, au même titre qu'un pointeur sur variable, une adresse en mémoire : celle d'une fonction.
- La désignation d'une fonction par son adresse (i.e. par un pointeur) apporte un élément de flexibilité supplémentaire dans la conception du code d'un programme.

Déclaration

- Un pointeur de fonction est une variable et est déclaré en tant que telle.
- La syntaxe de déclaration se rapproche de celle d'un prototype de fonction, le nom de la fonction étant placé entre parenthèse et précédé d'une `*`.
- Déclaration et initialisation d'un pointeur pour une fonction retournant un entier et prenant comme arguments deux entiers :

```
int (*pf_get_int)(int ,int ) = NULL;
```

Un example simple : **** ex0_pf.c ****

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

void print1(int val){ fprintf(stdout,"Num = %d\n",val);}
void print2(int val){ fprintf(stdout,"N := %d\n",val);}

int main()
{
        void (*pf) (int) = NULL;
        int val = 5;

        fprintf(stdout,"pf points to : %p\n",pf);

        pf = &print1;
        fprintf(stdout,"pf points to : %p\n",pf);
        (*pf) (val);

        pf = &print2;
        fprintf(stdout,"pf points to : %p\n",pf);
        (*pf) (val);

        return 0;
}
```

Output :

```
$ ./ex0
pf points to : (nil)
pf points to : 0x400564
Num = 5
pf points to : 0x400590
N := 5
```

Affectation, Appel

- Affectation de l'adresse d'une fonction à un pointeur (l'opérateur d'adressage `&` est optionnel) :

```
/* la fonction doit exister et avoir un prototype identique à celui du pointeur */
pf_get_int = &saisie_entier_borne;
```

- Appel d'une fonction via un pointeur (l'opérateur de déréférencement `*` est optionnel) :

```
int a = (*pf_get_int) (0,100);
```

**** ex1_pf.c ****

Pointeur de fonction comment argument d'une fonction

Un pointeur de fonction peut être passé comme argument d'une fonction `f()` dans deux contextes différents :

- en tant qu'adresse "banalisée" dans la mémoire, le type de l'argument sera alors `void *`.
- en tant qu'adresse d'une fonction à utiliser dans la fonction `f()`. Il est alors nécessaire de passer à la fonction `f()` les arguments de la fonction à appeler.

```
/* Prototype de la fonction f() */
int saisie_entier_gen(int min, int max, int (*pf) (int, int));
...
/* Appel */
int a = saisie_entier_gen(0, 100, pf_get_int);
...
/* Appel, méthode alternative */
int a = saisie_entier_gen(0, 100, &saisie_entier_borne);
```

Bâsoupe aucó avec ça

*pf_get_int =
&saisie_..._borne

**** ex2_pf.c ****

Αυτό είναι λίγο περίεργο

Pointeur de fonction comme valeur de retour d'une fonction

Un pointeur de fonction peut être passé comme valeur de retour d'une fonction `f()` dans deux contextes différents :

- en tant qu'adresse "banalisée" dans la mémoire, le type de retour de la fonction sera alors `void *`.
- en tant qu'adresse d'une fonction et la syntaxe devient alors un peu obscure.

```
/* Prototype de la fonction */
int (* choix_saisie_entier_gen(char c)) (int,int) ;
...
/* Retour */
return &saisie_entier_borne_abs;
...
/* Appel */
pf_get_int = choix_saisie_entier('a');
```

Εδώ είναι σαν να γράφεις
int (*f)(int,int) choix_entier_
saisie_...gen(char c)

**** ex3_pf.c ****

Lisibilité : définition de types spécifiques aux pointeurs de fonction

Pour rendre le code plus lisible on peut définir un type spécifique avec `typedef`.

```
typedef int (*pf_saisie) (int, int);
...
/* Prototype de la fonction */
pf_saisie choix_saisie_entier_gen(char c);
...
/* Appel */
pf_get_int = choix_saisie_entier('a');
```

Είναι η ίδια σύνταξη με το MAP φορ

**** ex4_pf.c ****

Notion de généricité

- Une fonction est dite générique si son implémentation est indépendante du type et, éventuellement, du nombre de ses arguments.
- Par exemple, l'algorithmique du tri est indépendante du type de choses à trier et une fonction générique de tri doit pouvoir prendre comme argument un tableau à trier indépendamment du type de données qu'il contient. La seule fonction qui soit alors dépendante du type est la fonction de comparaison.
- Dans le même ordre d'idée, les structures de type *arbre*, *pile*, *table de hachage ou graphes* et les algorithmes associés (*parcours*, *recherche*, *insertion*,...) existent indépendamment du type qu'elles contiennent.

Pointeur de fonctions et généricité

- La fonction `int saisie_entier_gen(int min, int max, int (*pf) (int, int))` peut permettre une certaine souplesse de programmation : la fonction de saisie à effectivement utiliser est une variable et peut donc être défini dynamiquement (par l'utilisateur du programme, via un fichier, en fonction du contexte,...).
- Cependant cette fonction n'est pas générique :
  - elle ne permet pas de saisir autres choses que des entiers ;
  - elle prend forcément deux arguments d'entrées de type entier ;
  - elle ne peut retourner qu'un entier.
- Afin de la rendre générique, son prototype doit être :

```
void* saisie_gen(void* (*f) (void* arg), void* arg);
```

- Cela signifie :
  - que `saisie_gen()` retourne forcément un pointeur (ou, en trichant un peu, un entier puisqu'un pointeur a une valeur entiere) ;
  - que la fonction `f()` doit elle même retourner un pointeur (idem) ;
  - que les arguments à passer à la fonction `f()` doivent être passés sous la forme d'un seul pointeur (sur une structure si les arguments sont de types hétérogènes) ;
  - que la fonction `f()` devra commencer par une récupération des arguments via un cast de `void*` vers autre chose et se terminer par un cast pour retourner qqch de type `void *`

Remarques

Les pointeurs de fonction apporte une grande souplesse au code...mais sont aussi une grande source d'erreurs de syntaxe ou de bugs pas évidents à trouver.

`**** ex5_pf.c ****`

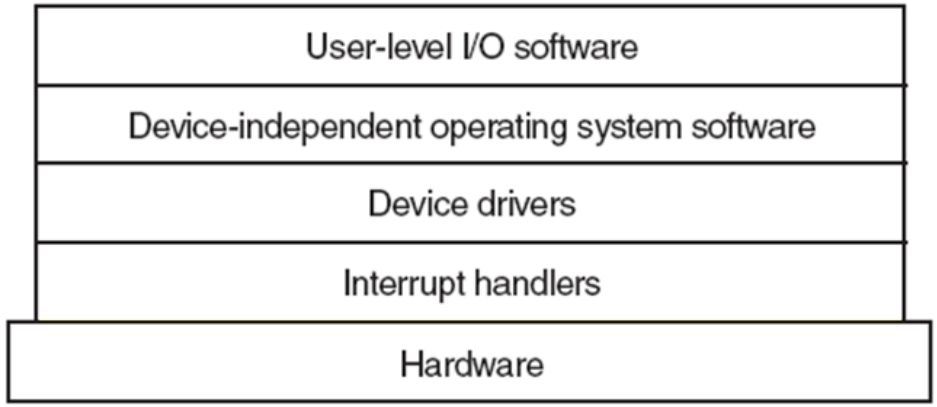
Pointeur de fonctions et threads

- Un intérêt majeur des pointeurs de fonction : les threads (ou *processus légers*, *fils d'exécution*)
  - Executer plusieurs fonctions de façon concurrente au sein du même processus
  - Les threads partagent le même espace mémoire (même processus)
  - Mais chaque thread a sa pile, son contexte, ses variables locales, etc.
- Un thread est crée avec un pointeur de fonction !

```
void* my_function(void* args) { ... }
...
pthread_t thread_id;
pthread_create(&thread_id, NULL, my_function, (void
*)function_args);
```

- **Suite aux cours 7 et 8 !**

$\mapsto$ from a physical device to a user program (and back)



1. The I/O software must provide to programmers a **device-independent** access **interface**
   - the OS must handle the different device drivers
   - the user code should be unaware of the details of the device for accessing it
2. **Uniform naming**
   - various devices of different type share the same I/O facilities and naming space (e.g. `/mnt/cdrom/`, `/mnt/sdcard1/`, etc.)
3. The OS must hide to the user the low-level device control
   - handling interrupts, a/synchronous commands
   - **handling I/O errors**
     - controller should be the first to fix errors at low level, then device driver

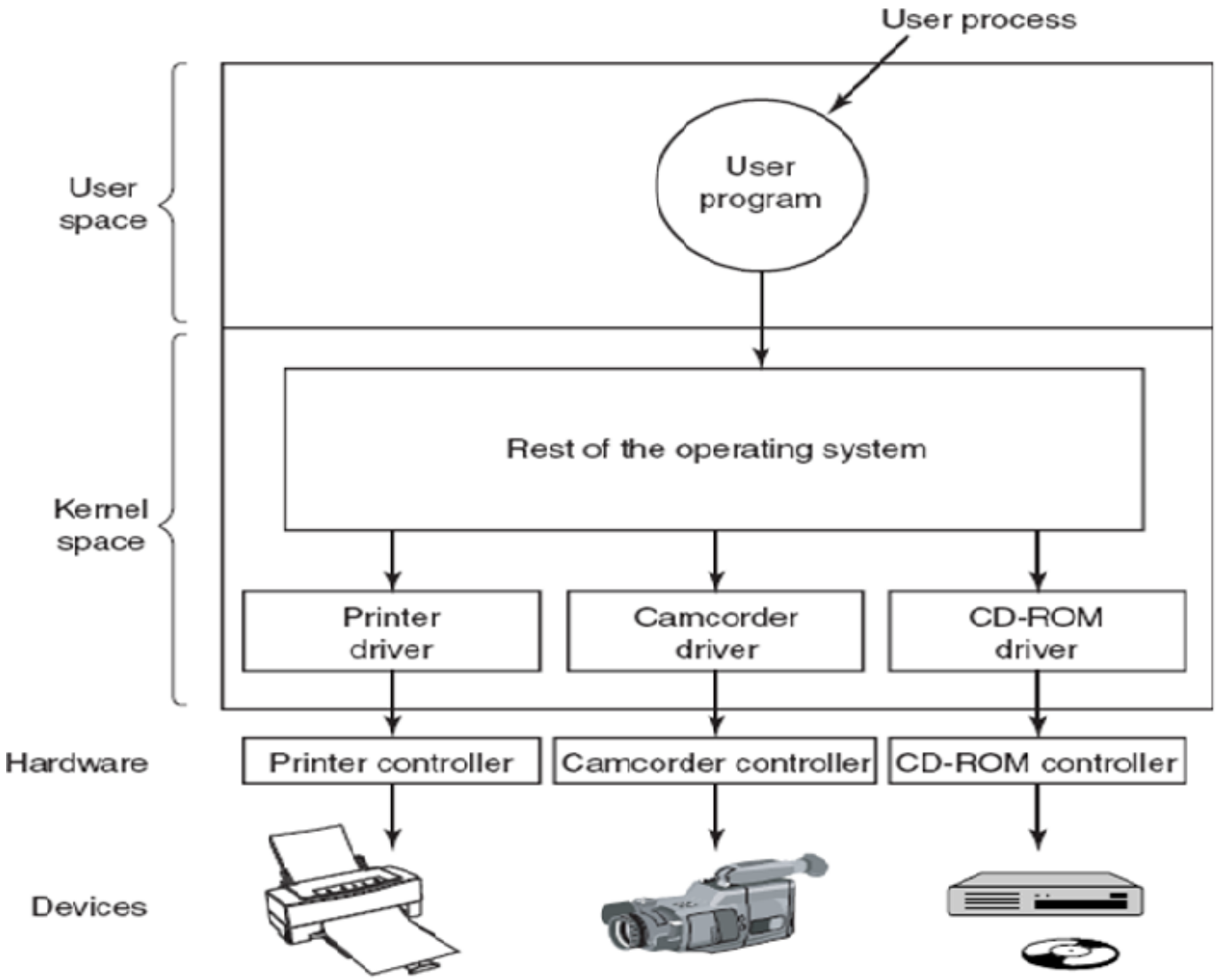
typical I/O devices (PC)

Two (basic) types of I/O devices :

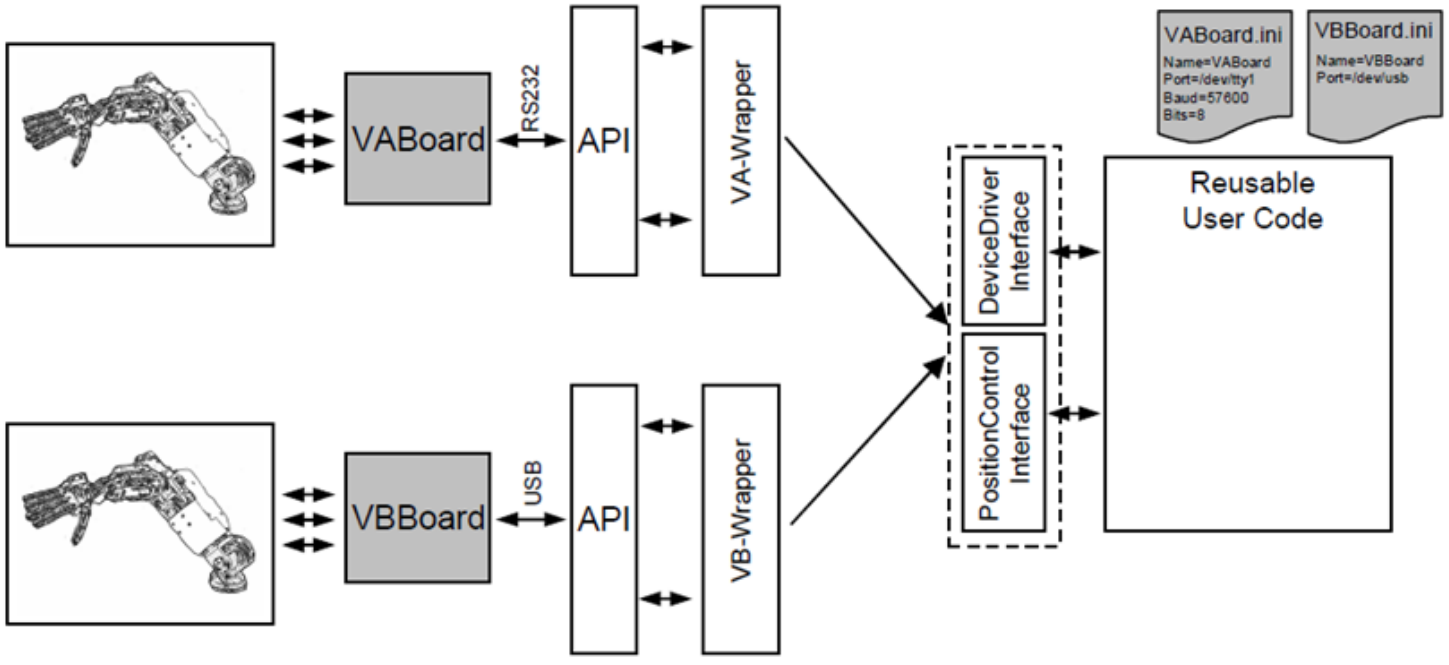
- **Block**
  - Blocks can be read independently of one another
  - e.g. hard-disks, CD-ROM, ..
- **Character**
  - A stream of characters - no block structure
  - e.g. printer, mouse, network interface

Device	**Data rate**
Keyboard	10 bytes/s
Mouse	100 bytes/s
56K modem	7 KB/s
ADSL 2+	3 MB/s
Fast Ethernet	12.5 MB/s
802.11n wifi	18.75 MB/s
USB 2.0	60 MB/s
FireWire 1 (1394a)	100 MB/s
Gigabit Ethernet	125 MB/s
32bit PCI 2.2	133 MB/s
PCI Express x1	250 MB/s
USB 3.0	625 MB/s
PCI Express x16	8 GB/s
Memory bus (recent)	20 GB/s

1) device-independent interface

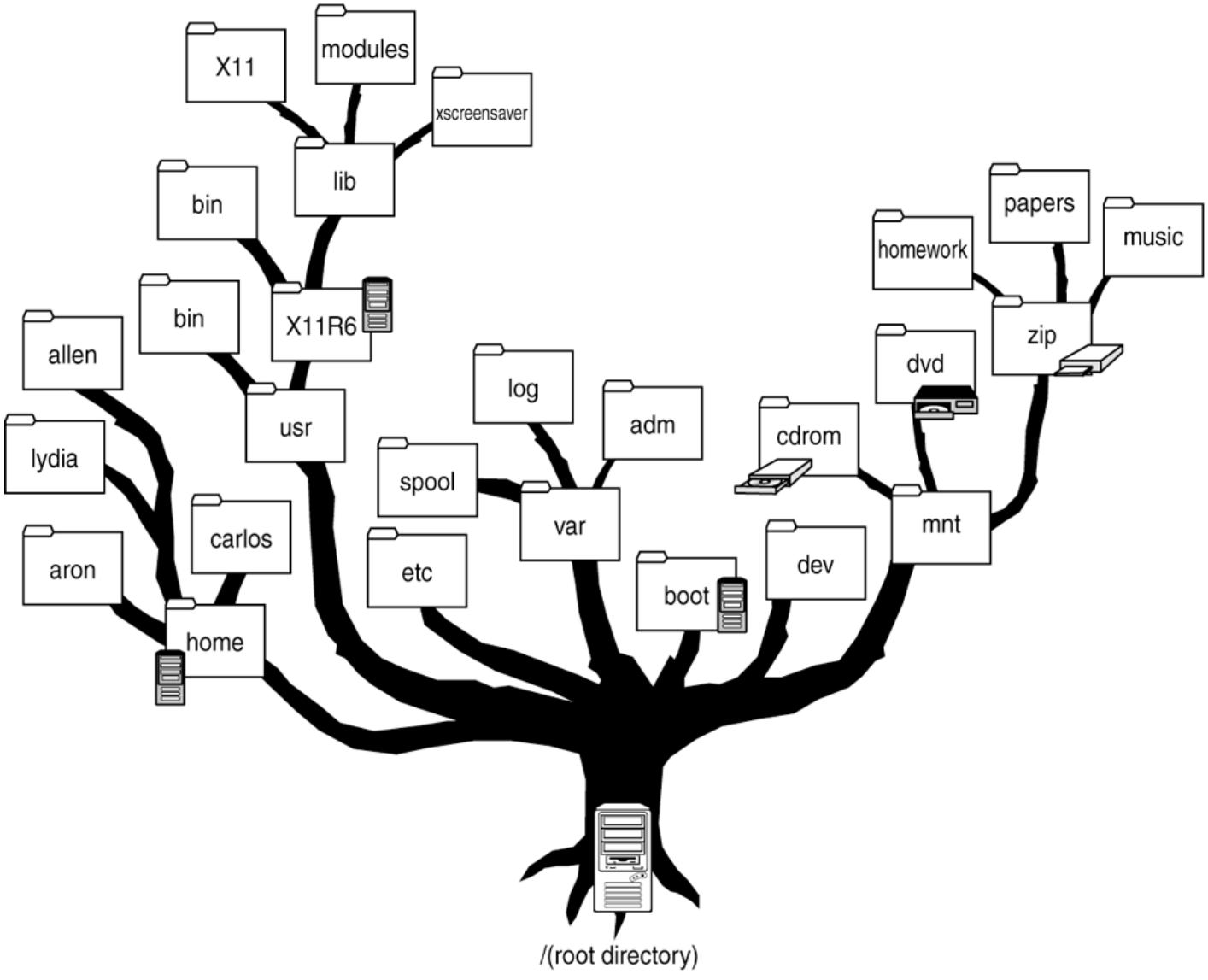


1) device-independent interface $\mapsto$ similar issues in robotics

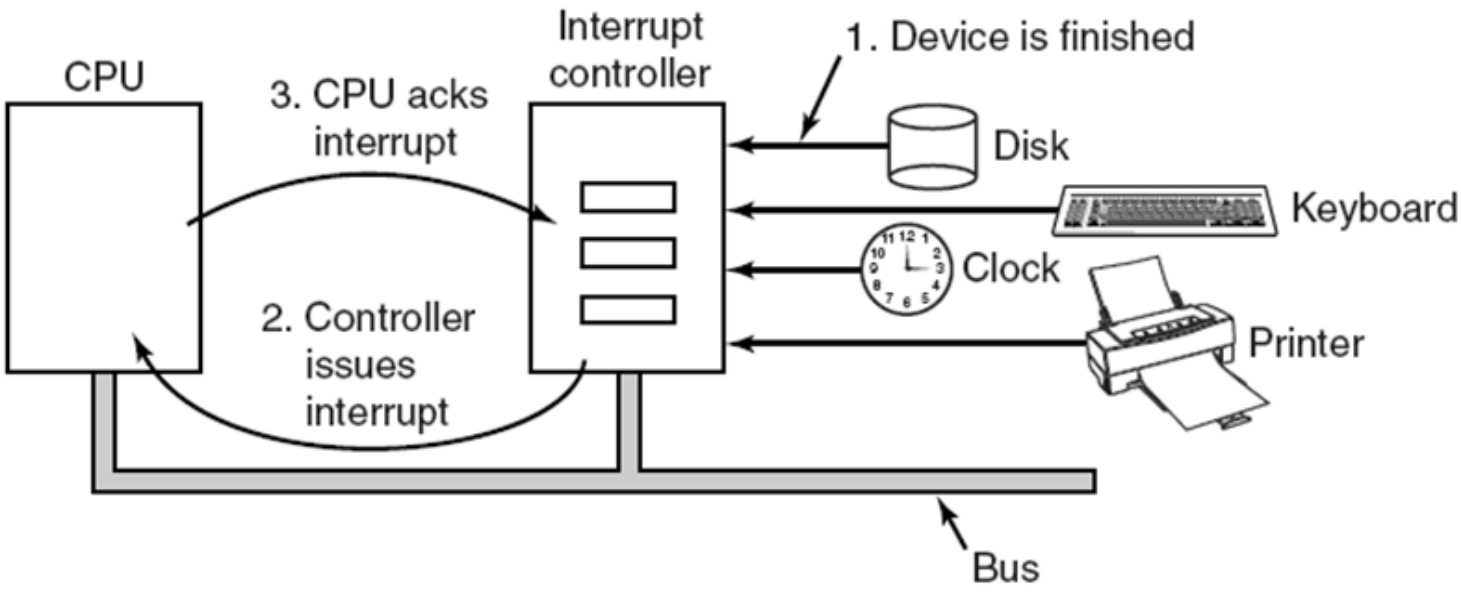


$\mapsto$ when we spoke about **middlewares** for robotics

2) uniform naming



3) low-level control, handle interrupts, fix errors ..



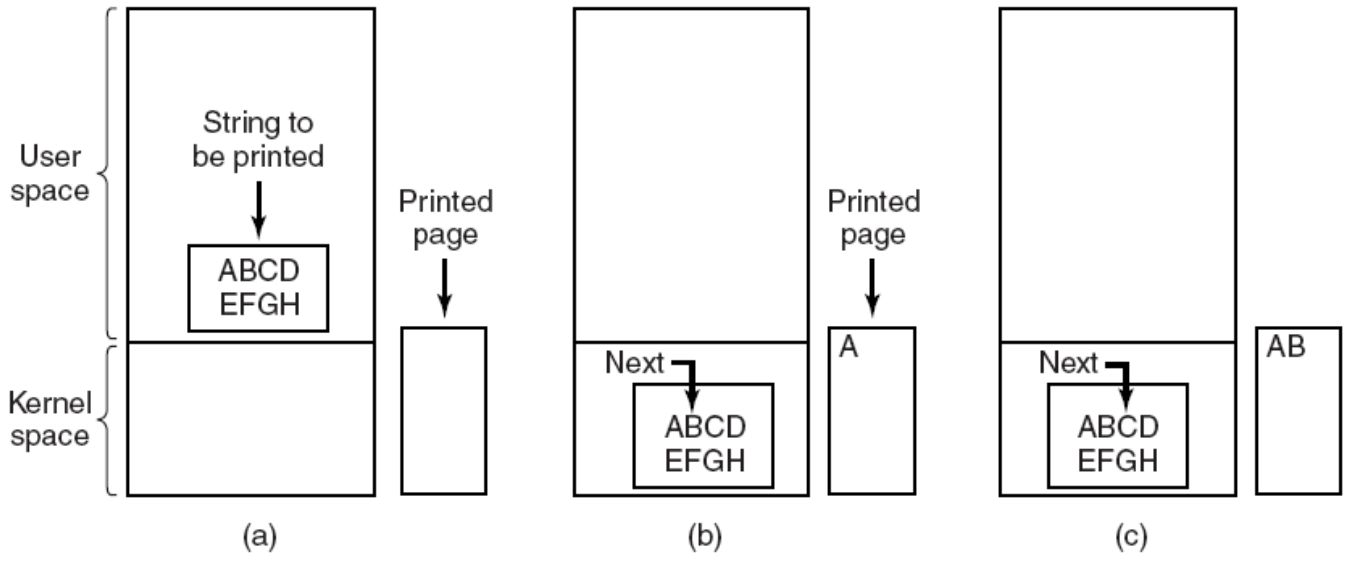
$\mapsto$ the OS must "hide" this to the user program ..

- 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.

4) managing how programs can access to the device

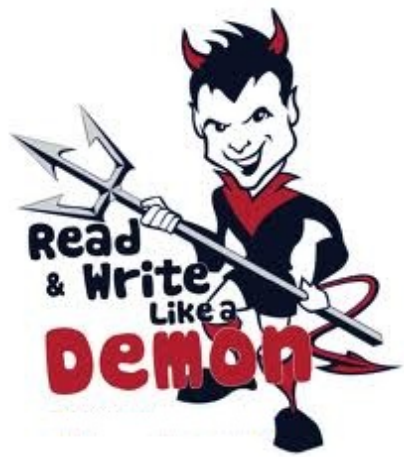


$\mapsto$ example : a process printing a string

what if the process holds the device ?
what if other processes need to use the device ?

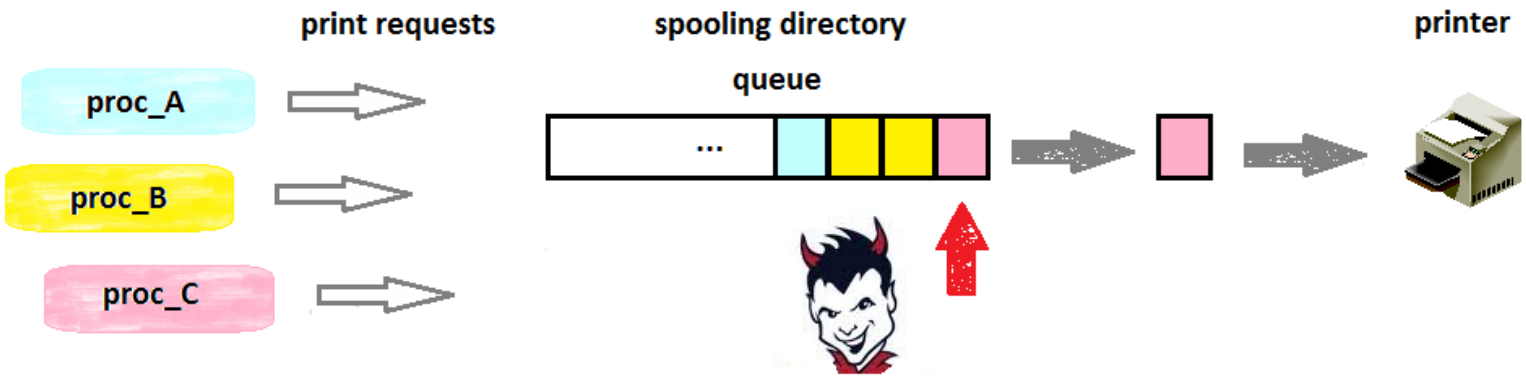
- Sometimes programs do not I/O directly on the device
- An intermediate process is usually interposed between the multitude of user processes and the device
- The idea is to prevent direct access to the device to the users, to prevent a bad process to keep the file open when not using it, or monopolize the resource
- An example : using a printer in a multi-program fashion, without giving the permission to access the printer directly to processes

$\mapsto$ **spooling & demon**

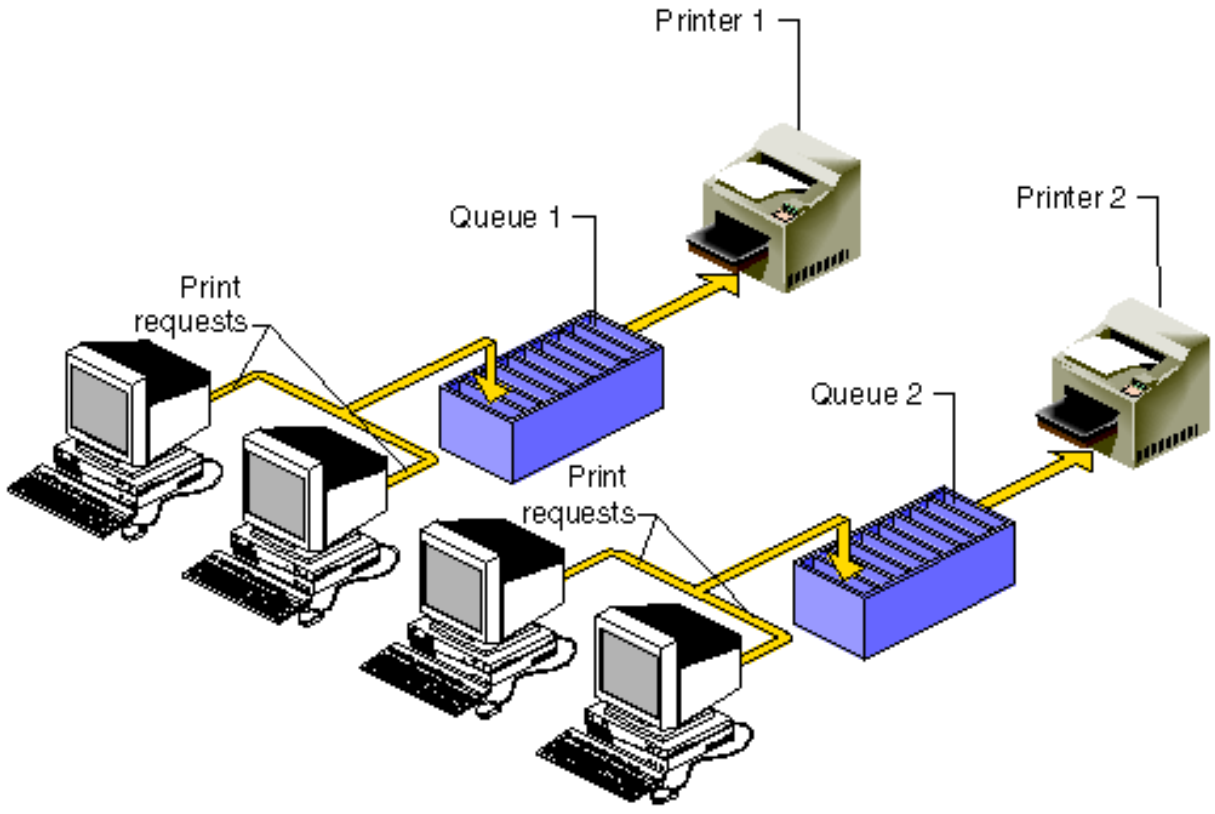


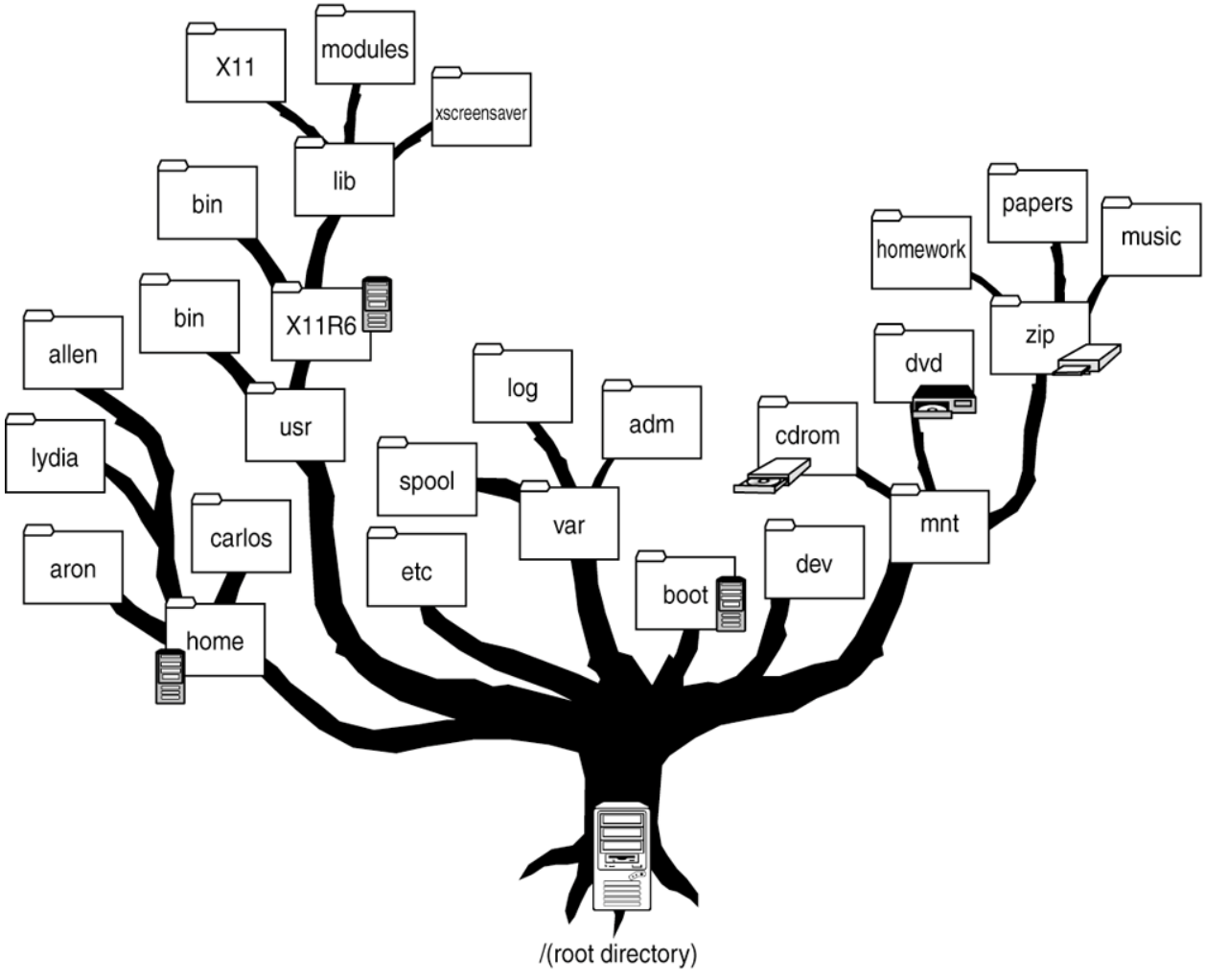
Spooling & demon

- A special directory is created, aka **spooling directory**
- All processes that want to use the printer (but are not allowed to use it !), put their file in the spooling directory
- A **demon process** is the only process allowed to use the special file of the printer
- The demon manages the printing process, taking files from the spooling directory and sending files to the printer



- Spooling is also used for network file transfers (e.g. sending e-mails, shared printers, ..)




X11
modules
xscreensaver
lib
bin
bin
X11R6
allen
usr
lydia
carlos
aron
home
log
adm
spool
var
etc
boot
dev
cdrom
mnt
dvd
homework
papers
music
zip
/(root directory)

- 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 répertoire

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

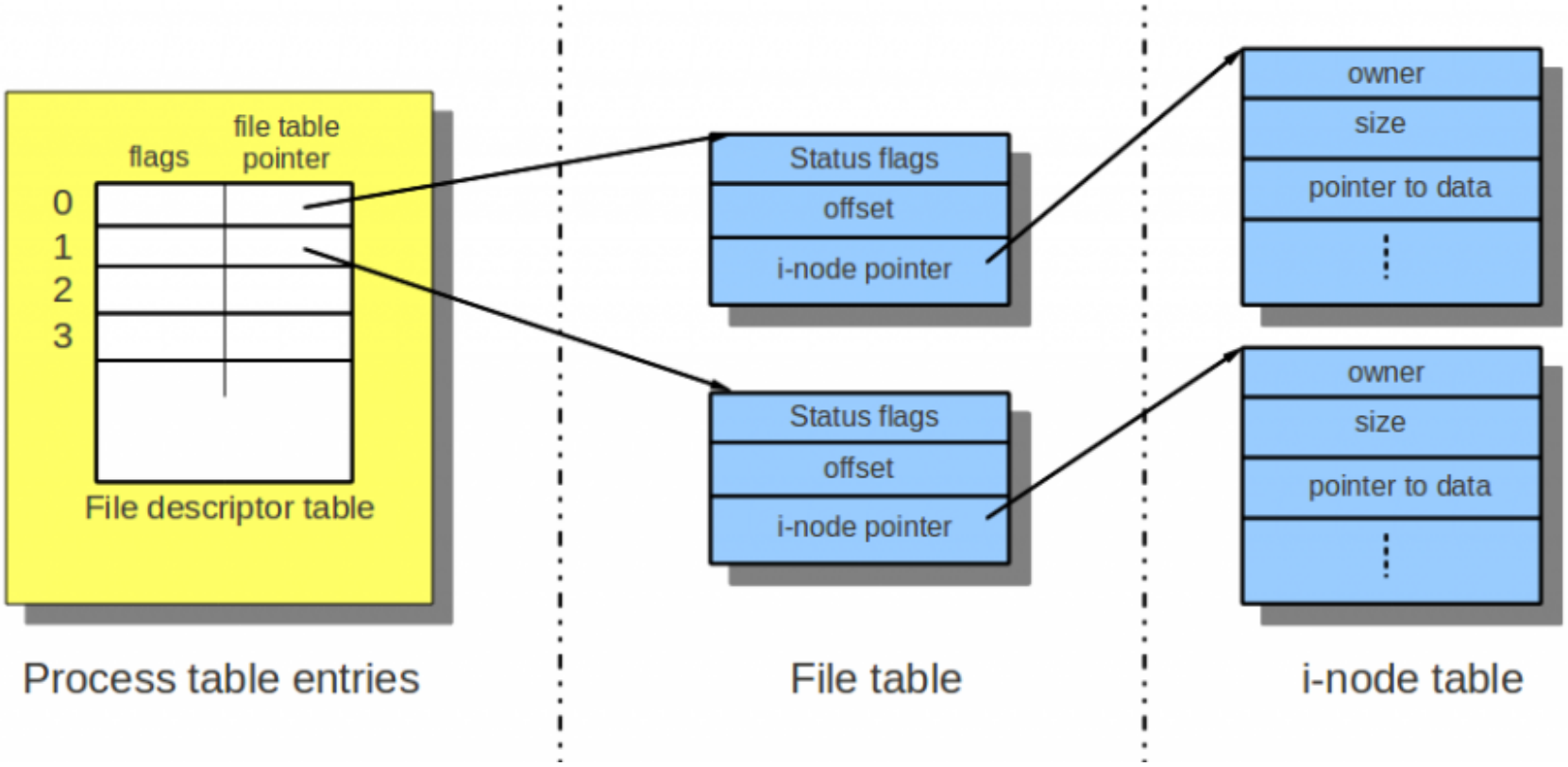
```
int stat(const char *path, struct stat *buf);
```

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**.

Accès séquentiel

- Appels système de bas niveau (`open()` , `read()` , `write()` , ...)
- Les données sont traitées dans l'ordre où ils se trouvent dans le fichier (octet par octet) ;
- Lectures et écritures binaires uniquement
- Mode d'accès simple, pas forcément pratique, moins portable

Accès direct

- Fonctions de la bibliothèque `stdio` (`fopen()` , `fprintf()` , `printf()` , ...)
- accès bufferisé : le fichier est chargé (tout ou en partie) dans un buffer en mémoire centrale ;
- Nombreuses fonctions d'entrées/sorties haut niveau (texte, écriture ou lecture formatée, etc.)
- Mode d'accès plus élaboré, souvent plus pratique et plus rapide

- 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**.

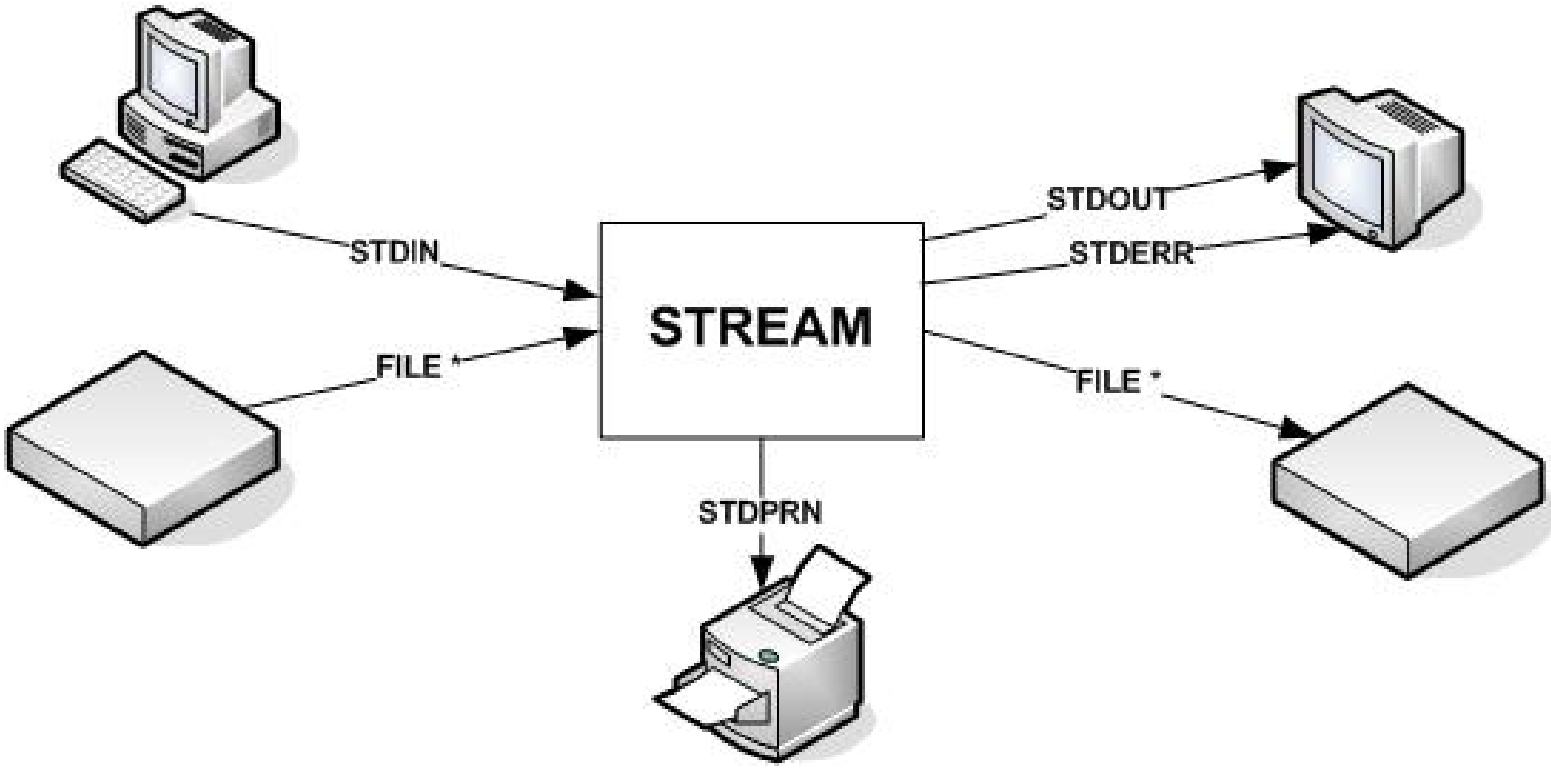
Fichiers de type bloc

- Correspondent aux périphériques structurés en blocs : disque dur, CDROM.
- Rarement manipulés directement par l'utilisataire

Fichiers de type caractère

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

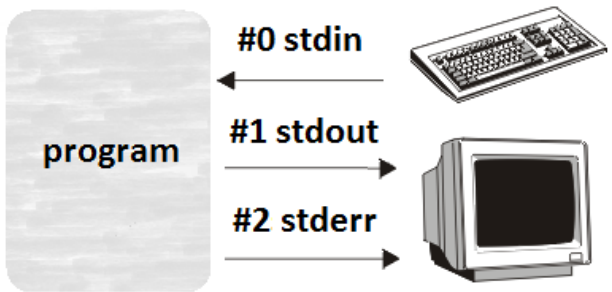
- Fonctions associées aux fichiers spéciaux et aux périphériques : `mknod()` (cf. cours sur les tubes), `ioctl()`, `select()` (pas traitées dans ce cours).


STDIN
FILE *
STREAM
STDOUT
STDERR
FILE *
STDPRN

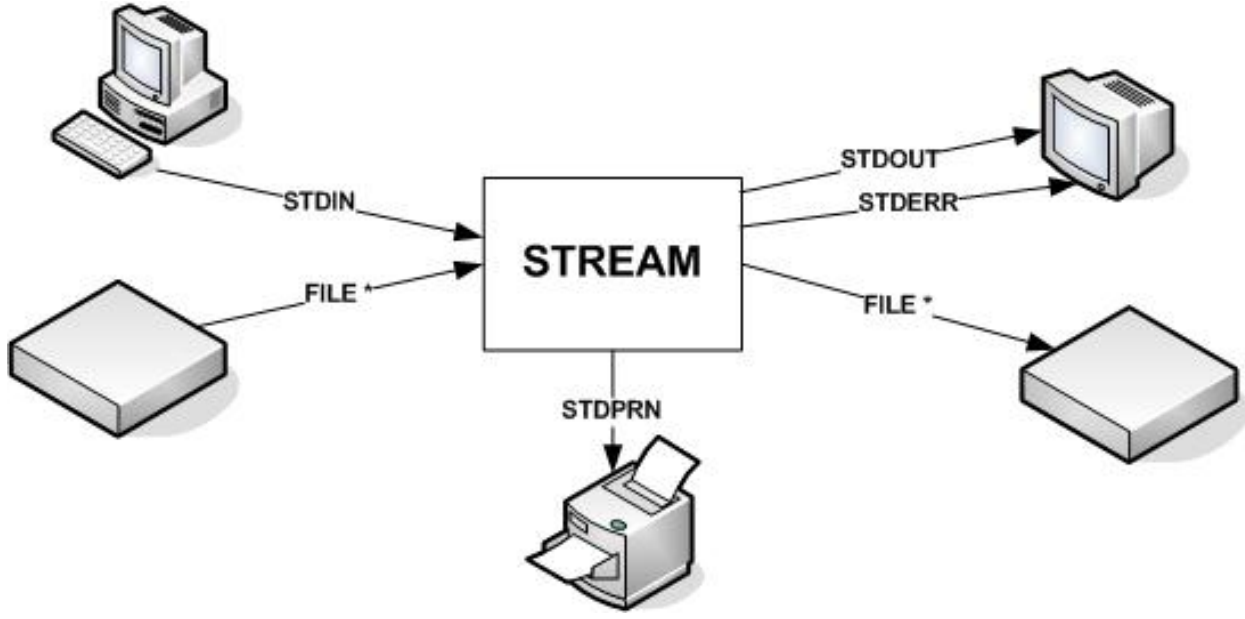
Streams

- **stream** = a file or a physical device
  - `FILE` structure
- flexible, portable way of reading and writing data
- streams are always manipulated through a **pointer**
  1. open a stream ..
  2. .. then access it (to read/write) ..
  3. .. then close it

Predefined streams in UNIX



- `stdin,stdout` can be used for files, programs, I/O devices (keyboard, console, ..)
  - `stdout` : default is console
  - `stdin` : default is keyboard
- `stderr` always prints to console or screen



Redirection

- Redirect `stdout` to a file : $>$
  `my_program > output_file`
- Redirect `stdin` from a file to a program : $<$
  `my_program < input_file`
- Redirect the `stdout` of a program to the `stdin` of another : | (**pipe**)
  `my_program_sending_output | program_receiving_input`

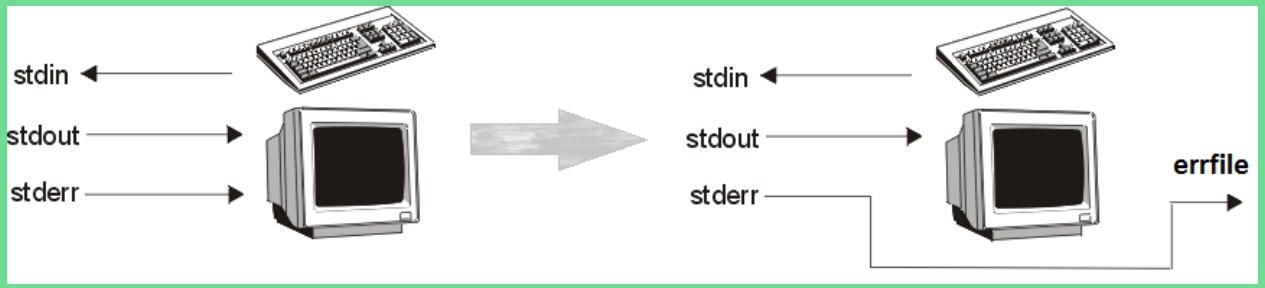
Example : printing

Using `fprintf` on preopened `stdout` stream.

```
1  #include <stdio.h>
2  void main()
3  {
4      fprintf(stdout, "a line of text\n");
5  }
```

Functionally identical to using `printf`

TEST

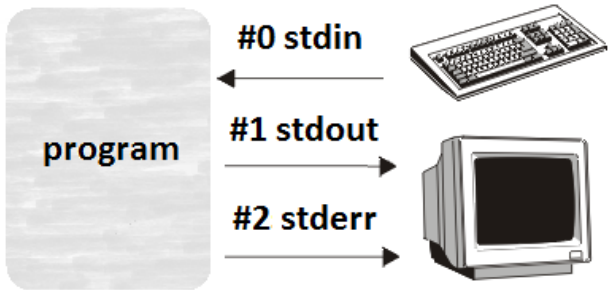



```c
#include <stdio.h>
int main(int argc, char* argv)
{
   char buff[200];
   int j=sprintf(buff,"This␣goes␣to␣stdout");

   printf("res␣:%s␣\n␣size␣:␣%d\n", buff, j);
         return 0;
}
```

- How can we redirect stderr to stdout of program myProgram ?
- How can we redirect stdout of program myProgram to errfile ?
- How can we redirect stderr of program myProgram to errfile ?

What is written on errfile ?

Predefined streams in UNIX



Formatted I/O :

- `int printf(char *format, ...)` : prints to `stdout`
- `int scanf(char *format, ...)` : reads from `stdin`
- `int fprintf(FILE *stream, char *format, args..)` : prints to `stream` (e.g. `stderr`)

Example : formatting output

```
1  #include <stdio.h>
2
3  void main( )
4  {
5     char  buffer[200];
6     char  s[] = "icub";
7     char  c = 'c';
8     int   i = 42;
9     float pi = 3.14159265f;
10
11    int j;
12    j  = sprintf( buffer,     "String:␣␣␣␣%s\n", s );
13    j += sprintf( buffer + j, "Character:␣%c\n", c );
14    j += sprintf( buffer + j, "Integer:␣␣␣%d\n", i );
15    j += sprintf( buffer + j, "Float:␣␣␣␣␣%f\n", pi );
16
17    printf( "Output:\n%s\ncharacter␣count␣=␣%d\n", buffer, j );
18 }
```

- 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` (qui n'a pas besoin d'être déclarée).
- 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
```

```
1  #include <stdio.h>
2
3  int main(){
4      FILE *pF;
5      pF = fopen("./test.txt","r");
6      if(!pF)
7          perror("Ouverture");
8      return 0;
9  }
```

errno

```
#include <errno.h>
extern int errno;
```

- errno is a special system variable that is set when a system call fails
- to be used in a C program, it has to be declared as extern

Error functions

```
#include <string.h>
char *strerror(int errnum);

#include <error.h>
void error(int status, int errnum, const char *format, ...);

#include <stdio.h>
void perror(const char *s);
```

Note :

- error flushes stdout and prints over stderr
- perror describes the last error encountered during a call to a system or library function, returned to errno

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 sur une chaine de caractère 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 formatées en mode texte

fprintf () / fscanf () : déjà vu

Lecture et écriture NON formatées en mode texte

- Plus simple, plus rapide
- **int** puts(**const char** *s) , **int** fputs(**const char** *s, FILE *stream) : identique printf et fprintf sans le formatage.
- **int** putchar(**int** c) , **int** fputc(**int** c, FILE *stream) : écrit un caractère (c est converti en **unsigned char**)

```
1 puts("Hello world !"); // ecrit Hello world !
2 fputs("Hello world !", stdout); // idem
3 putchar('A'); // ecrit A
4 putchar(65); // idem (A = code ASCII 65)
```

- Entrées : **int** getchar() ; **int** fgetc(FILE *stream) : lit un caractère (le renvoie casté en int)
- **char** *fgets (**char** *string , **int** n, FILE *stream)
  - Lit n-1 caractères dans stream , ou jusqu'à tomber sur une fin de chaîne (\0)
  - Les stocke dans string , avec un marqueur de fin de chaîne (\0)
  - Renvoie string si quelque chose a été lu ou NULL sinon

```
1 char buffer_texte[64];
2 fgets(buffer_texte, 64, stdin); // Lit au plus 63 caracteres
```

Le cas gets

- Il existe une fonction : `char *gets(char *string)`
  - Lit des caractères dans `stdin` jusqu'à tomber sur une fin de chaîne (\0) et les stocke dans `string` .
- Problème : **aucun moyen de contrôler combien de caractères sont lus !**
  - Quelle que soit la taille N de `string` , si on lit N+1 caractères dans stdin on obtient une erreur mémoire !
  - Nombreux bugs, vecteur de nombreuses attaques, comportement imprévisible.
- **A ne jamais utiliser !** A la place, utiliser `fgets` sur stdin (slide précédente)

```
GETS(3)                    Linux Programmer's Manual                    GETS(3)

NAME
       gets - get a string from standard input (DEPRECATED)

SYNOPSIS
       #include <stdio.h>

       char *gets(char *s);

DESCRIPTION
       Never use this function.

       gets()  reads  a  line from stdin into the buffer pointed to by s until
       either a terminating newline or EOF, which it replaces with a null byte
       ('\0').  No check for buffer overrun is performed (see BUGS below).
```

Lecture en mode binaire

- 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 en mode binaire

- 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.

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 en mode binaire

- `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 en mode binaire

- `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`

- 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);
```

- 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.

Division par 8 d'un mot binaire représentant un entier

$73 >> 3 = 9$

Note : bitwise operators

A	B	A&B	A | B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Example 1 : what is the result of $73 >> 3$?

Example 2 : find the value of the i-th bit of a certain sequence, for example the 5-th bit of $x = 11011001$.

Accès séquentiel (open() , read() , write() , etc.)

- Opèrent sur un descripteur de fichiers
- Appels système bas niveau spécifiques à UNIX
- Accès non buffurisés

Accès direct (fopen() , fread() , fwrite() , etc.)

fprintf

- Opèrent sur une structure FILE
- Fonctions standard POSIX, disponibles sous de nombreux systèmes (sous Linux, utilisent en interne open , read etc.)
- Accès buffurisés, souvent plus rapides mais d'un usage plus complexe

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`

```
FILE *fp;
fp = fopen("path", w)
...
```

Code :

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

int main() {
    FILE *fichier_source, *fichier_destination;
    char buffer[1024]; // Tampon pour lire les données
    size_t lu;

    // Ouvrir le fichier source en lecture
    fichier_source = fopen("mon_fichier_source.txt", "rb");
    if (fichier_source == NULL) {
        perror("Erreur lors de l'ouverture du fichier source");
        return 1;
    }

    // Ouvrir le fichier destination en écriture
    fichier_destination = fopen("mon_fichier_destination.txt", "wb");
    if (fichier_destination == NULL) {
        perror("Erreur lors de l'ouverture du fichier destination");
        fclose(fichier_source); // Fermer le fichier source en cas d'erreur
        return 1;
    }

    // Lire et écrire les données par blocs
    while ((lu = fread(buffer, 1, sizeof(buffer), fichier_source)) > 0) {
        if (fwrite(buffer, 1, lu, fichier_destination) != lu) {
            perror("Erreur lors de l'écriture dans le fichier destination");
            break;
        }
    }

    // Fermer les fichiers
    fclose(fichier_source);
    fclose(fichier_destination);

    if (ferror(fichier_source) != 0) {
        perror("Erreur de lecture");
    }
    if (ferror(fichier_destination) != 0) {
        perror("Erreur d'écriture");
    }

    return 0;
}
```

Αυτό είναι καινούριο

Exercice 2

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

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

```
$ ./fileTransfer < input.txt
hello my name is icub
i am a humanoid robot
and i am happy
File ended
```

```
$ ./fileTransfer < input.txt > output.txt
File ended
```

Utiliser : `fgets, fputs`

Code :

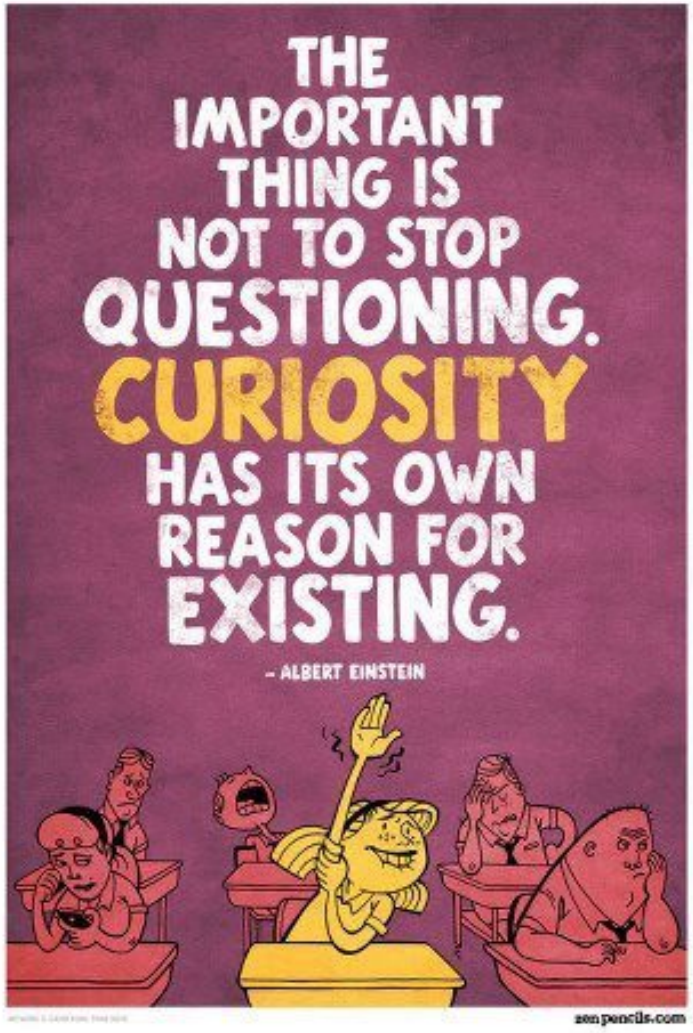
```
#include <stdio.h>

#define MAX_LINE_LENGTH 1024

int main() {
    char line[MAX_LINE_LENGTH];

    while (fgets(line, MAX_LINE_LENGTH, stdin) != NULL) {
        fputs(line, stdout);
    }

    printf("File ended\n");
    return 0;
}
```

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.

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

#define MAX_LINE_LENGTH 1024

int main() {
    FILE *fichier_source, *fichier_destination;
    char lignes[MAX_LINE_LENGTH][MAX_LINE_LENGTH]; // Tableau pour stocker les lignes
    int nb_lignes = 0;
    char line[MAX_LINE_LENGTH];

    // Ouvrir le fichier source en lecture
    fichier_source = fopen("mon_fichier_source.txt", "r");
    if (fichier_source == NULL) {
        perror("Erreur lors de l'ouverture du fichier source");
        return 1;
    }

    // Lire toutes les lignes du fichier source et les stocker dans le tableau
    while (fgets(line, MAX_LINE_LENGTH, fichier_source) != NULL) {
        strcpy(lignes[nb_lignes], line);
        nb_lignes++;
    }

    // Fermer le fichier source
    fclose(fichier_source);

    // Ouvrir le fichier destination en écriture
    fichier_destination = fopen("mon_fichier_destination.txt", "w");
    if (fichier_destination == NULL) {
        perror("Erreur lors de l'ouverture du fichier destination");
        return 1;
    }

    // Écrire les lignes dans l'ordre inverse
    for (int i = nb_lignes - 1; i >= 0; i--) {
        fputs(lignes[i], fichier_destination);
    }

    // Fermer le fichier destination
    fclose(fichier_destination);

    return 0;
}
```

Questions ?


THE IMPORTANT THING IS NOT TO STOP QUESTIONING. CURIOSITY HAS ITS OWN REASON FOR EXISTING.
– ALBERT EINSTEIN
zenpencils.com