# Les fichiers

Il existe plusieurs façons de travailler avec les fichiers :
Grace à une interface de haut niveau, bufférisée, utilisant des structures FILE (fopen(), fread(), fwrite() et fclose() ) :
Au contraire en bas niveau, avec des entiers identifiants les fichiers, et grâce aux commandes open(),read(),write() et close().
L'interface de haut niveau ne nous concerne pas ici. En Système, nous travaillons en bas niveau.

# Ouverture d'un fichier

int open(char * pathname, int flags, mode_t mode)
le pathname, c'est le nom du fichier (trivial)
les flags permettent de décrire le mode d'ouverture (O_RDONLY, O_WRONLY ou O-RDWR). A combiner avec d'autres attributs tels que O_CREAT, O_APPEND ou O_TRUNC). On combine ces attributs avec des OU (|)
Enfin, les modes correspondent aux droits du fichiers.
Open() renvoie un entier, le numéro du fichier (en théorie à partir de 3, car 0, 1 et 2 sont déjà utilisés... Tiens... Pourquoi d'ailleurs ?

```
int fd, fd_in;


/* ouverture du fichier */
 fd=open("/tmp/message",O_WRONLY | O_TRUNC | O_CREAT,"0400");
 if (fd==-1) {
    perror("erreur création fichier");
    exit(-1);
 }

  fd_in=open(argv[1], O_RDONLY); //le fichier qu'on va lire
   if (fd_in == -1 ) {
     perror("open du fichier");
     exit(-1);
   }
```

# fermeture du fichier

int close(int fd)
Ferme le fichier; et renvoie 0 ou -1 si erreur.

# Lecture et écriture dans un fichier

ssize_t read(int fd, void *buffer, size_t taille)
ssize_t write(int fd, void *buffer, size_t taille)

Un buffer est un espace d'une taille définie arbitrairement qu'on pourra manipuler... On peut par exemple utiliser un tableau de XXX  char dans notre cas.
Char buffer[1024]; est un excellent candidat pour ce genre de choses.
Un tableau de char comme celui-ci est en fait un pointeur sur le début de la zone, donc, l'utilisation

est assez simple.
Lecture : On indique le nombre de caractère qu'on veut lire (taille). La primitive read() déclenche la lecture et remplit le buffer. Elle renvoit le nombre de caractères lus ou -1 en cas de problème. Le nb de caractères peut être inférieur à la taille demandée, en cas de fin de fichier, ou si on lit des fichiers un peu particulier. (on gardera bien la valeur de result, si on veut manipuler ensuite le buffer, pour éviter de sortir de l'espace utilisé)
Ecriture : idem, on donne le buffer à écrire, et la taille du buffer.

```
char buffer[1024] ;
read(fd,buffer,1024); //remplira le buffer de 1024 caractères lus max..
write (fd2, buffer, 1024); //écrira les 1024 caractères du buffer (en espèrant qu'ils existent !!)

//Nota : read renvoie 0 quand il n'arrive plus à lire, ou la taille qu'il a réellement lu
```

## Exercice 1

Utiliser l'appel système open() pour ouvrir un fichier qui sera spécifié sur la ligne de commande. Lire les 10 premiers octets du fichier avec l'appel système read(), puis les afficher sur la sortie standard en utilisant write(). Rappel : 0=stdin, 1=stdout, 2=stderr.
Toujours en utilisant un buffer statique, écrire un programme fonctionnant comme cat, c'est-à-dire ouvrant un par un les fichiers spécifiés sur la ligne de commande, les lisant, et les écrivant sur la sortie standard.

## Exercice 2 -

Réécrire un programme cp simple (copie d'un fichier vers un autre). Pour créer le fichier "cible",n'oubliez pas d'utiliser les flags O_CREAT, O_TRUNC, et le troisième paramètre de open(), pour positionner correctement les droits du fichier nouvellement créé (regardez ce qui se passe si vous ne mettez pas cet argument).

## Dup() et dup2()

```
#include <unistd.h>
int dup(int oldfd);int dup2(int oldfd, int newfd);
```

**dup()** et **dup2()** créent une copie du descripteur de fichier *oldfd*.
Après un appel réussi à **dup()** ou **dup2()**, l'ancien et le nouveau descripteur peuvent être utilisés de manière interchangeable.

**dup()** utilise le plus petit numéro inutilisé pour le nouveau descripteur. **dup2()** transforme *newfd* en une copie de *oldfd*, fermant auparavant *newfd* si besoin est. Ce dernier permet donc de masquer complétement la substitution.

**dup()** et **dup2()** renvoient l'entier correspondant au nouveau descripteur.

**dup : duplique un descripteur et renvoie le premier descripteur libre dans la table du processus**

```
int fd = open("tutu",O WRONLY|O CREAT) ;
close(STDOUT FILENO) ;
dup(fd) ; // renvoie 1
```

```
close(fd) ; // ne sert plus à rien
printf("j'écris un truc") ; // écrit dans tutu
```

**dup2 : permet de choisir le descripteur (et écraser l'ancien)**

```
 int fd = open("tutu",O WRONLY|O CREAT) ;
dup2(fd,STDOUT FILENO) ;
close(fd) ;
printf("tutu") ; // écrit dans tutu
```

Dans le premier exemple, voici un état des descripteurs

| descripteur | Au début | Après open | Aprés close | Après dup | Après close |
|---|---|---|---|---|---|
| 0 | stdin | stdin | stdin | stdin | stdin |
| 1 | stdout | stdout | **rien** | **tutu** | tutu |
| 2 | stderr | stderr | stderr | stderr | stderr |
| 3 | rien | **tutu** | tutu | tutu | **rien** |

Il est important de se souvenir que lors d'un **fork**(), ou d'un **exec**(), les descripteurs de fichiers sont préservés (donc, le code de recouvrement d'un **exec**() utilisera les descripteurs de fichiers actuels, et, dans le cas d'un **fork**(), le processus forké aussi).

# Exercices

Écrire un programme qui redirige la sortie standard vers un fichier, puis effectue un "**ls**" vers ce fichier. Il faut donc ouvrir ce fichier avec **open**(), puis utiliser **dup()** ou **dup2**() afin que les accès ultérieurs à la sortie standard se fassent sur le fichier. Ensuite, on peut faire un **execlp**() pour lancer **ls**, et l'affichage se fera tout seul vers le fichier.

Révisions sur **fork**() : écrire un programme qui exécute le programme spécifié sur la ligne de commande, et en fonction de son code de retour, affiche "OK" ou "ERREUR". Rappels: pour récupérer le code de retour, utiliser waitpid(); et OK correspond à un code de retour nul.

```
Exemple:
programme eject doit afficher OK si le cdrom s'est ouvert, et sinon ERREUR.
```

Modifier le programme précédent pour qu'il n'affiche rien d'autre que OK ou ERREUR, c'est-à-dire que le programme qui sert de "condition" ne doit rien afficher. Pour l'empêcher de faire de l'affichage, on propose de rediriger la sortie vers le fichier /dev/null. Attention, il ne faut pas faire un simple exec mais un fork+exec. Attention aussi à ne détourner la sortie standard que dans le fils, afin que le père puisse encore afficher OK ou ERREUR à la fin du programme.

# Les pipes

Int pipe(int fd[2]);
**pipe** crée une paire de descripteurs de fichiers, pointant sur un i-noeud de tube, et les place dans un tableau *filedes*. *filedes[0]* est utilisé pour la lecture, et *filedes[1]* pour l'écriture.

En général deux processus (créés par fork) vont se partager le tube, et utiliser les fonctions **read** et **write** pour se transmettre des données. Donc, on va ECRIRE dans filedes[1], et de l'autre coté, un

autre processus lira dans filedes[0]..

*Question : Comment les deux processus ont ils accès à ces descripteurs ???*

On pourra alors tout combiner, des close(), des dup() ou dup2(), pour mettre en place de la communication entre les processus.

Pipe() renvoie -1 si il y a eu un problème.

```
if ( pipe(p) == -1 ) {
  perror("pipe");
  exit(-1);
}
```

# Exercices

En utilisant fork(), créer deux processus communiquant par un tube (lui-même créé avec l'appel système pipe()). Le fils lira depuis l'entrée standard et écrira dans le tube les caractères mis en majuscules avec toupper(). Le père lira depuis le tube et écrira sur la sortie standard.

Dans cet exercice, on appelle co-processus un programme lancé par un autre programme, et contrôlé par ce dernier par l'intermédiaire de ses entrée et sortie standards. On veut ici écrire un programme dispatch, et des programmes addition, multiplication, soustraction (par exemple). Lorsqu'on lance addition, ce programme attend 2 nombres sur son entrée standard, séparés par des '\n', et renvoie leur somme sur la sortie standard (puis il attend à nouveau deux nombres). Ensuite, on lance dispatch, qui doit effectuer des opérations en utilisant les co-processus.

Écrire les programmes **addition, multiplication, soustraction**. Utiliser scanf pour lire les nombres, printf pour les afficher (en clair: faire simple, rapide, concis)

Écrire le programme **dispatch**, qui doit commencer par lancer les co-processus, puis attend sur l'entrée standard un ordre du style "addition 2 4". Avec strcmp(), trouver quel est le co-processus à utiliser, lui envoyer l'ordre, puis afficher le résultat.

*Nota (attention, le pipe ne fonctionne que dans un sens !)*