Les entrées/sorties

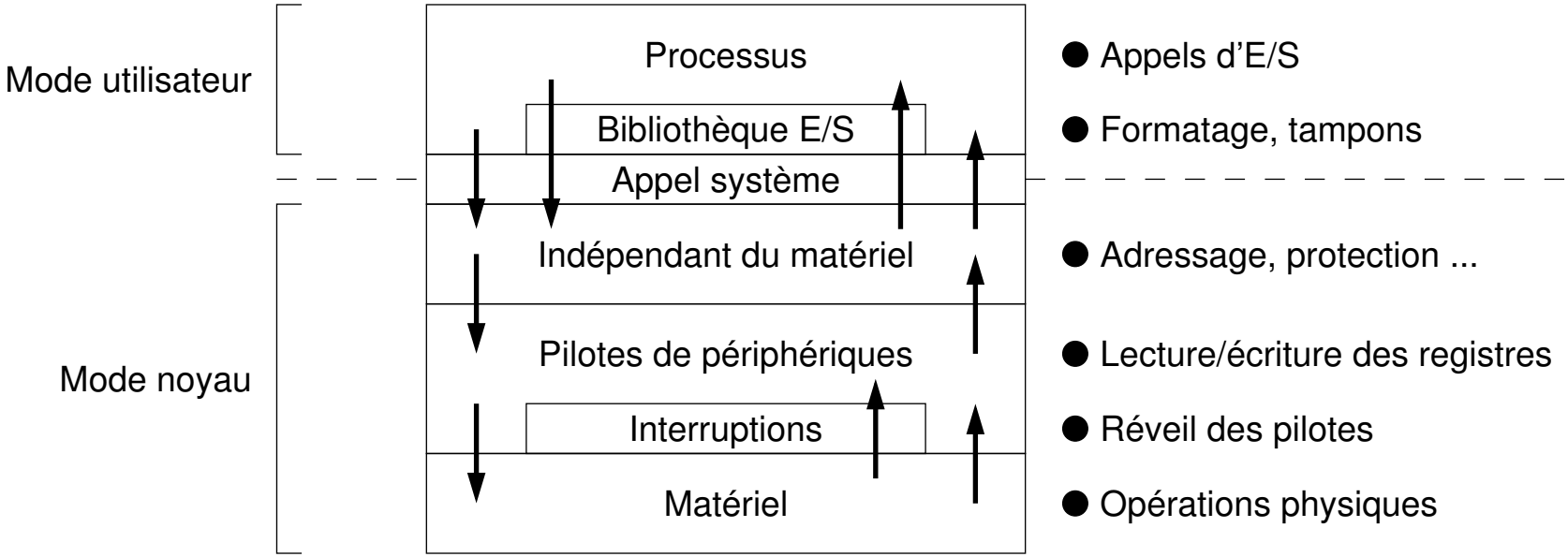
Descripteurs de fichiers
Parcours des répertoires
Tubes de communication
Flux de haut niveau
Projection en mémoire

Fabrice Harrouet
École Nationale d'Ingénieurs de Brest
harrouet@enib.fr
http://www.enib.fr/~harrouet/

Principes

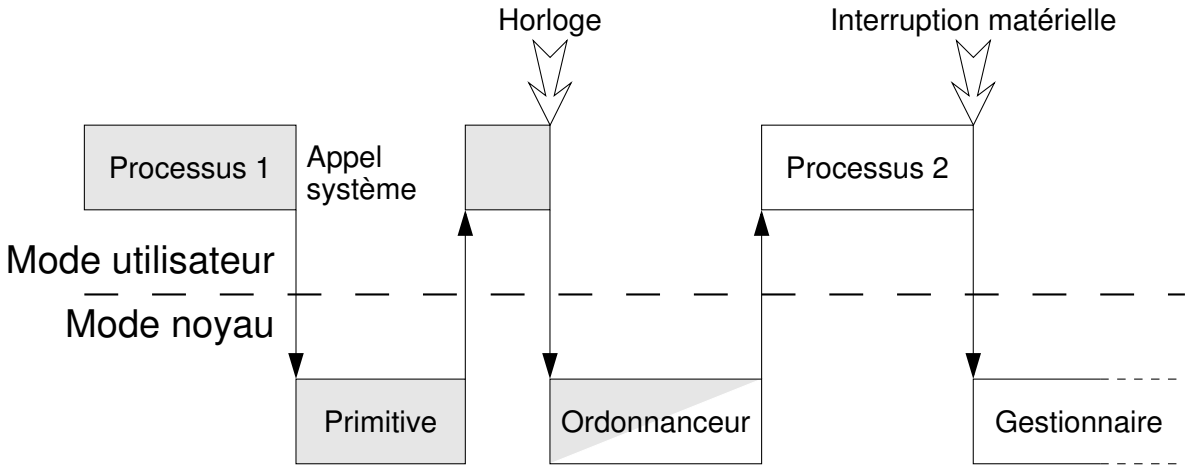
- **De manière générale :** *"tout est fichier"* **!**
  - Mise en œuvre homogène des entrées/sorties
  - Se résume à des lectures/écritures dans des flots
  - Réutilisation des traitements dans divers contextes
  - Généralisation du terme *"fichier"*
    - Fichier *"réel"* d'un système de fichiers
    - Terminal
    - Tube de communication
    - *Socket*
    - ...
- **Deux notions principales**
  - Les descripteurs de fichiers (système)
  - Les flux de données (espace utilisateur)

Les couches d'entrées/sorties



Mode noyau / mode utilisateur

- **Mode utilisateur**
  - Mode de fonctionnement *"normal"* d'un processus
  - Traitements dans son propre espace d'adressage
  - Pas de risque majeur (sauf pour lui)
- **Mode noyau**
  - Accès au matériel, à la mémoire physique
  - Gestion des processus



Appel système $\not\equiv$ fonction

- **Appel système** (`man 2 service`)
  - Permet d'accéder aux services du système
    - Passage en mode noyau
    - Vérification si processus autorisé
  - Mécanisme d'interruption très couteux en temps
    - Sauvegarde des données du processus ...
- **Fonction** (`man 3 service`)
  - Calcul dans l'espace utilisateur
  - Encapsulation des appels systèmes
    - Services de plus haut niveau
    - Regrouper plusieurs invocations en une seule
    - Rôle de la bibliothèque standard du *C*

Quelques précautions

- **Utilisation de** `errno` (`man 3 errno`)
  - `#include <errno.h>`
    `extern int errno;`
  - Indicateur positionné en cas d'erreur
    - Depuis les appels systèmes
    - Depuis quelques fonctions de bibliothèque
  - Un ensemble de constantes indiquant la nature de l'erreur
  - Utilisation classique
    - Effectuer l'appel
    - Si le résultat indique une erreur (`-1` généralement)
      → Lire la valeur de `errno`

Quelques précautions

▷ **Utilisation de `errno`**

◇ Description d'une erreur (`man 3 strerror`)

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

◇ Retourne une chaîne allouée statiquement décrivant l'erreur `errnum`

◇ Signaler une erreur (`man 3 perror`)

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

◇ Écrit `msg` et une description de `errno` dans la sortie d'erreur

◇ `msg` peut être un pointeur nul

Quelques précautions

- **Les appels sytèmes** *"lents"*
  - Certains appels systèmes sont atomiques
    - Information immédiatement disponible dans le noyau
  - D'autres peuvent prendre du temps (*"lents"*)
    - Il faut attendre les informations
  - Les processus peuvent recevoir des signaux (`man 2 sigaction` , `man 2 kill`)
  - Si un processus reçoit un signal en cours d'appel système
    - Le processus est relancé pour traîter le signal
    - Les informations ne sont pas forcément disponibles !
    - L'appel système échoue → `errno` vaut `EINTR`
  - Sans gravité → recommencer l'appel
  - Précisé dans la section `ERRORS` du manuel de l'appel

Utilisation de `errno`

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

#define RESTART_SYSCALL(result,syscall) \
  do { (result)=(syscall); } \
  while(((result)<0)&&(errno==EINTR));

int main(void)
{
int fd;
char buffer[256];
ssize_t nb;
RESTART_SYSCALL(fd,open("XXX",O_RDONLY));
if(fd==-1)
  { perror("open()"); exit(EXIT_FAILURE); }
RESTART_SYSCALL(nb,read(fd,buffer,255));
if(nb==-1)
  { perror("read()"); exit(EXIT_FAILURE); }
buffer[nb]='\0';
fprintf(stderr,"Success ! --> %s\n",buffer);
close(fd);
return EXIT_SUCCESS;
}
```

```
$ ./prog
open(): No such file or directory
$ echo Hello > XXX
$ ./prog
Success ! --> Hello

$ chmod -r XXX
$ ./prog
open(): Permission denied
$ rm XXX
$ mkdir XXX
$ ./prog
read(): Is a directory
$
```

Descripteurs de fichiers

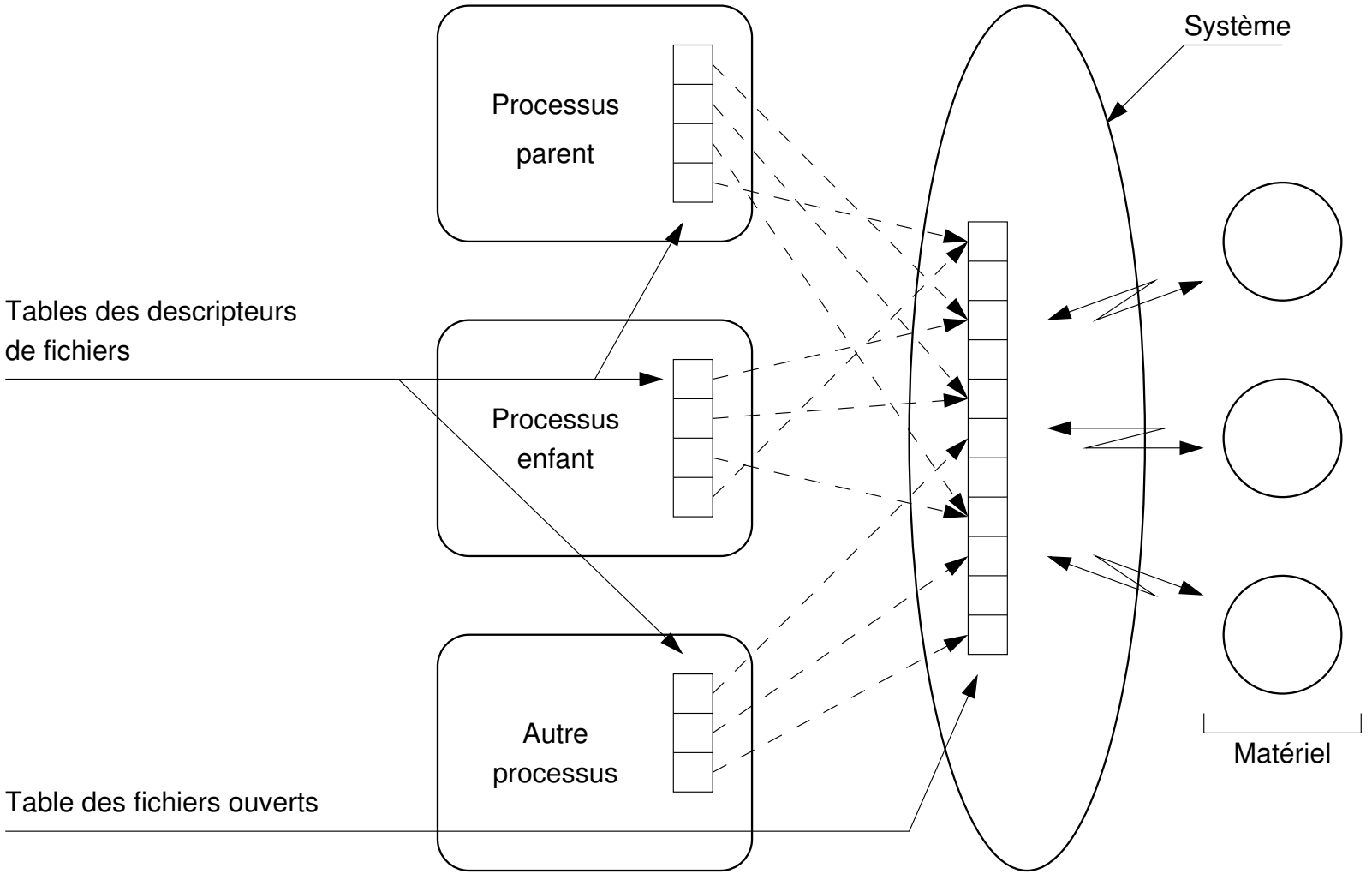
▷ **Identifiant d'un flot** (*File Descriptor*)

- ◇ Un simple entier (type `int`) désignant un flot ouvert
- ◇ *API* relativement indépendante du procédé physique
- ◇ Abstraction plus élevée que les pilotes de périphériques

▷ **Rôle, utilisation**

- ◇ Envoyer/obtenir des blocs de données
- ◇ Données stockées dans l'espace d'adressage du processus
- ◇ Données non formatées, binaires ou textuelles
- ◇ Pas de tampon (le minimum)
- ◇ À chaque opération $\rightarrow$ appel système

Descripteurs de fichiers



Descripteurs de fichiers

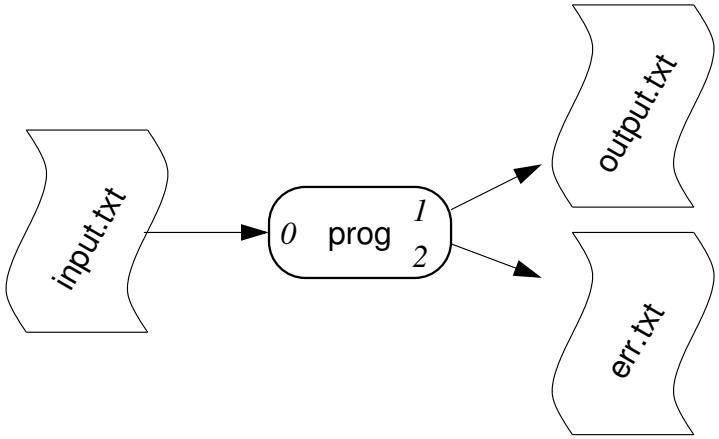
- **Opérations spécifiques**
  - Ouverture du flot
  - Traitements particuliers (positionnement . . .)
- **Opérations génériques** (*"un simple tuyau"*)
  - Lecture/écriture
  - Scrutation
  - Paramétrage
  - Fermeture

Flots par défaut

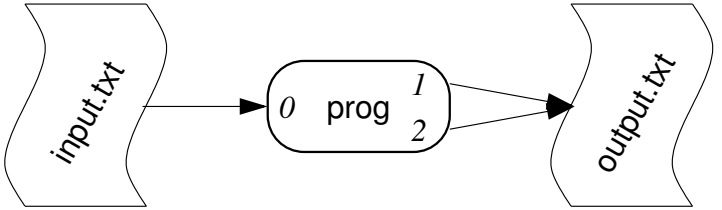
- **Ouverture implicite de 3 flots pour chaque processus**
  - Entrée standard → lecture depuis le terminal
    - Descripteur de fichier `0` (`STDIN_FILENO` dans `unistd.h`)
  - Sortie standard → écriture vers le terminal
    - Descripteur de fichier `1` (`STDOUT_FILENO` dans `unistd.h`)
  - Sortie d'erreurs → écriture vers le terminal
    - Descripteur de fichier `2` (`STDERR_FILENO` dans `unistd.h`)
- **Modifications explicites**
  - Redirection depuis la ligne de commande
  - Redirection/fermeture par le processus parent
  - Redirection/fermeture par le processus lui-même

Flots par défaut

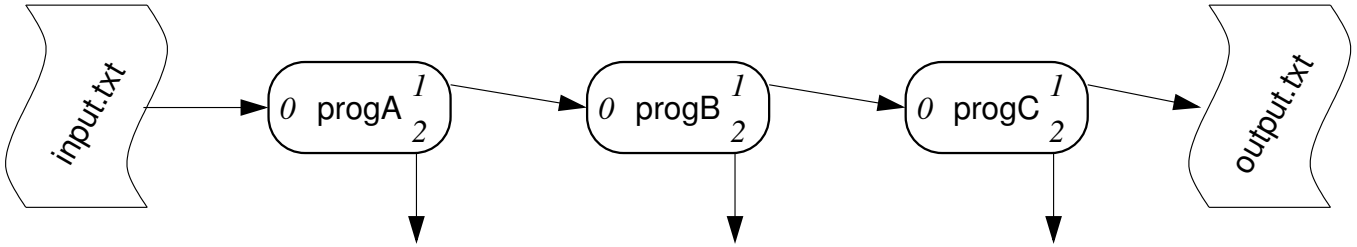
```
$ prog < input.txt > output.txt 2> err.txt
```




```
$ prog < input.txt > output.txt 2>&1
```




```
$ ( progA | progB | progC ) < input.txt > output.txt
```



Ouverture/fermeture d'un flot

- **Cas particulier ici pour l'ouverture**
  - Un fichier *"réel"* dans un système de fichiers
  - Plus général qu'il n'y paraît (`/dev/ttyS0`, `/dev/audio` ...)
- **Ouverture d'un fichier**
  - Appel système `open()` (`man 2 open`)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char * path,int flags,
         ...  /* mode_t mode */);
```

  - `path` : chemin absolu ou relatif du fichier à ouvrir

Ouverture/fermeture d'un flot

- **Ouverture d'un fichier**
  - `flags` : type d'ouverture et paramètres (combinaison bit à bit)
    - Ouverture `O_RDONLY`, `O_WRONLY` ou `O_RDWR`
    - `O_CREAT` : création si inexistant
    - `O_TRUNC` : vider l'ancien contenu
    - `O_APPEND` : ajout en fin de fichier
    - `O_NONBLOCK`, `O_SYNC`, `O_EXCL`, `O_NOCTTY` ...
  - `mode` : droits du fichier si `O_CREAT` (voir aussi `man 2 umask`)
    - Valeur entière (octale) ou combinaison bit à bit
    - `S_ISUID`, `S_ISGID`, `S_ISVTX`
      `S_IRWXU`, `S_IRUSR`, `S_IWUSR`, `S_IXUSR`
      `S_IRWXG`, `S_IRGRP`, `S_IWGRP`, `S_IXGRP`
      `S_IRWXO`, `S_IROTH`, `S_IWOTH`, `S_IXOTH`

Ouverture/fermeture d'un flot

- **Ouverture d'un fichier**
  - Retour de `open()`
    - Un descripteur de fichier si ouverture correcte
    - `-1` si une erreur survient
  - Nombreuses causes d'erreurs (consulter `errno`)
    - Fichier inexistant, droits insuffisants, `flags` incorrects ...
  - Variante : appel système `creat()` (`man 2 creat`)
    - `int creat(const char * path,mode_t mode);`
      $\equiv$ `open(path,O_CREAT|O_WRONLY|O_TRUNC,mode);`

Ouverture/fermeture d'un flot

- **Modes d'ouverture classiques**
  - `open("file.txt",O_RDONLY);`
    - Lecture, le fichier doit exister
  - `open("file.txt",O_WRONLY|O_CREAT|O_TRUNC,0666);`
    - Écriture, créer le fichier si inexistant, le vider si existant
    - Droits `rw-rw-rw-` si création
  - `open("file.txt",O_WRONLY|O_CREAT|O_APPEND,0666);`
    - Ajout en fin, créer le fichier si inexistant
  - `open("file.txt",O_RDWR|O_CREAT,0666);`
    - Lecture/écriture, créer le fichier si inexistant

Ouverture/fermeture d'un flot

▷ Synchronisation d'un flot

- ◇ Forcer l'écriture des tampons (système/driver) sur le périphérique
- ◇ Appel système `fsync()` (`man 2 fsync`)
  - ○ `#include <unistd.h>` `int fsync(int fd);`
- ◇ `fd` : descripteur de fichier quelconque
- ◇ Retour : `0` si OK, `-1` si mauvais `fd` ou flot inadapté

▷ Fermeture d'un flot

- ◇ Très général (pas uniquement les fichiers)
- ◇ Appel système `close()` (`man 2 close`)
  - ○ `#include <unistd.h>` `int close(int fd);`
- ◇ `fd` : descripteur de fichier quelconque
- ◇ Retour : `0` si OK, `-1` si mauvais `fd`

Lecture/écriture dans un flot

- **Lecture depuis un flot**
  - Très général (pas uniquement les fichiers)
  - Appel système `read()` (`man 2 read`)
    - ```
      #include <unistd.h>
      ssize_t read(int fd,void * buf,size_t count);
      ```
  - Extraction de `count` octets de `fd` vers `buf` (préallablement alloué !)
  - Retour : nombre d'octets extraits, `0` si fin de fichier, ou `-1` si erreur
  - Nombreuses causes d'erreurs (consulter `errno`)
    - `EBADF` si mauvais `fd`
    - `EINTR` si interrompu → relance
    - `EAGAIN` si ouverture `O_NONBLOCK` (ou vérouillage) et rien à lire
    - Dépendant de la nature exacte de `fd` ...
      ```

Lecture/écriture dans un flot

- **Gestion** *"subtile"* **du volume extrait !**
  - `nb=read(fd,buf,count);`
  - Si aucun octet n'est prêt au moment de l'appel
    - Blocage du processus → réveil lorsque `fd` devient prêt
    - Si `fd` est non-bloquant (`O_NONBLOCK`) → `nb=-1` , `errno=EAGAIN`
  - Si la *fin de fichier* est atteinte → `nb=0`
    - Les derniers octets lus n'indiquent pas la fin
    - Il faut faire une tentative de trop
  - Si une erreur survient → `nb=-1`
  - Si au moins 1 octet est prêt
    - On obtient dans `buf` les octets prêts
    - **Attention : il faut toujours contrôler le résultat !**
      → `1<=nb<=count`

Lecture/écriture dans un flot

▷ Lire exactement le volume attendu

```
ssize_t /* >0: nb read   0: EOF   -1: error */
readFully(int fd,
          void * buf,
          size_t count)
{
ssize_t r;
size_t remaining=count;
char * ptr=(char *)buf;
while(remaining)
  {
  RESTART_SYSCALL(r,read(fd,ptr,remaining));
  if(r<0) return -1;
  if(!r) break;
  ptr+=r;
  remaining-=r;
  }
return count-remaining;
}
```

Lecture/écriture dans un flot

▷ Écriture vers un flot

- ◇ Très général (pas uniquement les fichiers)
- ◇ Appel système `write()` (`man 2 write`)
  - ○
```
#include <unistd.h>
ssize_t write(int fd,const void * buf,size_t count);
```
- ◇ Envoi des `count` octets de `buf` dans `fd`
- ◇ Retour : nombre d'octets envoyés ou `-1` si erreur
- ◇ Nombreuses causes d'erreurs (consulter `errno`)
  - ○ `EBADF` si mauvais `fd`
  - ○ `EPIPE` si extémité fermée (*tube*, *socket*)
  - ○ `EINTR` si interrompu → relance
  - ○ `EAGAIN` si ouverture `O_NONBLOCK` (ou vérouillage) → relance
  - ○ Dépendant de la nature exacte de `fd` ...

Lecture/écriture dans un flot

- **Gestion** *"subtile"* **du volume envoyé !**
  - `nb=write(fd,buf,count);`
  - Si aucun octet ne peut être envoyé immédiatement
    - Blocage du processus → réveil lorsque `fd` devient prêt
    - Si `fd` est non-bloquant (`O_NONBLOCK`) → `nb=-1` , `errno=EAGAIN`
  - Si une erreur survient : `nb=-1` , `errno=EINTR`
  - Si au moins 1 octet peut être envoyé
    - La quantité de donnée correspondante est prise dans `buf`
    - **Attention : il faut toujours contrôler le résultat !**
      → `1<=nb<=count`
    - Pour un fichier ce n'est généralement pas le cas (`nb=count`)
    - Probable pour un tube de communication ou une connexion réseau (problème producteur/consommateur avec tampon)

Lecture/écriture dans un flot

▷ Écrire exactement le volume à transmettre

```
ssize_t /* >=0: nb written   -1: error */
writeFully(int fd,
           const void * buf,
           size_t count)
{
ssize_t r;
size_t remaining=count;
const char * ptr=(const char *)buf;
while(remaining)
  {
  RESTART_SYSCALL(r,write(fd,ptr,remaining));
  if(r<0) return -1;
  ptr+=r;
  remaining-=r;
  }
return count-remaining;
}
```

Lecture/écriture dans un flot

▷ Exemple : recopie de fichier

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define RESTART_SYSCALL(result,syscall) \
  do { (result)=(syscall); } while(((result)<0)&&(errno==EINTR));

#define MY_BUFFER_SIZE 0x400
char buffer[MY_BUFFER_SIZE];

ssize_t /* >=0: nb written   -1: error */
writeFully(int fd,const void * buf,size_t count) { /* cf page precedente */ }
```

Lecture/écriture dans un flot

```
int main(int argc,char ** argv)
{
if(argc<=2)
  {
  fprintf(stderr,"usage: %s input output\n",argv[0]);
  exit(EXIT_FAILURE);
  }
int input=open(argv[1],O_RDONLY);
if(input==-1)
  {
  fprintf(stderr,"open(%s): %s\n",argv[1],strerror(errno));
  exit(EXIT_FAILURE);
  }
int output=open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,0644);
if(output==-1)
  {
  fprintf(stderr,"open(%s): %s\n",argv[2],strerror(errno));
  close(input);
  exit(EXIT_FAILURE);
  }
```

Lecture/écriture dans un flot

```
int stop=0;
do
  {
  ssize_t nbR;
  RESTART_SYSCALL(nbR,read(input,buffer,MY_BUFFER_SIZE));
  switch(nbR)
    {
    case -1: perror("read"); stop=1; break;
    case 0: stop=1; break; // end of file
    default:
      {
      ssize_t nbW=writeFully(output,buffer,nbR);
      if(nbW!=nbR)
        {
        perror("writeFully");
        stop=1;
        }
      }
    }
  } while(!stop);
close(input); close(output); return EXIT_SUCCESS;
}
```

Positionnement dans un flot

- **L'appel système** `lseek()` (`man 2 lseek`)
  - Déplacer/consulter la position courante dans un flot de données
  - Position de la prochaine lecture/écriture
  - Position : le type `off_t` ($\simeq$ entier long)
    ```
    #include <sys/types.h>
    ```
  - ```
    #include <unistd.h>
    off_t lseek(int fd,off_t offset,int whence);
    ```
  - Se déplacer de `offset` octets dans le flot `fd` selon `whence` :
    - `SEEK_SET` : depuis le début du flot
    - `SEEK_CUR` : depuis la position courante
    - `SEEK_END` : depuis la fin du flot
  - Retour : nouvelle position (depuis le début) ou `-1` si erreur
    ```

Positionnement dans un flot

- **L'appel système `lseek()`**
  - Causes d'erreur (consulter `errno`) :
    - Mauvais arguments
    - Flot inadapté (*tube*, *socket* ...)
    - Position finale négative
  - On peut dépasser la fin !
    - On peut écrire à cette position
    - `read()` donne des `0` dans l'espace
  - Consulter la position courante : `pos=lseek(fd,0,SEEK_CUR);`
  - Aller à la fin : `pos=lseek(fd,0,SEEK_END);`
  - Aller au début : `pos=lseek(fd,0,SEEK_SET);`
  - Avancer de 8 octets : `pos=lseek(fd,8,SEEK_CUR);`
  - Reculer de 4 octets : `pos=lseek(fd,-4,SEEK_CUR);`

Duplication de descripteur

- **L'appel système** `dup()` (`man 2 dup`)
  - Attribuer un autre descripteur au même *"fichier"*
  - ```
    #include <unistd.h>
    int dup(int oldfd);
    int dup2(int oldfd, int newfd);
    ```
  - Retour : nouveau flot (`newfd` pour `dup2()`) ou `-1` si erreur
  - Causes d'erreur (consulter `errno`) :
    - Mauvais arguments
    - Trop de fichiers ouverts (très rare !)
  - `newfd` est fermé avant d'être utilisé
  - Les opérations sur l'un influent sur l'autre !
    ```

Duplication de descripteur

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

void print(const char * txt,int length) // Fonction prevue pour ecrire
{ write(STDOUT_FILENO,txt,length); }    // sur la sortie standard (console)

int main(void)
{
print("BEFORE DUP\n",11); // ecriture dans la console
int output=open("output.txt",O_WRONLY|O_CREAT|O_TRUNC,0644); // ouvrir un fichier
if(dup2(output,STDOUT_FILENO)==-1) // sortie standard --> fichier
  { perror("dup2"); return 1; }
close(output); // plus necessaire maintenant
print("AFTER DUP\n",10); // ecriture dans le fichier !
return 0;
}
```

Paramétrage des flots

- **L'appel système** `fcntl()` (`man 2 fcntl`)
  - `#include <unistd.h>`
    `#include <fcntl.h>`
    `int fcntl(int fd,int cmd,...);`
  - `fd` : le flot à paramétrer
  - `cmd` : la commande à appliquer (peut nécessiter des arguments)
  - Influer sur les paramètres d'un flot ouvert
    `F_DUPFD`, `F_GETFD`, `F_SETFD`, `F_GETFL`, `F_SETFL`
  - Vérouiller des portions de fichier
    `F_GETLK`, `F_SETLK`, `F_GETLKW`
  - Communication asynchrone
    `F_GETOWN`, `F_SETOWN`, `F_GETSIG`, `F_SETSIG`

Paramétrage des flots

- **Passage en mode non bloquant**
  - En cas d'opération ne pouvant avoir lieu immédiatement
    - Pas de bloquage du processus
    - Échec avec `errno=EAGAIN`

```
int /* 0: success   -1: error */
setBlockingFD(int fileDescriptor,
              int blocking)
{
int r,flags;
RESTART_SYSCALL(r,fcntl(fileDescriptor,F_GETFL));
if(r==-1)
  { perror("fcntl(F_GETFL)"); return -1; }
flags=(blocking ? r & ~O_NONBLOCK : r | O_NONBLOCK);
RESTART_SYSCALL(r,fcntl(fileDescriptor,F_SETFL,flags));
if(r==-1)
  { perror("fcntl(F_SETFL)"); return -1; }
return 0;
}
```

Scrutation des flots

- **Surveiller plusieurs flots simultanément**
  - Opération bloquante sur l'un → impossibilité de réagir aux autres
  - **Mauvaise** solution : boucle d'attente active → à éviter absolument !
    - Mettre les flots en mode non-bloquant
    - Si `EAGAIN` sur l'un on passe à un autre
    - Consomme inutilement du temps de calcul et de l'énergie !
  - Bonne solution : scrutation passive (`man 2 select`)
    - Préciser au système les flots à surveiller et bloquer le processus
    - Réveil lorsqu'un des flots surveillés a évolué ou après un délais
    - Systèmes conçus pour suspendre les processus en attente de ressources
      → Gestion de files d'attentes
      → Puissance de calcul pour les traitements qui en ont besoin
  - Voir le cours *Socket* du module *Réseaux*

Configuration du terminal

- **Caractéristiques en amont du flot**
  - Fonctionnement en mode ligne (canonique) ou caractère (brut)
  - Touches ou combinaisons spéciales
  - Paramètres de transmission sur les ports
  - Accessible par la structure `termios` (`man 3 termios`)
    - Fonctions de `termios.h`
  - Accessible par l'appel système `ioctl()` (`man 2 ioctl`)
    - Complexe et peu portable
    - De moins en moins utilisé
  - Accessible par la commande `stty` (`man 1 stty`)
    - Configuration externe au processus

Configuration du terminal

▷ Réaction immédiate à la saisie

```
#include <termios.h>
#include <unistd.h>

struct termios savedTermios; /* pour la restauration */

int initTerminal(void)
{
struct termios newTermios;
int r;
RESTART_SYSCALL(r,tcgetattr(STDIN_FILENO,&savedTermios));
if(r<0) { perror("tcgetattr"); return -1; }
newTermios=savedTermios;
newTermios.c_lflag&=~ICANON; /* desactiver le mode canonique          */
newTermios.c_cc[VMIN]=1;     /* des qu'un seul caractere est saisi ... */
newTermios.c_cc[VTIME]=0;    /* ... l'envoyer immediatement           */
RESTART_SYSCALL(r,tcsetattr(STDIN_FILENO,TCSANOW,&newTermios));
if(r<0) { perror("tcsetattr"); return -1; }
return 0;
}
```

Configuration du terminal

▷ Restauration du terminal

- ◇ Remettre le terminal dans un état compatible avec la saisie de commandes
- ◇ *via* `atexit()` (`man 3 atexit`)
  (terminaison normale du processus)
- ◇ *via* `sigaction()` (`man 2 sigaction`)
  (terminaison par un signal)

```
/* struct termios savedTermios; */

int restoreTerminal(void)
{
int r;
RESTART_SYSCALL(r,tcsetattr(STDIN_FILENO,TCSANOW,&savedTermios));
if(r<0) { perror("tcsetattr"); return -1; }
return 0;
}
```

Configuration du terminal

```
#include <stdio.h>

int main(void)
{
int c;
do
  {
  c=fgetc(stdin);
  fprintf(stdout,"-->%.8x [%c]\n",c,c);
  } while(c!=EOF);
return 0;
}

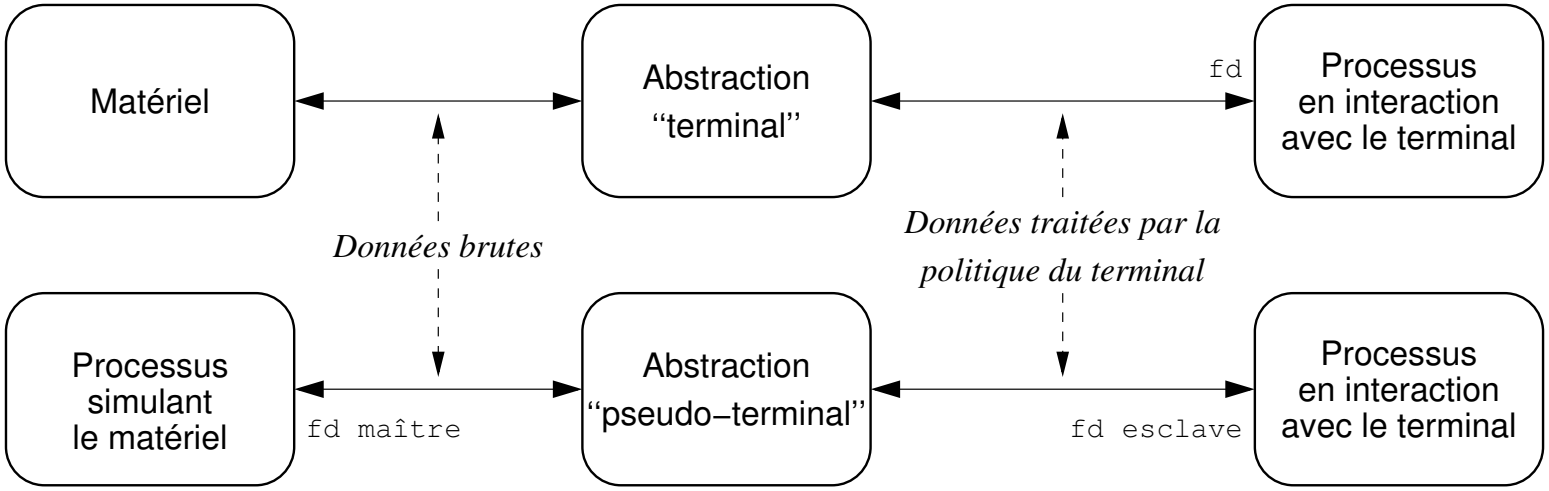
$ echo "abcd" | ./prog
-->00000061 [a]
-->00000062 [b]
-->00000063 [c]
-->00000064 [d]
-->0000000a [
]
-->ffffffff [ÿ]
```

```
$ ./prog
a
-->00000061 [a]
-->0000000a [
]
bcd
-->00000062 [b]
-->00000063 [c]
-->00000064 [d]
-->0000000a [
]
^D-->ffffffff [ÿ]
$ stty -icanon
$ ./prog
a-->00000061 [a]
b-->00000062 [b]
c-->00000063 [c]
d-->00000064 [d]
```

Pseudo-terminaux

▷ Simuler par logiciel un terminal réel

- ◇ Interagir avec des programmes qui veulent absolument un terminal (`ssh`, `su` ...)
- ◇ Descripteur *"esclave"* pour interagir avec le pseudo-terminal (aucune différence avec le cas d'un terminal réel)
- ◇ Descripteur *"maître"* pour simuler le comportement du matériel (remplace la liaison physique)



Pseudo-terminaux

▷ Du côté *"maître"*

```
#include <stdlib.h>

int r,masterFd;
RESTART_SYSCALL(masterFd,open("/dev/ptmx",O_RDWR|O_NOCTTY));
if(masterFd==-1) { perror("open master"); return -1; }
RESTART_SYSCALL(r,grantpt(masterFd));
if(r==-1) { perror("grantpt"); return -1; }
RESTART_SYSCALL(r,unlockpt(masterFd));
if(r==-1) { perror("unlockpt"); return -1; }
char * ttyName=ptsname(masterFd); // nom du terminal esclave
if(!ttyName) { perror("ptsname"); return -1; }
```

▷ Du côté *"esclave"*

```
int slaveFd;
RESTART_SYSCALL(slaveFd,open(ttyName,O_RDWR));
if(slaveFd==-1) { perror("open slave"); return -1; }
```

Attributs des fichiers

- **L'appel système** `lstat()` (`man 2 lstat`)
  - ```
    #include <sys/types.h>
    #include <sys/stat.h>
    #include <unistd.h>
    int lstat(const char * path,struct stat * infos);
    ```
  - Écrit dans la structure **`infos`** selon les attributs du fichier **`path`**
  - Informations de diverses nature :
    Type de fichier, droits d'accès, taille, propriétaire, groupe, nombre de références, dates d'accès, de modification des attributs et du contenu
    ...
  - Retour : `0` si ok, `-1` si erreur
  - Causes d'erreur (consulter **`errno`**) :
    - Fichier inexistant, droits insuffisants ...
    ```

Attributs des fichiers

- **Le champ `st_mode` de la structure `stat`**
  - De type `mode_t` (voir appel système `open()`)
  - Lecture des droits : et bit à bit avec les constantes de `mode_t`
  - Décodage du type de fichier à l'aide de macros :
    - `S_ISBLK(infos->st_mode)` : périphérique en mode bloc
    - `S_ISCHR(infos->st_mode)` : périphérique en mode caractère
    - `S_ISDIR(infos->st_mode)` : répertoire
    - `S_ISFIFO(infos->st_mode)` : *tube* de communication
    - `S_ISLNK(infos->st_mode)` : lien symbolique
    - `S_ISREG(infos->st_mode)` : fichier régulier
    - `S_ISSOCK(infos->st_mode)` : *socket*

Attributs des fichiers

- **L'appel système** `stat()` (`man 2 stat`)
  - Exactement comme `lstat()` mais suit les liens symboliques
  - `S_ISLNK(infos->st_mode)` est inutile
- **L'appel système** `fstat()` (`man 2 fstat`)
  - Exactement comme `stat()` mais sur un descripteur de fichier
  - `int fstat(int fd,struct stat * infos);`
  - On peut ouvrir un répertoire en lecture avec `open()`

Attributs des fichiers

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

int main(void)
{
struct stat infos;
if(fstat(STDIN_FILENO,&infos)!=-1)
  {
  if(S_ISBLK(infos.st_mode))       fprintf(stderr,"BLOCK DEVICE\n");
  else if(S_ISCHR(infos.st_mode))  fprintf(stderr,"CHARACTER DEVICE\n");
  else if(S_ISDIR(infos.st_mode))  fprintf(stderr,"DIRECTORY\n");
  else if(S_ISFIFO(infos.st_mode)) fprintf(stderr,"FIFO\n");
  else if(S_ISLNK(infos.st_mode))  fprintf(stderr,"SYMBOLIC LINK\n");
  else if(S_ISREG(infos.st_mode))  fprintf(stderr,"REGULAR FILE (%ld)\n",infos.st_size);
  else if(S_ISSOCK(infos.st_mode)) fprintf(stderr,"SOCKET\n");
  else fprintf(stderr,"???\n");
  }
else perror("fstat");
return 0;
}
```

```
$ ./prog
CHARACTER DEVICE
$ ./prog < /dev/console
CHARACTER DEVICE
$ ./prog < /dev/hda
BLOCK DEVICE
```

```
$ ./prog < file.txt
REGULAR FILE (12)
$ ./prog < /tmp
DIRECTORY
$ ./prog < /tmp/xmms_harrouet.0
SOCKET
$ ls | ./prog
FIFO
```

Attributs des fichiers

- **Lecture et modification de la taille** (`man 2 ftruncate`)
  - Champ `st_size` de la structure `stat`
    - Voir les appels `stat()`, `fstat()`, `lstat()`
    - Type `off_t`
    - Taille en octets d'un fichier, lien symbolique ou répertoire
  - ```
    #include <unistd.h>
    int ftruncate(int fd,off_t length);
    ```
  - Réduire (ou alonger) à `length` la taille du fichier désigné par `fd`
  - Évite la recopie et le renommage
  - Retour : `0` si ok, `-1` si erreur (consulter `errno`)
    ```

Attributs des fichiers

- **Modification des droits d'accès** (`man 2 chmod`)
  - ```
    #include <sys/types.h>
    #include <sys/stat.h>
    int chmod(const char * path,mode_t mode);
    int fchmod(int fd,mode_t mode);
    ```
  - Attribuer les droits `mode` au fichier
    - Désigné par le chemin `path`
    - Associé au descripteur `fd`
  - Voir le paramètre `mode` de l'appel système `open()`
  - Retour : `0` si ok, `-1` si erreur (consulter `errno`)
    ```

Attributs des fichiers

- **Modification de la propriété** (`man 2 chown`)
  - ```
    #include <sys/types.h>
    #include <unistd.h>
    int chown(const char * path,uid_t user,gid_t group);
    int fchown(int fd,uid_t user,gid_t group);
    int lchown(const char * path,uid_t user,gid_t group);
    ```
  - Modifier le propriétaire et le groupe du fichier
    - Désigné par le chemin `path`
    - Associé au descripteur `fd`
  - Le lien symbolique lui-même avec `lchown()`
  - Retour : `0` si ok, `-1` si erreur (consulter `errno`)
  - Voir `man 3 getpwnam/getgrnam` pour `user` et `group`
    ```

Parcours des répertoires

- **Parcours séquentiel (principe)**
  - Ouverture du répertoire par `opendir()` → structure `DIR`
  - Lecture des entrées sucessives par `readdir()` → structure `dirent`
  - Lecture du nom de l'entrée par le champ `d_name`
  - Retour éventuel à la première entrée par `rewinddir()` (seule fonctionnalité de déplacement *POSIX*)
  - Fermeture du répertoire par `closedir()`

Parcours des répertoires

- **La fonction** `opendir()` (`man 3 opendir`)
  - `#include <sys/types.h>`
    `#include <dirent.h>`
    `DIR * opendir(const char * path);`
  - Crée la structure `DIR` associée au répertoire désigné par `path`
  - Retour : adresse de cette structure ou pointeur nul si erreur
  - Causes d'erreur (consulter `errno`) :
    - Répertoire inexistant
    - Droits insuffisants
    - ...

Parcours des répertoires

- **La fonction** `readdir()` (`man 3 readdir`)
  - `#include <sys/types.h>`
    `#include <dirent.h>`
    `struct dirent * readdir(DIR * dir);`
  - Initialise dans `dir` la structure `dirent` décrivant l'entrée suivante
    - Seul champ *POSIX* : `char d_name[]`
    - Ne pas libérer cette structure
    - Contenu écrasé par le prochain `readdir()`
  - `.` et `..` toujours présents mais pas forcément indiqués !
  - Retour :
    - Adresse de cette structure si ok
    - Pointeur nul si erreur ou dernière entrée dépassée
  - Causes d'erreur (consulter `errno`) : `dir` incorrect

Parcours des répertoires

- **La fonction** `rewinddir()` (`man 3 rewinddir`)
  - ```
    #include <sys/types.h>
    #include <dirent.h>
    void rewinddir(DIR * dir);
    ```
  - Repositionne la lecture du répertoire au début
  - Les fonctions `seekdir()` et `telldir()` ne sont pas *POSIX*
- **La fonction** `closedir()` (`man 3 closedir`)
  - ```
    #include <sys/types.h>
    #include <dirent.h>
    int closedir(DIR * dir);
    ```
  - Ferme le répertoire et libère la structure `dir`
  - Retour : `0` si ok, `-1` si erreur (`dir` incorrect)

Parcours des répertoires

- **Parcours plus élaborés**
  - La fonction `scandir()` (`man 3 scandir`)
    - Rechercher les entrées d'un répertoire
    - Un pointeur de fonction pour inclure/exclure une entrée
    - Un pointeur de fonction pour trier les entrées
    - Fonction non *POSIX* (*BSD*)
  - La fonction `ftw()` (`man 3 ftw`)
    - Parcours récursif d'une arborescence
    - Un pointeur de fonction à invoquer sur chaque entrée
    - Une profondeur limitée
    - Fonction non *POSIX* (*System V*)
  - La fonction `glob()` (`man 3 glob`)
    - Utilisation de *"Jokers"* (`*.c` ...)

Parcours des répertoires

▷ Déterminer le répertoire courant

- ◇ Point de départ des chemins relatifs
- ◇ La fonction `getcwd()` (`man 3 getcwd`)

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

- ◇ Stocker dans `buf` le chemin désignant le répertoire courant
- ◇ Écriture de `size` octets maxi (y compris `'\0'`)
- ◇ `buf` doit être préalablement alloué à `size` octets minimum
- ◇ Retour : `buf` si ok, pointeur nul si erreur
- ◇ Causes d'erreur (consulter `errno`) :
  - ○ `ERANGE` → `size` insuffisant
  - ○ Droits d'accès ...

Parcours des répertoires

- **Changer le répertoire courant**
  - Les fonctions `chdir()` et `fchdir()` (`man 2 chdir` / `fchdir`)

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

  - Le point de départ des chemins relatifs devient `path` ou `fd`
  - `path` ou `fd` doivent désigner un répertoire
  - Retour : `0` si ok, `-1` si erreur
  - Causes d'erreur (consulter `errno`) :
    - Mauvais arguments, droits d'accès ...

Parcours des répertoires

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

#include <stdio.h>
#include <string.h>

int main(int argc,char ** argv)
{
char buffer[0x100];
const char * path=(argc>1 ? argv[1] : ".");
DIR * dir=opendir(path);
if(!dir)
  {
  fprintf(stderr,"opendir(%s): %s\n",path,strerror(errno));
  return 1;
  }
```

Parcours des répertoires

```
struct dirent * entry=readdir(dir);
while(entry)
  {
  struct stat infos;
  sprintf(buffer,"%s/%s",path,entry->d_name);
  if(lstat(buffer,&infos)!=-1) fprintf(stderr,"lstat(%s): %s\n",buffer,strerror(errno));
  else
    {
    if(S_ISBLK(infos.st_mode))       printf("%s: BLOCK DEVICE\n",buffer);
    else if(S_ISCHR(infos.st_mode))  printf("%s: CHARACTER DEVICE\n",buffer);
    else if(S_ISDIR(infos.st_mode))  printf("%s: DIRECTORY\n",buffer);
    else if(S_ISFIFO(infos.st_mode)) printf("%s: FIFO\n",buffer);
    else if(S_ISLNK(infos.st_mode))  printf("%s: SYMBOLIC LINK\n",buffer);
    else if(S_ISREG(infos.st_mode))  printf("%s: REGULAR FILE (%ld)\n",
                                            buffer,infos.st_size);
    else if(S_ISSOCK(infos.st_mode)) printf("%s: SOCKET\n",buffer);
    else printf("%s: ???\n",buffer);
    }
  entry=readdir(dir);
  }
closedir(dir);
return 0;
}
```

Références sur les fichiers

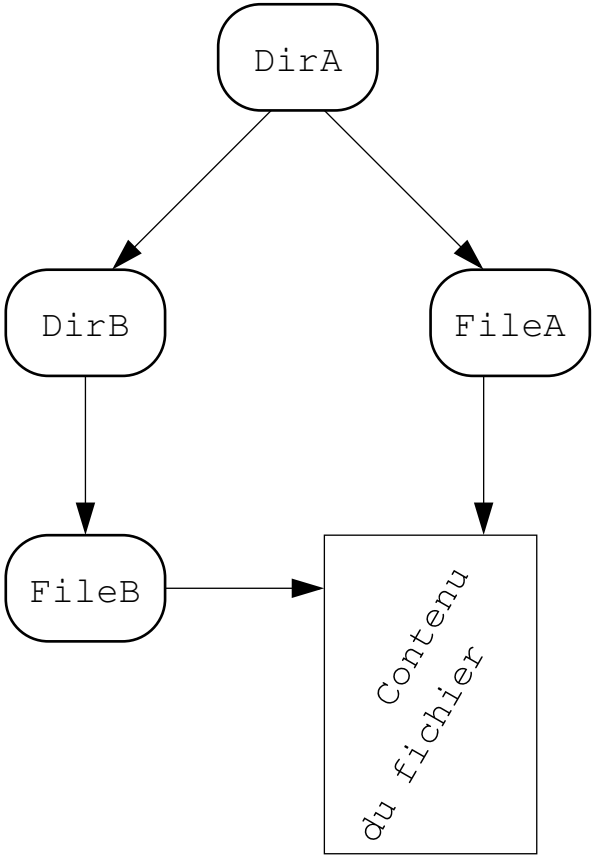
- **Ajout d'un lien physique** (`man 2 link`)
  - ```
    #include <unistd.h>
    int link(const char * oldPath,
             const char * newPath);
    ```
  - Le fichier `oldPath` est également désigné par `newPath`
    - `newPath` de doit pas déjà exister
    - Le champ `st_nlink` de la structure `stat` compte les références
    - La destruction de l'un d'eux n'entraine pas la destruction du fichier
    - Rester dans le même système de fichiers
  - Retour : `0` si ok, `-1` si erreur (consulter `errno`)
    ```

Références sur les fichiers

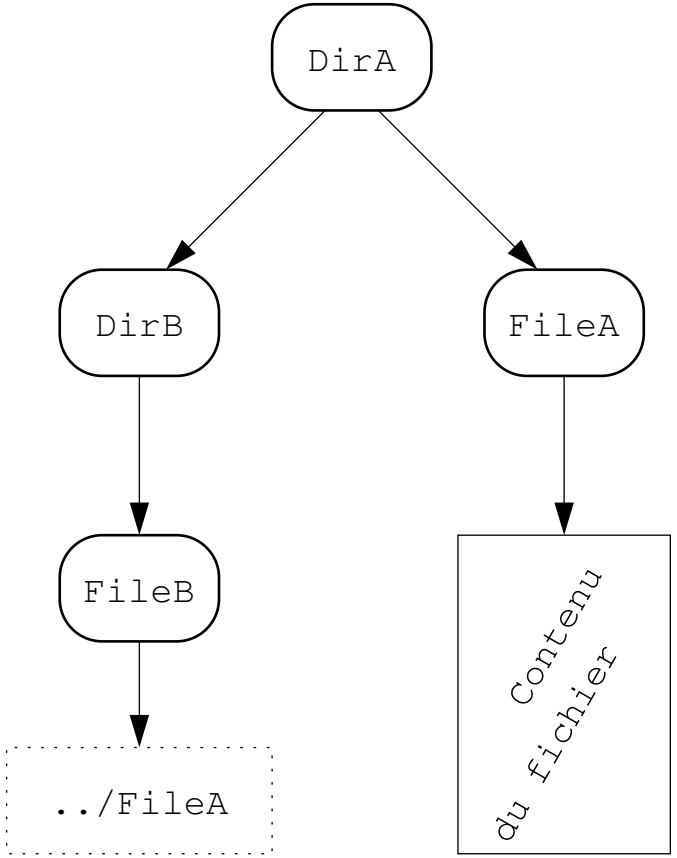
- **Ajout d'un lien symbolique** (`man 2 symlink`)
  - ```
    #include <unistd.h>
    int symlink(const char * oldPath,
                const char * newPath);
    ```
  - Le fichier `oldPath` est également désigné par `newPath`
    - `newPath` de doit pas déjà exister
    - Le fichier n'existe que sous le nom `oldPath`
    - `newPath` ne contient que **le nom** `oldPath`
  - Retour : `0` si ok, `-1` si erreur (consulter `errno`)
    ```

Références sur les fichiers

```
link("../FileA","FileB");
```




```
symlink("../FileA","FileB");
```



Références sur les fichiers

- **Création d'un répertoire** (`man 2 mkdir`)
  - `#include <sys/stat.h>`
    `#include <sys/types.h>`
    `int mkdir(const char * path,mode_t mode);`
  - Crée le répertoire `path` avec les droits `mode`
    - Il ne doit pas exister
    - Voir l'argument `mode` de `open()`
  - Retour : `0` si ok, `-1` si erreur (consulter `errno`)
- **Suppression d'un répertoire** (`man 2 rmdir`)
  - `#include <unistd.h>`
    `int rmdir(const char * path);`
  - Supprime le répertoire `path` (doit être vide)
  - Retour : `0` si ok, `-1` si erreur (consulter `errno`)

Références sur les fichiers

- **Suppression d'une référence** (`man 2 unlink`)
  - `#include <unistd.h>`
    `int unlink(const char * path);`
  - Supprime la référence sur le fichier `path`
  - Le contenu du fichier peut toujours exister après
    - Autres liens dans le système de fichier
    - `open()` référence et `close()` déréférence
    - `0` références → destruction
  - Retour : `0` si ok, `-1` si erreur (consulter `errno`)
- **La fonction** `remove()` (`man 3 remove`)
  - `#include <stdio.h>`
    `int remove(const char * path);`
  - Appelle `unlink()` ou `rmdir()` selon la nature de `path`

Références sur les fichiers

- **Déplacement/renommage** (`man 2 rename`)
  - ```
    #include <stdio.h>
    int rename(const char * oldPath,
               const char * newPath);
    ```
  - Remplace la référence `oldPath` par `newPath`
    - Écrase la référence `newPath` si elle existe
    - Si `newPath` est un répertoire, il doit être vide
    - Rester dans le même système de fichiers
  - Retour : `0` si ok, `-1` si erreur (consulter `errno`)
    ```

Les tubes de communication

- **Un tuyau unidirectionnel**
  - Un côté pour écrire, un côté pour lire, accédés par des descripteurs
    - Unidirectionnel : `O_RDONLY` ou `O_WRONLY` mais pas `O_RDWR`
  - Utilisés par des processus distincts pour communiquer
    - Flot continu de données
  - Utilisation d'un tampon interne de capacité limitée
    - Écriture bloquante si tampon plein (producteur/consommateur)
  - Plusieurs lecteurs et plusieurs écrivains possibles
    - Écriture alors que tous les descripteurs du côté lecture sont fermés
      → erreur `EPIPE` et signal `SIGPIPE` (`man 2 sigaction`)
    - Pour obtenir la *fin de fichier* en lecture :
      → le tampon doit être vide et
      → **tous** les descripteurs du côté **écriture** doivent être **fermés** !

Les tubes de communication

▷ Les tubes anonymes

◇ Appel système `pipe()` (`man 2 pipe`)

```
#include <unistd.h>
int pipe(int fd[2]);
```

◇ `fd` : deux descripteurs qui sont initialisés par l'appel

- ○ `fd[0]` pour lire, `fd[1]` pour écrire

◇ Retour : `0` si ok `-1` si erreur (consulter `errno`)

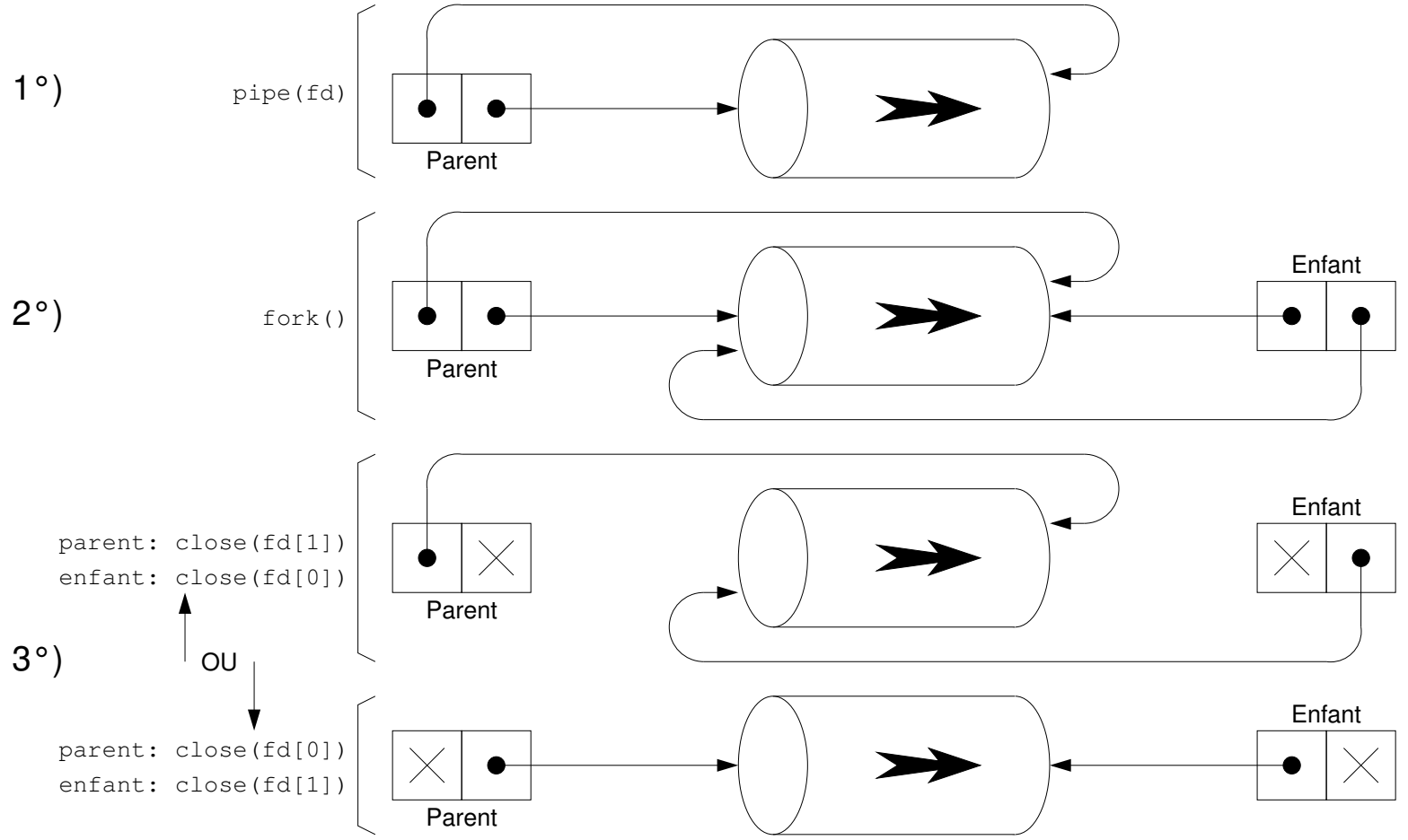
◇ On peut utiliser `fd[0]`, `fd[1]` ou les deux

◇ Généralement on partage le tube avec un processus enfant

- ○ Créer le tube puis le processus enfant
- ○ L'enfant conserve une extrémité du tube et ferme l'autre
- ○ Le parent fait l'inverse → flot unidirectionnel
- ○ Voir le cours *Socket* du module *Réseaux*

Les tubes de communication

▷ Les tubes anonymes : parent ← enfant ou parent → enfant



Les tubes de communication

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
#define RESTART_SYSCALL(result,syscall) /* ... */
ssize_t writeFully(int fd,const void * buf,size_t count) { /* ... */ }

void copyFromTo(int fromFd,int toFd)
{
char buffer[0x100];
for(;;)
  {
  ssize_t nbR; RESTART_SYSCALL(nbR,read(fromFd,buffer,0x100));
  switch(nbR)
    {
    case -1: perror("read"); return;
    case 0: return; /* end of file */
    default: if(writeFully(toFd,buffer,nbR)==-1) { perror("writeFully"); return ; }
    }
  }
}
```

Les tubes de communication

```
int main(void) /* communication enfant --> parent */
{
pid_t child; int r,fd[2];
RESTART_SYSCALL(r,pipe(fd)); /* creer le tube */
if(r==-1) { perror("pipe"); return -1; }
RESTART_SYSCALL(child,fork()); /* creer un processus enfant */
switch(child)
  {
  case -1: perror("fork"); return -1;
  case 0: /*---- dans le processus enfant ----*/
    RESTART_SYSCALL(r,close(fd[0]));       /* pas besoin de lire depuis le tube */
    writeFully(fd[1],"Hello Parent\n",13); /* ecriture dans le tube */
    RESTART_SYSCALL(r,close(fd[1]));       /* fini d'ecrire dans le tube */
    exit(0); /* terminaison du processus enfant */
  default: /*---- dans le processus parent ----*/
    RESTART_SYSCALL(r,close(fd[1]));       /* pas besoin d'ecrire dans le tube */
    copyFromTo(fd[0],STDOUT_FILENO);       /* lecture depuis le tube */
    RESTART_SYSCALL(r,close(fd[0]));       /* fini de lire depuis le tube */
    RESTART_SYSCALL(r,waitpid(child,(int *)0,0)); /* attente du processus enfant */
    if(r==-1) { perror("waitpid"); return -1; }
  }
return 0;
}
```

Les tubes de communication

```
int main(void) /* communication parent --> enfant */
{
pid_t child; int r,fd[2];
RESTART_SYSCALL(r,pipe(fd)); /* creer le tube */
if(r==-1) { perror("pipe"); return -1; }
RESTART_SYSCALL(child,fork()); /* creer un processus enfant */
switch(child)
  {
  case -1: perror("fork"); return -1;
  case 0: /*---- dans le processus enfant ----*/
    RESTART_SYSCALL(r,close(fd[1]));      /* pas besoin d'ecrire dans le tube */
    copyFromTo(fd[0],STDOUT_FILENO);      /* lecture depuis le tube */
    RESTART_SYSCALL(r,close(fd[0]));      /* fini de lire depuis le tube */
    exit(0); /* terminaison du processus enfant */
  default: /*---- dans le processus parent ----*/
    RESTART_SYSCALL(r,close(fd[0]));      /* pas besoin de lire depuis le tube */
    writeFully(fd[1],"Hello Child\n",12); /* ecriture dans le tube */
    RESTART_SYSCALL(r,close(fd[1]));      /* fini d'ecrire dans le tube */
    RESTART_SYSCALL(r,waitpid(child,(int *)0,0)); /* attente du processus enfant */
    if(r==-1) { perror("waitpid"); return -1; }
  }
return 0;
}
```

Les tubes de communication

▷ Les tubes nommés

- ◇ Tubes accessibles depuis le système de fichier
- ◇ Fonction `mkfifo()` (`man 3 mkfifo`)

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char * path,mode_t mode);
```

- ◇ `path` : emplacement du tube dans le système de fichiers
- ◇ `mode` : droits d'accès au tube (voir appel système `open()`)
- ◇ Retour : `0` si ok `-1` si erreur (consulter `errno`)
- ◇ Permettent une communication entre des processus sans filiation
  - ◦ Ils doivent avoir le chemin et les droits d'accès au tube
  - ◦ Ouverture comme un simple fichier (`O_RDONLY` ou `O_WRONLY` mais pas `O_RDWR`)

Les tubes de communication

▷ Les tubes nommés

- ◇ Dialoguer avec un processus utilisant des fichiers
  - ○ Créer le(s) tube(s) à la place du/des fichier(s)
  - ○ Lancer un processus qui le(s) contrôle
  - ○ Lancer le processus qui croit utiliser un/des fichier(s)
  - ○ L'illusion s'arrête là :
    - → après ouverture c'est un tube semblable aux tubes anonymes
    - → toute la communication a lieu en mémoire (pas sur le disque !)
    - → on ne peut pas faire de `lseek()`
- ◇ `mkfifo` est également une commande

```
$ mkfifo monTube
$ echo "Coucou" > monTube
$ cat > monTube
ABCD
Hello
^D
$ rm monTube
```

```
$ cat monTube
Coucou
$ cat monTube
ABCD
Hello
$
```

Les flux

- *API* **de haut niveau pour les entrées/sorties**
  - Bibliothèque de fonctions déclarées dans `stdio.h`
  - Type *opaque* `FILE` manipulé exclusivement par pointeur
  - Dissimuler les subtilités des appels systèmes
  - Améliorer la portabilité des programmes
    - Fait partie de la norme *ANSI* du langage *C*
  - Puissantes fonctionnalités de formatage des données
  - Inconnu du système (espace utilisateur)
  - Compatible avec les opérations *"bas-niveau"*

Les flux

- ▷ *API* **de haut niveau pour les entrées/sorties**
  - ◇ Tampons internes → moins d'appels au système (plus efficace)
    - ○ Fonction `setvbuf()` (`man 3 setvbuf`)
    - ○ Tampon complet, vidé quand il est plein
    - ○ Tampon par ligne, `'\n'` → vidange (pour les terminaux)
  - ◇ Flux prédéfinis :
    - ○ Variables globales de type `FILE *`
    - ○ `stdin` : sur `STDIN_FILENO`
    - ○ `stdout` : sur `STDOUT_FILENO`, tampon par ligne
    - ○ `stderr` : sur `STDERR_FILENO`, sans tampon)

Ouverture/fermeture d'un flux

▷ Ouverture sur un fichier

- ◇ Fonction `fopen()` (`man 3 fopen`)
  `FILE * fopen(const char * path,const char * mode);`
- ◇ Ouvrir le fichier `path` selon `mode` :
  - ○ `"r"` : lecture depuis le début
  - ○ `"r+"` : lecture/écriture depuis le début
  - ○ `"w"` : écriture (fichier vidé ou créé)
  - ○ `"w+"` : lecture/écriture (fichier vidé ou créé)
  - ○ `"a"` : écriture à partir de la fin (fichier complété ou créé)
  - ○ `"a+"` : lecture/écriture à partir de la fin (fichier complété ou créé)
  - ○ `"b"` peut être ajouté pour indiquer un fichier binaire (conversion des fins de lignes, indispensable sous `M$DO$`)

Ouverture/fermeture d'un flux

- **Ouverture sur un fichier**
  - Retour : pointeur sur un nouveau `FILE` ou pointeur nul si erreur
  - Causes d'erreur (consulter `errno`) :
    - Mauvais mode
    - Erreurs de `open()` et `fcntl()`
- **Ouverture sur un descripteur donné**
  - Fonction `fdopen()` (`man 3 fdopen`)
    ```
    FILE * fdopen(int fd,const char * mode);
    ```
  - Le `mode` doit être compatible avec les propriétés du flot déjà ouvert
  - Commence à la position courante du flot
  - Mêmes retour et erreurs que `fopen()`
  - Utile pour simplifier la communication sur *tubes*, *sockets* ...

Ouverture/fermeture d'un flux

▷ **Réouverture sur un fichier**

◇ Fonction `freopen()` (`man 3 freopen`)

```
FILE * freopen(const char * path,const char * mode,
               FILE * stream);
```

◇ Comme `fopen()` mais on remplace un flux existant

- fermeture du flux `stream`
- Mise à jour selon `path` et `mode`

◇ Mêmes retour et erreurs que `fopen()`

◇ Le reste de l'application continue à utiliser le flux `stream`

◇ Limité à un fichier → `dup2()` sur le descripteur est plus général

Réouverture d'un flux sur un fichier

```
#include <stdio.h>

void print(const char * txt) // Fonction prevue pour ecrire
{                            // sur la sortie standard
fprintf(stdout,"%s",txt);    // (la console)
}

int main(void)
{
print("BEFORE FREOPEN\n"); // ecriture dans la console
if(!freopen("output.txt","w",stdout)) // sortie standard --> fichier
  {
  perror("freopen()");
  return 1;
  }
print("AFTER FREOPEN\n"); // ecriture dans le fichier !
return 0;
}
```

Ouverture/fermeture d'un flux

▷ Vidange d'un flux

- ◇ Fonction `fflux()` (`man 3 fflush`)
  `int fflush(FILE * stream);`
- ◇ Force l'écriture du tampon du flux `stream` (tous si pointeur nul)
- ◇ Ne synchronise pas le flot associé !
- ◇ Retour : `0` si ok, constante `EOF` si erreur
- ◇ Erreurs : comme `write()`

▷ Fermeture d'un flux

- ◇ Fonction `fclose()` (`man 3 fclose`)
  `int fclose(FILE * stream);`
- ◇ Ferme le descripteur associé et détruit la structure
- ◇ Retour : `0` si ok, constante `EOF` si erreur
- ◇ Erreurs : comme `fflush()` et `close()`

Opération sur les flux

- **Les erreurs**
  - `int ferror(FILE * stream);` (`man 3 ferror`)
    - Résultat non nul si un erreur est survenue sur `stream`
  - `void clearerr(FILE * stream);` (`man 3 clearerr`)
    - Efface l'indicateur d'erreur sur le flux `stream`
- **La fin de fichier**
  - `int feof(FILE * stream);` (`man 3 feof`)
    - Résultat non nul si la fin de fichier a été rencontrée sur `stream`
    - Seulement après lecture de `EOF`
- **Le descripteur de fichier**
  - `int fileno(FILE * stream);` (`man 3 fileno`)
    - Le descripteur de fichier associé à un flux
    - Utilisable par les appels système (cohérence !)

Lecture/écriture dans un flux

▷ Lecture de données

- ◇ Fonction `fread()` (`man 3 fread`)

```
size_t fread(const void * buf,size_t size,
             size_t nb,FILE * stream);
```

- ◇ Lit depuis `stream` `nb` éléments de taille `size`
- ◇ Les stocke dans `buf` devant être alloué au préalable
- ◇ Retour : Le nombre d'éléments correctement lus

▷ Écriture de données

- ◇ Fonction `fwrite()` (`man 3 fwrite`)

```
size_t fwrite(const void * buf,size_t size,
              size_t nb,FILE * stream);
```

- ◇ Écrit dans `stream` le bloc `buf` contenant `nb` éléments de taille `size`
- ◇ Retour : Le nombre d'éléments correctement écrits

Lecture/écriture dans un flux

- **Lecture formatée (texte)**
  - Fonction `fscanf()` (`man 3 fscanf`)
    `int fscanf(FILE * stream,const char * format,...);`
  - Extraire depuis `stream` selon les indications de `format`
    - Correspondance entre les symboles `%` et les arguments variables
    - Arguments passés par **adresse** pour leur mise à jour
    - Documenté dans de nombreux ouvrages sur le langage *C*
    - `"%d"` (entier), `"%g"` (réel) ...
  - Retour :
    - Le nombre de conversions réussies (`%`)
    - Constante `EOF` si pb de lecture (fin de fichier ou erreur) avant la première conversion

Lecture/écriture dans un flux

▷ Lecture d'une chaîne (texte)

- ◇ Fonction `fgets()` (`man 3 fgets`)
  `char * fgets(char * buf,int size,FILE * stream);`
- ◇ Extrait depuis `stream` une ligne de texte et la place dans `buf`
- ◇ Lecture jusqu'à `'\n'` ou la fin de fichier (`'\n'` est inscrit)
- ◇ Le caractère `'\0'` est ajouté à la fin de la chaîne
- ◇ Lecture de `size-1` caractères maximum
- ◇ `buf` doit être alloué à `size` octets minimum
- ◇ Retour : `buf` si ok, pointeur nul si erreur ou fin de fichier et rien lu
- ◇ Ne **jamais** utiliser la fonction
  `char * gets(char * buf);`
  Pas de limite à la saisie → débordement possible

Lecture/écriture dans un flux

▷ Lecture d'un caractère (texte ou donnée)

- ◇ Fonction `fgetc()` (`man 3 fgetc`)
  `int fgetc(FILE * stream);`
- ◇ Extrait un caractère depuis `stream`
- ◇ Retour : Caractère extrait ou `EOF` en cas d'erreur ou fin de fichier
- ◇ Le résultat est un `int` et non un `char` !
  - ○ Caractères valables [0;255] → `EOF` est codé sur plus d'un octet
  - ○ Convention : un caractère unique est transmis dans un `int`
- ◇ Macro `getc()` (`man 3 getc`)
  `int getc(FILE * stream);`
- ◇ Exactement comme `fgetc()` mais sous forme de macro
  - ○ Plus efficace
  - ○ Évaluations multiples de l'argument → effets de bord !

Lecture/écriture dans un flux

- **Réinjection d'un caractère (texte ou donnée)**
  - Fonction `ungetc()` (`man 3 ungetc`)
    `int ungetc(int c,FILE * stream);`
  - Remet le caractère `c` (normalement le dernier extrait) dans `stream`
  - La prochaine lecture fournira `c`
  - Ne fonctionne qu'une fois de suite
  - Retour : Caractère réinjecté ou `EOF` en cas d'erreur
  - Exemple d'utilisation : analyse lexicale
    - Fin d'un lexème → lecture caractère suivant
    - Premier caractère du lexème suivant
    - Dans `123<=i` , `<` marque la fin de `123` et fait partie de `<=`
      → le replacer dans le flot

Lecture/écriture dans un flux

▷ Écriture formatée (texte)

- ◇ Fonction `fprintf()` (`man 3 fprintf`)
  ```
  int fprintf(FILE * stream,const char * format,...);
  ```
- ◇ Écrit dans `stream` selon les indications de `format`
  - ○ Correspondance entre les symboles `%` et les arguments variables
  - ○ Documenté dans de nombreux ouvrages sur le langage *C*
  - ○ `"%d"` (entier), `"%g"` (réel), `"%s"` (chaîne) ...
- ◇ Retour : Le nombre de caractères écrits

▷ Écriture d'une chaîne (texte)

- ◇ Fonction `fputs()` (`man 3 fputs`)
  ```
  int fputs(const char * str,FILE * stream);
  ```
- ◇ Écrit la chaîne `str` dans `stream` sans `'\0'`
- ◇ Retour : non négatif si ok, `EOF` en cas d'erreur

Lecture/écriture dans un flux

- **Écriture d'un caractère (texte ou donnée)**
  - Fonction `fputc()` (`man 3 fputc`)
    `int fputc(int c,FILE * stream);`
  - Écrit le caractère `c` dans `stream`
  - Caractère passé dans un `int`
  - Retour : Caractère transmis ou `EOF` en cas d'erreur
  - Macro `putc()` (`man 3 putc`)
    `int putc(int c,FILE * stream);`
  - Exactement comme `fputc()` mais sous forme de macro
    - Plus efficace
    - Évaluations multiples des arguments → effets de bord !

Projection d'un fichier en mémoire

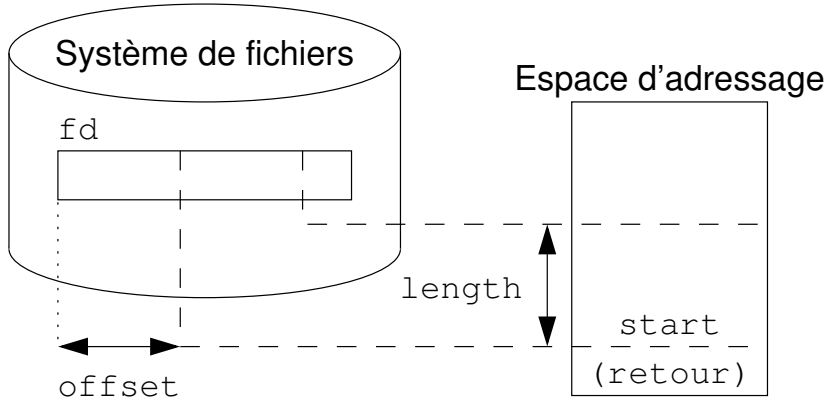
- **Principe**
  - Accéder au contenu d'un fichier comme à un simple bloc mémoire
    - Accès aléatoires par simples pointeurs
  - Pas d'allocation préalable
    - Le système charge/décharge les pages de manière transparente
    - Volume de données > mémoire du système
  - Bien plus rapide qu'une lecture/écriture classique
  - Moyen de communication
    - Synchronisation avec les projections d'autres processus
    - Synchronisation avec le fichier physique
  - Projection partagée avec les processus fils
  - Fichier physique référencé tout au long de la projection

Projection d'un fichier en mémoire

- **L'appel système** `mmap()` (`man 2 mmap`)
  - ```
    #include <unistd.h>
    #include <sys/mman.h>
    void * mmap(void * start,size_t length,int prot,
                int flags,int fd,off_t offset);
    ```
  - Projette en mémoire le contenu du fichier désigné par `fd`
  - `start` : adresse désirée pour la projection
    (limite de page, géneralement pointeur nul → choix du système)
  - `offset` : début du fichier à ignorer
  - `length` : taille de la quantité à projeter
  - `prot` : protection de la zone mémoire (ou bit à bit)
    - `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`
    - Mode d'ouverture de `fd` !
    - Dépendant du système
    ```

Projection d'un fichier en mémoire

- **L'appel système** `mmap()`
  - `flags` : options (ou binaire)
    - `MAP_FIXED` : échec si `start` non respecté
    - `MAP_SHARED` : Synchronisation entre les processus et le fichier
    - `MAP_PRIVATE` : copie privée de la projection (taille limitée)
    - `MAP_SHARED` ou `MAP_PRIVATE` obligatoire
  - Retour : Adresse de la projection ou `MAP_FAILED` si erreur (consulter `errno` : `fd` incompatible, mauvais arguments ...)



Projection d'un fichier en mémoire

- **L'appel système** `munmap()` (`man 2 munmap`)
  - `#include <unistd.h>`
    `#include <sys/mman.h>`
    `int munmap(void * start,size_t length);`
  - La zone mémoire [`start`;`start+length`[n'est plus accessible
    - Zone totale de projection fichier
  - La synchronisation avec le fichier est forcée (si `MAP_SHARED`)
  - Le descripteur de fichier peut être fermé avant cet appel
  - Retour : `0` si ok, `-1` si erreur (mauvais arguments)

Projection d'un fichier en mémoire

- **L'appel système** `msync()` (`man 2 msync`)
  - `#include <unistd.h>`
    `#include <sys/mman.h>`
    `int msync(void * start,size_t length,int flags);`
  - Mettre à jour dans le fichier la zone mémoire [`start`;`start+length`[
    - Zone partielle → plus rapide
    - Utile pour les projections `MAP_SHARED`
  - `FLAGS` : type de synchronisation (ou bit à bit)
    - `MS_ASYNC` : demande de mise à jour dès que possible
    - `MS_SYNC` : mise à jour immédiate
    - `MS_INVALIDATE` : provoquer le rechargement des autres processus
    - `MS_ASYNC` ou `MS_SYNC` obligatoire
  - Retour : `0` si ok, `-1` si erreur (mauvais arguments)

Projection d'un fichier en mémoire

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

int main(void)
{
int fd=open("file.txt",O_RDWR);
struct stat infos;
if(!fstat(fd,&infos))
  {
  off_t sz=infos.st_size;
  char * p=(char *)mmap((void *)0,sz,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
  close(fd);
  if(p!=(char *)MAP_FAILED)
    {
    for(off_t i=0;i<sz/2;i++) { char c=p[i]; p[i]=p[sz-i-1]; p[sz-i-1]=c; }
    munmap(p,sz);
    }
  }
return 0;
}
```

Projection d'un fichier en mémoire

```
$ du -m file.txt       # 256 Mo de memoire, swap desactive
257     file.txt
$
$ head -2 file.txt
00-0000 01-0000 02-0000 03-0000 04-0000 05-0000 06-0000 07-0000 08-0000 09-0000
0A-0000 0B-0000 0C-0000 0D-0000 0E-0000 0F-0000 10-0000 11-0000 12-0000 13-0000
$
$ tail -2 file.txt
F4-0100 F5-0100 F6-0100 F7-0100 F8-0100 F9-0100 FA-0100 FB-0100 FC-0100 FD-0100
FE-0100 FF-0100
$
$ ./prog
$
$ head -3 file.txt

0010-FF 0010-EF
0010-DF 0010-CF 0010-BF 0010-AF 0010-9F 0010-8F 0010-7F 0010-6F 0010-5F 0010-4F
$
$ tail -2 file.txt
0000-31 0000-21 0000-11 0000-01 0000-F0 0000-E0 0000-D0 0000-C0 0000-B0 0000-A0
0000-90 0000-80 0000-70 0000-60 0000-50 0000-40 0000-30 0000-20 0000-10 0000-00$
$
```