Cours de Programmation Système

Révisions de C

Isabelle Ryl & Yann Hodique

{ryl,hodique}@lifl.fr

LIFL / USTL Lille 1

Liens et références

- http ://www.lifl.fr/∼hodique/Public/CUnixFc
- *Le noyau linux* – D. P. Bovet, Marco Cesati
- *Le langage C - Norme ANSI* – B. W. Kernighan, D. M. Ritchie
- les pages de manuel

Rappels généraux

Introduction

- Programme C : une fonction principale et des fonctions auxiliaires.
- Les fonctions sont réparties dans un ou plusieurs fichiers.

```
#include <stdio.h> /* importations */
#define coeff 2 /* macros */
int note_finale (int note_brute) { /* fonction aux */
  return note_brute * coeff;
}

int main (void) { /* fonction principale */
  int n;
  scanf("%d",&n);
  printf("%d\n", note_finale(n));
  return 0;
}
```

Compilation

- Par convention, un fichier source doit avoir l'extension `.c`
- Commandes de compilation :
  - `gcc essai.c` compile `essai.c` en un exécutable `a.out`
  - `gcc -o essai essai.c` produit un exécutable `essai`
- Options de compilation :
  - se rapporter à `man gcc`
  - en TP `-Wall -ansi -pedantic` obligatoires
    - ex : `gcc -Wall -ansi -pedantic -o essai essai.c`
  - `-Werror` *très fortement* conseillé

Généralités

- Toute fonction doit être déclarée avant d'être utilisée.
- Types de base : `char`, `short`, `int`, `long`, `float`, `double`, `...`
- structures de contrôle et opérateurs "classiques"

Le C est un langage **normalisé** par l'ANSI (American National Standards Institute). Sont définis :

- la syntaxe du langage
- les bibliothèques (regroupements de fonctions) standards
- les aspects indéfinis du langage :-) (taille d'un entier, d'un pointeur, etc.)

Entrées/sorties basiques

Sortie standard

La fonction `printf` définie dans `stdio.h` permet d'écrire sur la sortie standard.

```
int printf(const char *format, ...);
```

- La chaîne de caractères `format` contient des *caractères directs*, des *séquences d'échappement*, et des `spécifications de conversion`
- La valeur retournée est le nombre effectif de caractères écrits ou un code d'erreur négatif (`man 3 printf` pour plus de détails)

Exemple :

```
int n = 0;
char car = 'a';
printf("Un caractère : %c,\n et un entier %d\n, car, n);
```

Entrée standard

La fonction `scanf` définie dans `stdio.h` permet de lire sur l'entrée standard.

```
int scanf(const char *format, ...);
```

- Les données sont lues sur l'entrée standard conformément aux spécifications données par `format`
- Les valeurs lues sont affectées aux arguments suivant `format` qui doivent être des *pointeurs* valides
- Retourne le nombre d'objets lus et affectés dans le cas normal. `man 3 scanf` pour les autres possibilités.

Entrée/Sortie standards

Exemple :

```
int n;
char c;
int result;
result = scanf("%d %c", &n, &c);
printf("Saisies: %d et %c, resultat %d\n", n, c, result);
```

Caractères d'échappement :

\b	backspace	\f	saut de page
\n	fin de ligne	\r	retour chariot
\t	tabulation horizontale	\v	tabulation verticale
...		...	

Conversions

Voir `man 3 printf` et `man 3 scanf` pour une description complète
La forme générique d'une conversion est `%<nombre><lettre>` où `<nombre>` est appelé "précision" et `<lettre>` "spécifieur".

spécifieur	effet
d	entier sous forme décimale
i	entier (sous forme héxadécimale si précédé de `0x` ou `0X`, ou sous forme octale si précédé de 0)
c	`scanf` : les caractères (1 par défaut) sont mis dans le tableau `printf` : affichage d'un caractère
s	`scanf` : chaîne de caractère sans caractère d'espacement `printf` : affiche les caractères jusqu'au `\0` (par défaut)

Exemples

Conditions initiales :

- `char chaine[] = "qwertyuiop";`
- `char saisie[20];`

`printf("%s\n", chaine);`	`qwertyuiop`
`printf("%.5s\n", chaine);`	`qwert`
`printf("%c\n", chaine[0]);`	`q`
`scanf("%s", saisie);`	`q w e`
`printf("%s\n", saisie);`	`q`
`scanf("%5c", saisie);`	`qqqqq`
`printf("%s\n", saisie);`	`qqqqqDüÿ>`
`scanff("%5c", saisie);`	`q w e`
`saisie[5] = '\0';`	
`printf("%s\n", saisie);`	`q w e`

Tableaux

- Un tableau contient des valeurs d'un même type
- Les tableaux sont indicés par des valeurs entières de `0` à `longueur - 1`
- *Il n'y a pas de marque de fin de tableau*
- Déclaration :

```
int tent[10]; /* tableau de 10 entiers */
char tcar[8]; /* tableau de 8 caractères */
```

- Indexation :

```
tent[3] = 1; /* 4ème case du tableau */
tcar[6+1] = 'a'; /* dernière case du tableau */
tent[10] = 42; /* en dehors du tableau,
comportement indéterminé */
```

Utilisation des tableaux

```
int main (void) {
  int t[10], i;
  for (i = 0; i < 10; i++)
    t[i] = 10 + i;
  for (i = 0; i <= 10; i++)
    printf("%d ", t[i]);
}
```

Affichage :

```
10 11 12 13 14 15 16 17 18 19 -1073742824
```

Autre possibilité :

```
Segmentation fault
```

Chaînes de caractères

- les chaînes de caractères sont des tableaux de caractères
- par convention, la fin d'une chaîne est marquée par le caractère `'\0'` (code ASCII 0)
- toutes les fonctions d'entrées/sorties standards utilisent cette convention.

```
int main (void) {
  char machaine[10];
  int i;
  for (i = 0; i < 5; i++)
    machaine[i] = 'a' + i;
  machaine[5] = '\0';
  printf("%s!\n", machaine);
  return 0;
}
```

L'exécution de ce programme provoquera l'affichage `abcde!`

Parcours de chaîne

```
int main (void) {
  char machaine[10];
  int i = 0, cpt = 0;
  scanf("%s", machaine);
  while (machaine[i] != '\0') {
    if (machaine[i] == 'a')
      cpt++;
    i++;
  }
  printf("nb de 'a' dans %s : %d\n", machaine, cpt);
  return 0;
}
```

Exemple d'exécution :

```
bash$ a.out
atlas
nb de 'a' dans atlas : 2
bash$
```

Pointeurs

- Un pointeur sert à stocker une adresse physique et à accéder à l'objet situé à cette adresse.
- Il doit être déclaré en fonction du type pointé
  - `char * pc;` : pointeur de caractère
  - `int * pi;` : pointeur d'entier
  - `machin * pm;` : pointeur de machin
- L'opérateur unaire `&` donne l'adresse d'un objet en mémoire
- `&` ne s'applique que si il a un sens ! (pas aux expressions, variables `register`, ...)

Pointeurs (2)

```
char c = 'a';
```

déclare une variable de type `char`, de nom c, de valeur initiale a, d'adresse &c

```
char * p = &c;
```

déclare une variable de type `char *`, de nom p, de valeur initiale &c, d'adresse &p

On dit que p pointe sur c

Opérateur de dépointage

- L'opérateur `*` appliqué à un pointeur donne accès à l'objet pointé par ce pointeur

```
char * p;
char c = 'a';
p = &c;
```

- p et &c ont la même valeur
- `*`p et c valent `'a'`

Exemples

```
int x = 1, y = 2, z[10];

int *pi;

pi = &x;

y = *pi;

*pi = 0;

pi = &z[5];
```

Un pointeur ne peut pointer qu'un objet du type pour lequel il a été déclaré.

Passage de paramètre

```
void avance (int i) { i = i+1; }

int main (void) {
  int n = 5;
  avance(n);
  printf("%d",n);
  return 0;
}
```

Quel est l'affichage ? pourquoi ?

Passage de paramètres (2)

```
void avance (int *i) { *i = *i + 1; }

int main (void) {
  int n = 5;
  avance(&n);
  printf("%d",n);
  return 0;
}
```

Quel est l'affichage ? pourquoi ?

Pointeurs et tableaux

- Déclaration d'un tableau de 100 entiers :

```
int t[100];
```

- Prise d'un pointeur sur le premier élément de `t`

```
int *p;
p = &t[0];
```

- Copie du contenu de `t[0]` dans une variable `x` par le pointeur précédent

```
int x; x = *p;
```

Pointeurs et tableaux (2)

- Si p pointe sur un élément d'un tableau
  - `p + 1` pointe sur le suivant
  - `p + i` pointe sur le i^{me} après p
  - `p - i` pointe sur le i^{me} avant p
- Exemple
  - p pointe sur t[0]
  - Alors `p+i` est l'adresse de `t[i]` (quelle que soit la taille des éléments pointés)

Un tableau est un pointeur

- La valeur d'une variable de type tableau est l'adresse du premier élément du tableau

`p = &t[0]`	`p = t`
`*(t+i)`	`t[i]`
`t+i`	`&t[i]`
`*(p+i)`	`p[i]`

- Un tableau est une constante ! (inutilisable en l-value)

Arithmétique des pointeurs

- Opérations additives entre entier et pointeur
- Soustraction de deux pointeurs
- Toutes les comparaisons
- Par convention, on affecte `NULL` à un pointeur invalide

Exemple :

```
int strlen (char *s) {
  char *p = s;
  while (*p != '\0') p++;
  return p-s;
}
```

Chaînes de caractères et pointeurs

Les chaînes de caractères sont des tableaux donc des pointeurs.

```
void strcpy (char *d, char *s) {
  int i = 0;
  while ((d[i] = s[i]) != '\0') i++;
}

void strcpy (char *d, char *s) {
  while ((*d = *s) != '\0') {d++; s++}
}

void strcpy (char *d, char *s) {
  while ((*d++ = *s++) != '\0');
}

void strcpy (char *d, char *s) {
  while ((*d++ = *s++));
}
```

Chaînes constantes

- Les constantes de type chaîne sont exprimées entre guillemets :
  `"bonjour"`
- `"a"` et `'a'` ont des sens totalement différents
- Les caractères d'échappement peuvent être utilisés :
  `"bonjour, \n \t tout le monde"`
- Attention à distinguer ces deux déclarations
  - `char machaine[] = "bonjour";`
  - `char *machaine = "bonjour";`

Exemples

```
int pi[3] = {0,2,3};
int *qi;
int y;

*pi = 1;
y = *pi + 1;
*pi = *pi + 10;
*pi += 1;
qi = pi;
y = *(++qi);
y = *qi++;
```

Entrées et sorties bas niveau

Bas niveau

- pas de formatage
- pas de conversions
- uniquement lecture/écriture de **bytes**
- pas de bufferisation

Fichiers

- haut niveau
  - C : pointeur sur une structure `FILE`

```
File * f;
f = fopen("plop.txt","w");
fprintf(f, "une chaine\n");
```

  - C++ : objet `fstream`
- bas niveau
  - descripteur de fichier : un entier
  - `int f;`

Ouverture d'un fichier

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

- `pathname` : nom du fichier à ouvrir (chemins UNIX classiques)
- `flags` : mode d'accès au fichier (combinables par `|`)
  - `O_RDONLY`
  - `O_WRONLY`
  - `O_CREAT`
  - `O_APPEND`
  - `...`
- la valeur de retour est un descripteur de fichier valide ou `-1` en cas d'erreur

Création avec permissions

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

- valeur de retour, `pathname` et `flags`, idem que précédemment
- mode :
  - n'a du sens que si `O_CREAT` est précisé
  - permissions associées au fichier si il est créé

Remarque : après ouverture, l'index de lecture est positionné au début du fichier.

Exemple

essai.c :

```
#include <fcntl.h>

int main(void) {
  int f = fopen("toto", O_CREAT, 0640);
  return 0;
```

Exécution :

```
bash$ ls -l
total 0
bash$ essai
bash$ ls -l
-rw-r-----   1 yann yann         0 Apr 24 16:28 toto
```

Erreurs

En cas d'erreur, un code spécifique est attribué à la variable globale `errno`. La fonction `perror` permet d'obtenir un message d'erreur humainement lisible.

```
void perror(const char *);
```

Exemple :

```
...
if(open("plop", O_WRONLY) == -1) {
  perror("plop");
  exit(1);
}
...
```

Exécution :

```
bash$ essai
plop: No such file or directory
```

Fermeture d’un fichier

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

- Les fichiers ouverts par un programme sont fermés automatiquement à la terminaison
- Le nombre de descripteurs disponibles est limité
- Il est plus propre et plus sûr de fermer ses fichiers soi-même

Lecture d'un fichier

```
int read(int fd, char * buff, int count);
```

- Les lectures sont séquentielles
- `fd` doit être un descripteur valide de fichier ouvert en lecture
- `buf` est un pointeur sur sur la zone où les octets lus seront stockés (tableau ou zone allouée par `malloc()`, **pas** une chaîne de caractères.
- `count` est le nombre maximal d'octets à lire.
- le résultat de la fonction est soit :
  - `0` si la fin du fichier est atteinte
  - `-1` si une erreur s'est produite
  - un entier inférieur à `count` représentant le nombre d'octets effectivement lus

Exemple

```
int main(void) {
  int f, nb;
  char buffer[256];
  if((f = open("plop", O_RDONLY)) == -1) {
    perror("Ouverture");
    exit(1);
  }
  while((nb = read(f, buffer, 255)) > 0) {
    buffer[nb] = '\0';
    printf("%s",buffer);
  }
  if(nb == -1) {
   perror("Lecture");
    exit(1);
  }
  return close(f);
}
```

Écriture dans un fichier

```
int write(int fd, const char * buff, int count);
```

- Les écritures sont séquentielles
- `fd` doit être un descripteur valide de fichier ouvert en écriture
- `buf` est un pointeur sur sur la zone d'où les octets seront lus pour être écrits dans le fichier.
- `count` est le nombre d'octets à écrire.
- le résultat de la fonction est soit :
  - `-1` si une erreur s'est produite
  - un entier inférieur à `count` représentant le nombre d'octets effectivement écrits

Exemple

```
int main(void) {
  int f, nb;
  char buffer[] = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
  if((f = open("plop", O_WRONLY)) == -1) {
    perror("Ouverture");
    exit(1);
  }
  nb = write(f, buffer, 40);
  if(nb == -1) {
   perror("Ecriture");
    exit(1);
  }
  return close(f);
}
```

Descripteurs standards

Au début de l'exécution d'un programme, 3 descripteurs sont ouverts automatiquement :

- l'entrée standard : `0`
- la sortie standard : `1`
- la sortie d'erreur : `2`

Ces fichiers peuvent être utilisés de la même façon que les autres.

Exemple

```
#define SIZE 256

int main(void) {
  int nb;
  char buffer[SIZE];

  while((nb = read(0, buffer, SIZE)) > 0) {
    write(1, buffer, nb);
  }
  if(nb == -1) {
    perror("Lecture");
    return 1;
  }
  return 0;
```

Positionnement

```
long lseek(int fd, long offset, int pos);
```

- Permet de rompre l'accès séquentiel aux fichiers
- `fd` est un descripteur valide
- `offset` est la longueur du déplacement (entier signé)
- `pos` le point de départ du déplacement :
  - `SEEK_SET` : début du fichier
  - `SEEK_CUR` : position courante dans le fichier
  - `SEEK_END` : fin du fichier
- la valeur de retour est
  - la nouvelle position dans le fichier
  - -1 en cas d'erreur (dans ce cas la position courante est inchangée)

Exemple

```
if((f = open("plop", O_RDONLY)) == -1) {
  perror("Ouverture");
  exit(1);
}

nb = read(f, buffer, SIZE);
lseek(f, 0, SEEK_SET); // retourne au début

nb = read(f, buffer, SIZE);
lseek(f, -nb, SEEK_CUR); // retourne au début

lseek(f, nb, SEEK_SET); // retourne à l'emplacement précéden
```

Taille de fichier

```
int main(void) {
  int f;
  char buffer[SIZE];
  long taille;

  if ((f = open("plop", O_RDONLY)) == -1) {
    perror("Ouverture");
    exit(1);
  }
  taille = lseek(f, 0, SEEK_END);
  lseek(f, 0, SEEK_SET);

  printf("toto: %d octets\n", taille);
  return 0;
}
```

Entrées / sorties de données structurées

Il est possible d'utiliser les entrées/sorties bas niveau pour lire autre chose que de simples octets. Mais il faut contrôler soi-même

- le formatage des données
- la taille des données

```
struct {
  int i1;
  char c;
  int i2 } v1, v2;

...
write(f, &v1, sizeof v1);
...
read(f, &v2, sizeof v2);
...
```

Système de fichiers

Problématique

- Point de vue utilisateur : sauvegarder des données, les organiser, y avoir accès facilement. Essentiellement caractérisé par :
  - un nom ;
  - éventuellement une localisation.
- Point de vue système
  - gestion des ressources disques et autres ;
  - différentes informations comme la taille, la date de création, date de dernière modification, . . .
- Point de vue système multi-utilisateurs
  - partage des ressources entre les utilisateurs ;
  - protection des fichiers.

Les fichiers unix

En Unix, tout est fichier.

- Le terme de fichier désigne des ressources :
  - matérielles – disque, disquettes, terminal, . . .
  - logicielles – fichiers disques classiques contenant des données mémorisées sur le disque.
- Les primitives génériques d'accès aux fichiers permettent de réaliser des opérations de lecture et d'écriture sur toutes les ressources du système.
- En interne, chaque fichier est associé à une structure décrivant ses caractéristiques, appelée i-noeud ou i-node.

Organisation logique

- Les fichiers UNIX peuvent être des fichiers disques classiques ou des fichiers ressources
- Plusieurs disques peuvent être connectés à une machine ainsi que de nombreuses ressources.
- Chaque disque physique peut également être partitionné en plusieurs disques logiques.
- À chaque fichier correspond un i-noeud (i-node) qui contient entre autres, l'identification du disque logique du fichier et son numéro d'identification dans ce disque logique.
- Tous les fichiers apparaissent à l'utilisateur dans une arborescence unique.

Les i-noeuds

Un fichier n'est pas seulement repéré par son nom : à chaque fichier est associé un i-noeud. Chaque i-noeud contient les informations suivantes :

- identification du propriétaire et du groupe propriétaire du fichier ;
- type et droits du fichier ;
- taille du fichier en nombre de caractères (si possible) ;
- nombre de liens physiques du fichier ;
- trois dates (dernier accès au fichier, dernière modification du fichier, dernière modification du i-noeud) ;
- adresse des blocs utilisés sur le disque pour ce fichier (pour les fichiers disques) ;
- identification de la ressource associée (pour les fichiers spéciaux).

Les types de fichier

La caractéristique essentielle d'un fichier UNIX est son type, il permet de définir les opérations qui sont applicables à ce fichier.

- Les fichiers réguliers : fichiers de données, suites d'octets non organisées. Caractéristique essentielle : la taille qui permet au système de retrouver la fin de fichier.
- Les répertoires : fichiers de données munis d'une structure logique décrivant les entrées du répertoire.
- Les fichiers spéciaux : associés aux ressources du système (pas de taille) blocs (ex : disques) ou caractères (ex : terminaux).
- Les liens symboliques : dont le contenu est interprété comme un nom de fichier.
- Les tubes : communication entre processus.
- Les sockets.

Exemple

```
[yann@siav ~/]$ ls -l /dev
total 0
crw-rw----  1 yann audio     14,  12 May 18 17:48 adsp
crw-rw----  1 root video     10, 175 May 18 19:47 agpgart
crw-rw----  1 root root      36,   8 May 18 19:47 arpd
crw-rw----  1 yann audio     14,   4 May 18 17:48 audio
crw-------  1 yann root       5,   1 May 18 17:48 console
lrwxrwxrwx  1 root root           11 May 18 17:47 core -> /p
drwxr-xr-x  3 root root           60 May 18 17:47 cpu/
drwxr-xr-x  3 root root           60 May 18 17:47 discs/
```

```
[yann@siav ~/]$ ls -l /dev/pts
total 0
crw-------  1 yann yann 136, 0 May 18 17:50 0
crw--w----  1 yann yann 136, 1 May 18 23:37 1
crw--w----  1 yann yann 136, 2 May 18 23:37 2
```

Propriétaire et groupe

- Type, droits, modifications des droits, . . .
- Test des droits d'accès :

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

  - teste les droits d'accès au fichier par rapport au propriétaire et groupe propriétaire du processus ;
  - `mode` est une combinaison logique (opérateur `|`) des constantes :
    - `R_OK` : accès en lecture,
    - `W_OK` : accès en écriture,
    - `X_OK` : accès en exécution,
    - `F_OK` : test de l'existence du fichier.
  - retourne 0 si l'accès est autorisé, -1 sinon.

Modification du propriétaire et du groupe

La commande

```
chown prop file
```

permet de modifier le propriétaire et le groupe d'un fichier (cf `man chown`)

```
[root@siav ~yann/]# ls -l plop
-rw-r--r--  1 yann yann 10427 May 19 00:16 plop
[root@siav ~yann/]# chown root plop
[root@siav ~yann/]# ls -l plop
-rw-r--r--  1 root yann 10427 May 19 00:16 plop
[root@siav ~yann/]# chown .root plop
[root@siav ~yann/]# ls -l plop
-rw-r--r--  1 root root 10427 May 19 00:16 plop
[root@siav ~yann/]# chown yann.yann plop
[root@siav ~yann/]# ls -l plop
-rw-r--r--  1 yann yann 10427 May 19 00:16 plop
```

Les liens physiques

- Le nombre de liens physiques du fichier est le nombre de noms du fichier, le nombre de noms (chaînes de caractères) associés au même i-noeud.
- La commande `ln fichier lien` permet de créer des liens physiques. Le fichier spécifié ne peut être un répertoire et le répertoire dans lequel le lien est créé doit appartenir au même disque logique que le fichier.

Exemple

```
[yann@siav ~/tmp/plop]$ ls -li
total 0
34024808 -rw-rw-r--  1 yann yann 0 May 19 00:32 plop
[yann@siav ~/tmp/plop]$ ln plop plip
[yann@siav ~/tmp/plop]$ ls -li
total 0
34024808 -rw-rw-r--  2 yann yann 0 May 19 00:32 plip
34024808 -rw-rw-r--  2 yann yann 0 May 19 00:32 plop
```

Lecture d'un i-noeud

Les caractéristiques d'un i-noeud peuvent être lues par un appel à la fonction :

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

- permet d'obtenir dans la structure pointée par `buf` les caractéristiques du fichier de nom `file_name`
- retourne 0 en cas de succès et -1 en cas d'échec ;
- le processus doit posséder les droits d'accès en recherche sur tous les répertoires intervenant dans le chemin d'accès au fichier.

Remarque : le pointeur passé en paramètre doit pointer sur une structure (pas de pointeur non alloué !)

Structure stat

Principaux champs de la structure `stat` :

```
struct stat {
  dev_t         st_dev;      /* device */
  ino_t         st_ino;      /* inode */
  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 type (if inode device
  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 *
};
```

Tests

Des macros, applicables au champ mode de l'i-noeud, permettent de tester facilement le type d'un fichier (technique des masques) :

- `S_ISLNK(m)` est-ce un lien symbolique ?
- `S_ISREG(m)` est-ce un fichier régulier ?
- `S_ISDIR(m)` est-ce un répertoire ?
- `S_ISCHR(m)` est-ce un fichier caractères ?
- `S_ISBLK(m)` est-ce un fichier blocs ?
- `S_ISSOCK(m)` est-ce une socket ?
- `...`

Exemple

```
int main (void) {
    struct stat carac;
    char nom[80];
    scanf("%s", nom);
    if(stat(nom, &carac) == -1) {
        perror("stat");
        exit(1);
    }
    printf("Inode : %ld\n", carac.st_ino);
    printf("Disque : %ld\n", (long)carac.st_dev);
    printf("Lecture : %s\n",
           (access(nom, R_OK) ? "non" : "oui"));
    printf("Fichier regulier : %s\n",
           (S_ISREG(carac.st_mode) ? "oui" : "non"));
    return 0;
}
```

Organisation physique des disques

- Le système de gestion de fichiers permet de gérer l'espace disque et d'organiser les fichiers.
- Les disques sont gérés par blocs physiques.
- Plusieurs disques peuvent être utilises, chacun peut être partitionné en plusieurs disques logiques.
- Plusieurs systèmes de fichiers différents peuvent coexister dans plusieurs partitions.
- Il est possible d'utiliser des systèmes de fichiers distants par le biais de protocoles comme NFS.
- Les partitions de `swap` sont utilisées pour la gestion mémoire.

Organisation d'un fichier

Les blocs de données d'un fichier sont éparpillés parmi l'ensemble des blocs de données. L'i-noeud d'un fichier contient 13 numéros de blocs de données du fichier :

- 10 numéros de blocs de données ;
- le 11ème est un bloc indirect contenant des numéros de blocs de données ;
- le 12ème est un bloc indirect contenant des numéros de blocs indirects qui contiennent des numéros de blocs de données ;
- le 13ème est un bloc indirect contenant des numéros de blocs indirects qui contiennent des numéros de blocs indirects qui contiennent des numéros de blocs de données.

Montage

- Le système de fichiers de chaque disque logique constitue une arborescence.
- Chaque arborescence possède une table de fichiers propre.
- La racine de tout système de fichiers a pour i-noeud .
- Les arborescences peuvent être reliées entre elles par le mécanisme de montage : greffer la racine d'une arborescence non montée en un point accessible a partir de la racine absolue .
- L'un des disques est privilégié (disque système) : sa racine est la racine absolue du système (/) et les autres disques sont montés par rapport à celui-là.

Conséquences

- Tous les fichiers apparaissent dans une arborescence unique
- Plusieurs fichiers différents peuvent avoir le même numéro d'i-noeud correspondant à des entrées differentes dans des tables différentes !
- Le fichier est défini de manière non ambiguë par ses numéros d'i-noeud ET de disque logique.

Commande mount

- Commande permettant de monter un système de fichiers. (normalement réservée au super-utilisateur).
- Sans paramètre, affiche la table des montages actuels

```
[yann@siav ~/cvs/emacs]$ mount
/dev/hda5 on / type ext3 (rw,noatime)
none on /proc type proc (rw)
none on /proc/bus/usb type usbfs (rw)
none on /sys type sysfs (rw)
/dev/hda6 on /home type xfs (rw,noatime)
/dev/hda1 on /boot type ext3 (rw,noatime)
/dev/hda10 on /tmp type ext2 (rw,noatime)
/dev/hda8 on /usr type reiserfs (rw,noatime,notail)
/dev/hda9 on /var type reiserfs (rw,noatime,notail)
automount(pid4331) on /net type autofs (rw,fd=4,pgrp=4331,
```

Command mount (suite)

- `mount -a` monte tous les systèmes spécifiés dans `/etc/fstab` avec les options précisées
- Pour monter un système de fichiers particulier, la commande générale est : `mount -t <type> <device> <dir>`

```
mount -t msdos /dev/fd0 ./floppy
```

NFS

- NFS : Network File System, protocole spécifié et développé par SUN Micro-Systems.
- Permet un accès aux fichiers distants par l'intermédiaire des mêmes transparence fonctions que celles utilisées pour les accès locaux pour l'utilisateur.
- Établit un système de fichiers virtuel sur les machines permettant de voir dans une arborescence unique les fichiers locaux et distants.
- Principe : le système de fichiers d'une machine 1 (serveur) peut être rendu visible sur une machine 2 (client) par une extension du mécanisme de montage. Par exemple : monte le répertoire de la machine sur le répertoire de la machine locale.

NFS : remarques

- NFS est un serveur sans état : le serveur ne garde pas d'informations concernant le protocole (aucune trace des requêtes NFS précédentes ni sur la manière dont elles sont reliées entre elles).
- Choix de conception pour faciliter les reprises sur panne.
- Problèmes de la cohérence des informations pour les différents clients, gestion de caches, . . .
- Problèmes de sécurité :
  - identification des utilisateurs.
  - gestion des accès du super-utilisateur.
  - . . .

Exportation de fichiers

- Le fichier `/etc/exports` permet de spécifier sur une machine les répertoires qui peuvent être exportés ainsi que les types d'accès autorisés.

Manipulation des différents types de fichiers

Les fichiers réguliers

Opérations d'ouverture, de lecture, d'écriture, de fermeture.

- Primitives de base : entrees/sorties bas niveau.
- Fonctions de la bibliothèque standard C :
  - `fopen`, `fclose`, `...`
  - `fread`, `fwrite`
  - lectures/écritures de caractères : `fgetc`, `fputc`
  - lectures/écritures de chaînes de caractères : `fgets`, `fputs`
  - lectures/écritures formatées : `fscanf`, `fprintf`
- Flots en C++ :
  - objet `fstream`, méthodes `open` et `close` ;
  - opérateurs << et >>
  - méthodes `read` et `write`
  - `...`

Entrées/sorties sur les répertoires

- La structure des répertoires dépend du système de fichiers, de nombreuses structures différentes existent.
- Un répertoire est un fichier, il peut être ouvert et lu grâce à pour trouver les noms des fichiers qu'il contient. Difficile compte tenu des variations dans la forme des structures des répertoires.
- La solution : utiliser l'interface standard pour la lecture par programme des fichiers. Cette interface est indépendante du système de fichiers.

Ouverture d'un répertoire

Le type `DIR` est defini dans `dirent.h` : structure permettant de manipuler un répertoire.

```
DIR *opendir(const char *name);
```

- Ouvre le repertoire de nom `name` ;
- alloue un objet de type `DIR` pour ce repertoire ;
- renvoie l'adresse de l'objet alloué ;
- retourne `NULL` en cas d'erreur.

Remarques :

- Après ouverture, le pointeur est positionné sur la première entrée du répertoire.
- Il n'est pas nécessaire à l'application de connaître les détails de la définition de la structure `DIR`.

Lire une entrée d'un répertoire

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

lit la prochaine entrée du répertoire désigné par ;

- retourne un pointeur sur une structure `dirent` décrivant cette entrée ;
- retourne `NULL` si la lecture du répertoire est terminée ou en cas d'erreur.

Le seul champ de la structure `dirent` imposé par la norme est un champ `char d_name[]` qui contient au plus `NAME_MAX` caractères significatifs (sans compter le `'\0'`)

Autres fonctions sur les répertoires

- Fermer un repertoire :

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

retourne `0` en cas de succes et `-1` en cas d'erreur.

- Les écritures directes sont impossibles (consistance du système de fichiers).
- Créer un répertoire vide :

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

- Éffacer un répertoire vide :

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

- Effets de bord d'autres fonctions (par exemple créer un fichier dans un répertoire le modifie par ajout d'une entrée).

Exemple

Lister tous les fichiers du répertoire courant :

```
int main (void) {
  DIR *dir;
  struct dirent *entree;
  if ((dir = opendir(".")) == NULL) {
    perror("Ouverture");
    exit(1);
  }
  while ((entree = readdir(dir)) != NULL)
    printf("%s\n", entree->d_name);
  if (closedir(dir) == -1) {
    perror("Fermeture");
    exit(1);
  }
  return 0;
}
```

Liens symboliques

- Les liens symboliques sont des fichiers dont le contenu est interprété comme un nom de fichier.
- Les fonctions usuelles suivent les liens symboliques (i.e. les rendent transparents à l'utilisateur).
- Création d'un lien symbolique :
  - par la commande `ln` avec l'option `-s`,
  - par appel à la fonction

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

- consultation des caractéristiques :

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

Exemple

```
[yann@siav ~]$ ls -li
total 0
34024808 -rw-rw-r--  1 yann yann 0 May 19 00:32 plop
[yann@siav ~]$ ln -s plop plip
[yann@siav ~]$ ls -li
total 0
34024800 lrwxrwxrwx  1 yann yann 4 May 26 00:33 plip -> plop
34024808 -rw-rw-r--  1 yann yann 0 May 19 00:32 plop
[yann@siav ~]$ ls
plip@  plop
```

- les numéros d'i-noeud sont différents
- la commande `ls` identifie clairement les liens :
  - par des symboles (`->` ou `@`
  - par le type du fichier (`l`)

Lecture d'un lien

La fonction `readlink` permet de lire le contenu d'un lien (`unistd.h`) :

```
int readlink(const char *path, char *buf, size_t bufsiz);
```

- lit le contenu du lien `path` ;
- place la valeur du lien dans le buffer `buf` qui a la taille `bufsiz` (cette taille doit être fixée AVANT l'appel) ;
- si `buf` est trop petit, la chaîne est tronquée ;
- retourne le nombre de caractères effectivement placés dans `buf` (attention il n'y a PAS de `'\0'`) ;
- retourne -1 en cas d'echec.

Exemple

```
int main (void) {
  int nb;
  char nom[80];
  char buffer[501];

  scanf("%s", nom);
  if ((nb = readlink(nom, buffer, 500)) == -1) {
    perror("readlink");
    exit(1);
  }
  buffer[nb] = '\0';
  printf("Valeur du lien : %s\n", buffer);
  return 0;
}
```

Processus

Processus

- Objet dynamique correspondant à l'exécution d'un programme.
- Un processus possède un espace d'adressage qui définit l'ensemble des objets qui lui sont propres (instructions et données).
- Le processus peut s'exécuter dans deux modes différents :
  - en mode utilisateur, le processus exécute des instructions du programme et accède aux données de son espace d'adressage.
  - en mode noyau, le processus exécute des instructions du noyau et à acces a l'ensemble des données du système (par exemple lors des appels système).
- Chaque processus possède un espace d'adressage de données propres, plusieurs processus peuvent partager le même programme (code réentrant).

Naissance des processus

- Tout processus peut creer de nouveaux processus.
- Tout processus (sauf le premier) est créé par un appel à la primitive `fork()`
- La primitive `fork()` a pour effet de dupliquer le processus appelant.
- Les processus sont organisés en arborescence en fonction de leur processus créateur appelé père.
- Le noyau du système a en charge la gestion des différents processus et le partage des ressources entre eux, en particulier l'ordonnancement des processus : choisir parmi les processus en attente celui qui doit être activé.

Caractéristiques d'un processus

- Identite du processus (`pid`)
- Identite du pere du processus (`ppid`)
- Liens avec les utilisateurs :
  - propriétaire réel du processus (`uid` de l'utilisateur qui a lancé le processus) ;
  - propriétaire effectif du processus (différent du propriétaire réel par exemple lorsqu'un processus correspond à l'exécution d'un programme dont le bit u est positionné) ;
  - groupe réel du processus ;
  - groupe effectif du processus (différent du groupe réel par exemple lorsqu'un processus correspond a l'exécution d'un programme dont de bit g est positionné).

Remarque. Un processus ayant des droits privilégiés peut modifier ses propriétaires et groupes réels ou effectifs (procédure utilisée à la connexion d'un utilisateur).

Caractéristiques (suite)

- Le répertoire de travail du processus.
- La date de création du processus.
- Les temps CPU consommés par le processus en modes utilisateur et noyau ainsi que par ses fils terminés.
- Le masque de création des fichiers.
- La table des descripteurs de fichiers.
- L'état du processus.
- L'événement attendu par le processus s'il est à l'état endormi.
- Les informations pour le traitement des signaux.
- Les verrous sur les fichiers.
- ...

Fonctions donnant les caractéristiques

- `pid_t getpid(void)` Identité du processus
- `pid_t getppid(void)` Identité du processus père
- `pid_t getuid(void)` Identité du propriétaire réel
- `pid_t geteuid(void)` Identité du propriétaire effectif
- `pid_t getgid(void)` Groupe propriétaire réel
- `pid_t getegid(void)` Groupe propriétaire effectif
- `int chdir(const char *path)` Change le répertoire courant
- `char *getcwd(char *buf, size_t size)` Rép. courant

État d'un processus

Au cours de sa vie, un processus passe par différents états :

- état transitoire à sa creation ou lors de la création d'un fils ;
- état actif (en mode noyau ou utilisateur) (état R donne par `ps`) ;
- état prêt, en attente de la CPU (état R donne par `ps`) ;
- état endormi, en attente d'un événement (attente d'entrées/sorties, attente de terminaison d'un processus, attente d'un signal, . . .) (état S ou D donné par `ps`) ;
- état suspendu (état T donne par `ps`) ;
- état zombi, processus terminé mais dont le père n'a pas encore pris connaissance de la terminaison (état Z donné par `ps`).

Organisation mémoire

Le processus est constitué de 4 segments mémoire :

- le bloc de contrôle qui contient les informations utiles au système. Cette partie n'est pas directement accessibles aux utilisateurs. Les éléments du bloc de contrôle sont de deux natures :
  - les informations utiles lorsque le processus est actif (descripteurs de fichiers, signaux, . . .), elles appartiennent à l'espace d'adressage du processus,
  - les informations utiles au système même lorsque le processus n'est pas actif (identité, priorité, . . .), elles n'appartiennent pas à l'espace d'adressage du processus.

  Le bloc de contrôle a une taille fixe quel que soit le processus.

Organistation mémoire (suite)

- Les instructions qui appartiennent à l'espace d'adressage du processus et peuvent être partagées entre plusieurs processus si le code est réentrant, de taille fixe pour un programme donné.
- Les données manipulées par le programme qui appartiennent par excellence à l'espace d'adressage du processus et dont la taille varie au grès des allocations mémoire.
- La pile dont la taille varie en fonction de l'imbrication des appels de fonctions (allocation des variables locales, sauvegarde des contextes, . . .).

Primitive `fork`

```
pid_t fork(void);
```

- permet de creer un processus fils ;
- après la création, les deux processus semblent avoir exécuté l'appel à la primitive `fork`, chacun des processus continue son exécution à partir de l'instruction qui suit le `fork` ;
- la valeur de retour de `fork` permet de différencier les processus père et fils :
  - 0 dans le fils ;
  - le `pid` du fils dans le père.
- la fonction retourne -1 en cas d'echec, dans ce cas il n'y a pas création d'un nouveau processus.

Exemple 1

```
int main (void) {
    int a = 10;
    int numero;
    numero = fork();
    if(numero == -1) {
        perror("Erreur du fork");
        exit(1);
    }
    a = a + numero;
    printf("%d\n",a);
}
```

Exemple 2

```
int main (void) {
    int pid = fork();
    printf("coucou \n");
    numero = fork();
    if(pid != 0) {
        printf("Processus pere de pid %d", getpid());
        printf(" Creation d'un fils de pid %d\n", pid);
    } else {
        printf("Processus fils de pid %d", getpid());
        printf(" Variable pid = %d\n", pid);
    }
    return 0;
}
```

Exemple 3

```
int main(void) {
    int pid = fork();
    if(pid != 0) {
        printf("Processus de pid %d", getpid());
        printf(" de pere %d\n", getppid());
        printf("Fin du pere, valeur du fork %d\n", pid);
    } else {
        // sleep 2;
        printf("Processus de pid %d", getpid());
        printf(" de pere %d\n", getppid());
        printf("Fin du fils, valeur du fork %d\n", pid);
    }
    return 0;
}
```

Génétique de processus

Le processus fils hérite les caractéristiques de son père excepté :

- le `pid` du fils est différent de celui du père (le `pid` étant l'identifiant d'un unique processus) ;
- les temps CPU (ils sont mis a 0 pour le fils)
- les verrous sur les fichiers ;
- les signaux pendants (voir cours sur les signaux) ;
- la priorité (la priorité est utilisée pour l'ordonnancement, la priorité du fils est initialisée à une valeur standard lors de sa création).

Terminaison des processus

- Tout processus UNIX possède une valeur de retour (valeur de retour de la fonction main, ou code utilisé pour la fonction) à laquelle son père peut acceder.
- Tout processus se terminant passe dans l'etat zombie (état Z indiqué par la commande `ps`), jusqu'à ce que son père prenne connaissance de sa terminaison.
- Le mécanisme de processus permet à un processus d'accéder au code de retour de ses processus fils de manière asynchrone.

Remarque. Si le processus père se termine sans avoir pris connaissance de la terminaison d'un de ses fils, celui-ci est adopté par le processus de `pid 1` qui prend connaissance du code de retour du fils et lui permet ainsi de se terminer.

Primitive `wait`

```
pid_t wait(int *status);
```

- si le processus ne possede aucun fils, la fonction retourne -1 ;
- si le processus possède des fils qui ne sont pas en l'etat zombie, le processus est bloqué jusqu'au passage d'un des fils dans l'état zombie (ou réception d'un signal) ;
- si le processus possède au moins un fils zombie, la fonction retourne le `pid` de l'un des fils zombie choisi par le système ! Dans ce cas :
  - le processus zombie considéré disparait de la liste des processus (maintenue par le systeme),
  - si l'adresse `status` est differente de `NULL`, elle reçoit les informations sur la terminaison du processus zombie.

Primitive `waitpid`

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- le parametre `pid` peut avoir les valeurs suivantes :
  - < -1 tout processus fils dans le groupe $\mid pid \mid$
  - -1 tout processus fils,
  - 0 tout processus fils du même groupe que l'appelant,
  - > 0 processus d'identité `pid`
- le paramètre `options` est une combinaison (|) des constantes `WNOHANG` (appel non bloquant), et `WUNTRACED` (un appel avec cette option indique si le processus demandé est bloqué, si l'information n'a pas encore été transmise) ;
- la fonction retourne :
  - -1 en cas d'erreur,
  - 0 en cas d'echec en mode non bloquant (le processus demandé existe mais n'est ni bloqué ni stoppé),
  - le `pid` du fils en cas de succès.

Interprétation du code de retour

La valeur reçue par l'intermédiaire de la fonction `wait` n'est PAS directement la valeur retournée par le processus fils ! Les macros suivantes doivent etre utilisées pour tester cette valeur (entière) :

- `WIFEXITED(stat)` : vrai si le fils s'est terminé normalement ;
- `WEXITSTATUS(stat)` : fournit le code de retour du fils (les 8 bits de poids faible) uniquement s'il s'est terminé normalement ;
- `WIFSIGNALED(stat)` : vrai si le fils s'est terminé à cause d'un signal ;
- `WTERMSIG(stat)` : retourne le numero du signal ayant provoqué la terminaison du fils si le fils s'est terminé à cause d'un signal ;
- `WIFSTOPPED(stat)` : vrai si le fils est stoppé (utilisable uniquement avec l'option `WUNTRACED` ;
- `WSTOPSIG(stat)` : retourne le numero du signal qui a stoppé le fils si le fils a ete stoppé par un signal.

Exemple

```
int main(void) {
    int retour;
    int pid = fork();
    if(pid != 0) {
        printf("Processus de pid %d de pere %d\n",
               getpid(), getppid());
        if (wait(&retour) == -1) {
            perror("Pas de fils pour le wait");
            exit(1);}
        if(WIFEXITED(retour) != 0) {
            printf("fin normale du fils detectee ");
            printf(" code de retour %d\n",
                   WEXITSTATUS(retour));}
        printf("Fin du pere, valeur du fork %d\n", pid);
        exit(0);
    }
    else {
        printf("Fin du fils %d de pere %d\n",
               getpid(), getppid());
        exit(3);}
}
```

Commande `exec`

```
int execvp(const char *file, char *const argv[]);
```

- recouvre le programme en cours qui exécute l'appel par le programme mentionné en premier paramètre ;
- le paramètre représente le tableau d'arguments du programme à appeler (argv[0] doit etre le nom du programme et le tableau doit se terminer par `NULL`
- il n'y a pas de création de processus, le processus poursuit son exécution mais exécute désormais le nouveau programme ;
- retourne -1 en cas d'erreur ;
- si l'appel n'a pas declenché d'erreur, il n'y a pas de retour ;
- la particuliarité de `execvp` est de rechercher le fichier a exécuter dans les répertoires contenus dans le `PATH` (comme le fait le `shell`) s'il n'y a pas de `/` dans le nom de fichier spécifié.

Remarque. Voir aussi les autres fonctions de la famille par `man 3 exec`.

Exemple

```
/* affiche.c */
int main (int argc, char ** argv) {
    int i;
    printf("Processus %d\n", getpid());
    if(argc < 2) return 1;
    for (i = 1; arv[i] != NULL; i++)
        printf("numero %d = %s\n", i, argv[i]);
    return 0;
}
```

```
/* essai.c */
int main (void) {
    char * arg[4] = {"affiche", "tralala", "3", NULL};
    printf("Processus %d\n", getpid());
    execvp(arg[0], arg);
    perror("execvp");
    return -1;
}
```

Tubes

Généralités

- Les tubes sont des mécanismes permettant aux processus de communiquer entre eux
- Les tubes appartiennent au système de fichier UNIX, ils sont décrits par un i-node
- Les tubes peuvent donc être manipulés par l'intermédiaire de descripteurs de fichiers et par exemple des primitives d'entrées/sorties bas niveau `read` et `write`
- Les tubes sont des moyens de communication unidirectionnels

Généralités (2)

- Un tube correspond au plus à deux entrées dans la table des fichiers ouverts (une entrée en lecture et une entrée en écriture).
- Les données ne sont pas formatées, elles apparaissent comme un flot de caractères. Le tube est géré en file, i.e. la première donnée écrite dans le tube est également la première donnée lue.
- Attention, un tube a une capacité finie !
- Le nombre de lecteurs d'un tube est le nombre de descripteurs associés à l'entrée en lecture sur le tube. Si ce nombre est nul, il est impossible d'écrire dans le tube.
- Le nombre de rédacteurs d'un tube est le nombre de descripteurs associés a l'entrée en ecriture sur le tube. Si ce nombre est nul, la primitive `read` détecte une fin de fichier.

Création d'un tube

La primitive

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

- permet de créer un tube ;
- a pour paramètre un tableau de deux entiers qui va permettre de stocker les deux descripteurs ;
- range dans `filedes[0]` le descripteur permettant de lire dans le tube et dans `filedes[1]` le descripteur permettant d'écrire dans le tube ;
- retourne 0 en cas de succes et -1 en cas d'erreur.

Tubes vs fichiers

- Comme les tubes n'ont pas de noms, il est impossible de les ouvrir grâce à la primitive `open`. En conséquence, un processus peut acquérir un descripteur sur un tube, soit en le créant, soit par héritage.
- Seuls le processus ayant crée le tube et sa descendance peuvent y acceder. Si un processus perd son accès au descripteur sur le tube (par exemple par un appel à `close`, il n'a aucun moyen de le récuperer par la suite.
- Les acces en lecture et écriture sont effectués par des appels à `read` et `write` comme pour les fichiers.
- Les lectures dans un tube sont effaçantes il s'agit bien d'une extraction des données du tube et non d'une consultation (des fonctions comme `lseek` n'ont aucun sens dans le cas des tubes).

Exemple

```
int main (void) {
    int pid;
    int fd[2];

    if(pipe(fd) == -1) {perror("Creation du tube"); exit(1);}
    pid = fork();
    if (pid != 0) {
        printf("Processus pere de pid %d\n", getpid());
        close(fd[0]);
        write(fd[1], "Bonjour", 8);
    } else {
        char mot[8];
        printf("Processus fils de pid %d\n", getpid());
        close(Fd[1]);
        read(fd[0], mot, 8);
        printf("Fils >> mot recu : %s\n", mot);
    }
    return 0;
}
```

Pourquoi fermer les descripteurs

```
int main (void) {
    int pid;
    int fd[2];

    if (pipe(fd) == -1) {
        perror("Creation du tube");
        exit(1);
    }
    pid = fork();
    if(pid == 0) {
        printf("Processus fils de pid %d\n", getpid());
        close(fd[0]);
        write(fd[1], "Bonjour", 8);
        close(fd[1]);
    }
```

Pourquoi fermer les descripteurs

```
 else {
        char mot[9];
        int n;
        printf("Processus pere de pid %d\n", getpid());
        close(fd[1]); // si absent, processus ne termine pas
        while ((n = read(fd[0], mot, 8)) != 0) {
            mot[n] = '\0';
            printf("Pere >> mot recu %s\n", mot);
        }
        printf("Pere >> terminaison\n");
    }
    return 0;
}
```

Lecture sur un tube

- L'appel à la primitive `read` est, par défaut, bloquant (i.e. le processus effectuant l'appel est arrêté jusqu'à ce qu'il y ait quelque chose à lire dans le tube).
- Comme dans le cas des fichiers, un appel `read(fd[0], buffer, size)` demande la lecture de `size` octets qui seront placés à l'adresse `buffer`. La valeur retournée est le nombre d'octets effectivement lus (i.e. disponibles dans le tube au moment de la lecture).
- Si le nombre d'ecrivains est nul, `read` détecte la fin de fichier et retourne 0.

Il faut toujours fermer tous les descripteurs inutiles dès que possible!!!

Écriture dans un tube

L'écriture est réalisée par appel à la primitive `write`.

- Si le nombre de lecteurs est nul, le rédacteur reçoit un signal et se termine si le signal n'est pas traité).
- Si le nombre de lecteurs n'est pas nul, le processus écrit les données dans le tube.

Primitive dup

```
int dup(int oldfd);
```

- crée une copie d'un descripteur de fichier ;
- retourne le nouveau descripteur (le plus petit libre) ;
- retourne -1 en cas d'erreur ;
- l'ancien descripteur et le nouveau peuvent être utilisés de manière interchangeable, ils partagent la position courante dans le fichier.

Redirection de la sortie standard

```
int main (void) {
    int pid;
    int fd[2];
    int n;

    if (pipe(fd) == -1) {perror("Creation du tube"); exit(1)
    pid = fork();
    if(pid == 0) {
        char buffer[20];
        close(fd[1]);
        n = read(fd[0], buffer, 19);
        while (n != 0) {
            buffer[n] = '\0';
            printf("Fils --> %s n = %d\n", buffer, n);
            n = read(fd[0], buffer, 19);
        }
        close(fd[0]);
        printf("Fils >> terminaison\n");
    }
```

Redirection de la sortie standard

```
 else {
        int i;
        printf("Processus pere de pid %d\n", getpid());
        close(fd[0]);
        close(1);
        dup(fd[1]);
        close(fd[1]);
        for (i = 0; i < 5; i++) {
            printf("Bonjour");
            fflush(stdout);
            sleep(1);
        }
        printf("Pere >> Fin\n");
    }
    return 0;
}
```

Redirection de la sortie standard

```
[yann@siav ~/tmp]$ ./a.out
Processus pere de pid 17393
Fils --> Bonjour n = 7
Fils --> Bonjour n = 7
Fils --> Bonjour n = 7
Fils --> Bonjour n = 7
Fils --> Bonjour n = 7
Fils --> Pere >> Fin
 n = 12
Fils >> terminaison
```

Les tubes nommés

Les tubes nommés ou fifo ont été introduits dans la version III d'UNIX.

- Leur but est de permettre à des processus sans lien de parenté particulier de communiquer par l'intermédiaire de tubes.
- Ils ont toutes les caractéristiques des tubes et ont en plus une référence dans le système de fichiers.
- Tout processus connaissant la référence d'un tube peut l'ouvrir avec la primitive `open`.
- Les fichiers correspondant à des tubes sont identifies par `ls`.

```
[yann@siav ~/tmp]$ ls -l plop
prw-r--r--  1 yann yann 0 Jun  8 23:45 plop|
```

Création d'un tube nommé

La primitive :

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

- permet de creer un tube nommé ;
- `pathname` désigne le nom (nom relatif ou nom absolu) du fichier associé au tube ;
- `mode` désigne le mode du fichier ainsi créé (comme pour `open`) ;
- attention, le tube est simplement créé, il faut ensuite l'ouvrir pour pouvoir l'utiliser.

Utilisation d'un tube nommé

- Un processus connaissant le nom d'un tube peut l'ouvrir grâce à la primitive `open` (si les droits du tube le permettent).
- Attention, `open` est dans ce cas bloquant (une ouverture en lecture est bloquante tant qu'il n'y a aucun écrivain sur le tube et vice versa).
- Le descripteur de fichier obtenu par `open` peut ensuite être utilisé comme tout autre descripteur.
- Il ne faut pas oublier de supprimer les fichiers associés aux tubes lorsque l'on en a plus besoin :

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

permet de supprimer proprement `pathname` du système de fichiers (cf `man 2 unlink`).