Programmation Système
En *C* sous *Linux* (*Debian* et *Ubuntu*)

Rémy Malgouyres
LIMOS UMR 6158, IUT, département info
Université Clermont 1
B.P. 86
63172 AUBIERE cedex
http://malgouyres.org

Une version *HTML* de ce document est consultable sur :

http://malgouyres.org/programmation-systeme

Table des matières

Introduction

Ce cours porte sur l'utilisation des appels système des systèmes de la famille Unix : *Linux, MacOS X, AIX, LynxOS, BeOS, QNX, OpenBSD, FreeBSD,NetBSD.*

Le rôle d'un système est de (vois la figure 1) :

- Gérer le matériel :
  - Masquer le matériel en permettant aux programmes d'intéragir avec le matériel à travers des pilotes ;
  - Partager les ressources entre les différents programmes en cours d'exécution (processus).
- Fournir une interfaces pour les programmes (un ensemble d'appels système)
- Fournir une interfaces bas niveau pour l'utilisateur (un ensemble de commande shell)
- Eventuellement une interface utilisateur haut niveau par un environnement graphique (*kde*, *gnome*)...

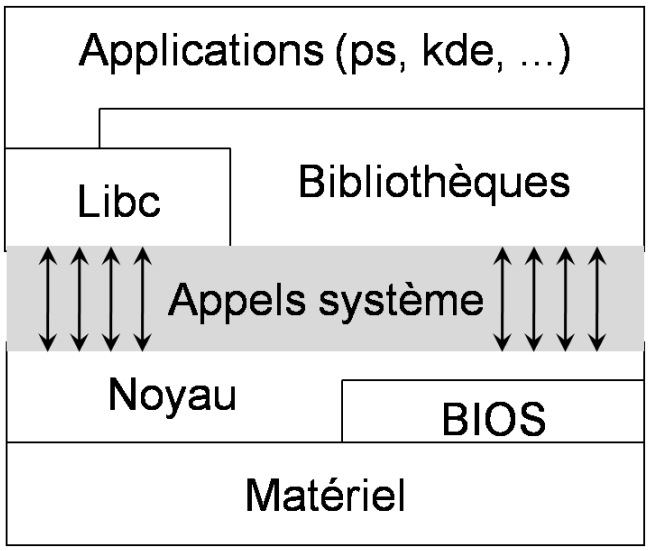

FIGURE 1 : Schéma d'un système d'exploitation de type Unix

La norme POSIX (Portable Operating System Interface uniX) est un ensemble de standards de l'IEEE (Institute of Electrical and Electronics Engineers). POSIX définit notamment :

- Les commandes shell de base (ksh, ls, man, ...)

- L'API (Application Programming Interface) des appels système.
- L'API des threads

Chapitre 1

Arguments d'un programme et variables d'environnement

Nous allons voir dans ce chapitre les passages d'arguments et variables d'environnement qui permettent à un shell de transmettre des informations à un programme qu'il lance. Plus généralement, ces techniques permettent à un programme de transmettre des informations aux programmes qu'il lance (processus fils ou descendants).

1.1 `atoi`, `sprinf` et `sscanf`

Parfois, un nombre nous est donné sous forme de chaîne de caractère dont les caractères sont des chiffres. Dans ce cas, la fonction `atoi` permet de réaliser la conversion d'une chaîne vers un `int`.

```
#include <stdio.h>

int main()
{
   int a;
   char s[50];

   printf("Saisissez des chiffres : ");
   scanf("%s", s);  \com{saisie d'une chaîne de caractères}
   a = atoi(s);  \com{conversion en entier}
   printf("Vous avez saisi : %d\n", a);
   return 0;
}
```

Plus généralement, la fonction `sscanf` permet de lire des données formatées dans une chaîne de caractère (de même que `scanf` permet de lire des données formatées au clavier ou `fscanf` dans un fichier texte).

```
#include <stdio.h>

int main()
```

```
{
   float x;
   char s[50];

   printf("Saisissez des chiffres (avec un point au milieu) : ");
   scanf("%s", s);  /* saisie d'une chaîne de caractères */
   sscanf(s, "%f", &x);  /* lecture dans la chaîne */
   printf("Vous avez saisi : %f\n", x);
   return 0;
}
```

Inversement, la fonction `sprintf` permet d'écrire des données formatées dans une chaîne de caractères (de même que `printf` permet d'écrire dans la console ou `fprintf` dans un fichier texte).

```
#include <stdio.h>

void AfficheMessage(char *message)
{
   puts(message);
}

int main()
{
   float x;
   int a;

   printf("Saisissez un entier et un réel : ");
   scanf("%d %f", &a, &x);
   sprintf(s, "Vous avez tapé : a = %d x = %f", a, x);
   AfficheMessage(s);
   return 0;
}
```

1.2 Arguments du main

La fonction `main` d'un programme peut prendre des arguments en ligne de commande. Par exemple, si un fichier `monprog.c` a permis de générer un exécutable `monprog` à la compilation,

```
gcc monprog.c -o monprog
```

on peut invoquer le programme `monprog` avec des arguments

```
./monprog argument1 argment2 argument3
```

Exemple. La commande `cp` du bash prend deux arguments :

```
$ cp nomfichier1 nomfichier2
```

Pour récupérer les arguments dans le programme *C*, on utilise les paramètres `argc` et `argv` du `main`. L'entier `argc` donne le nombre d'arguments rentrés dans la ligne de commande **plus 1**, et le paramètre `argv` est un tableau de chaînes de caractères qui contient comme éléments :

- Le premier élément `argv[0]` est une chaîne qui contient le nom du fichier executable du programme ;
- Les éléments suivants `argv[1]`, `argv[2]`, etc... sont des chaînes de caractères qui contiennent les arguments passés en ligne de commande.

Le prototype de la fonction `main` est donc :

```
int main(int argc, char**argv);
```

Exemple. Voici un programme longeurs, qui prend en argument des mots, et affiche la longueur de ces mots.

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

int main(int argc, char**argv)
{
   int i;
   printf("Vous avez entré %d mots\n", argc-1);
   puts("Leurs longueurs sont :");
   for (i=1 ; i<argc ; i++)
      {
         printf("%s : %d\n", argv[i], strlen(argv[i]));
      }
   return 0;
}
```

Voici un exemple de trace :

```
$ gcc longueur.c -o longueur
$ ./longueur toto blabla
Vous avez entré 2 mots
Leurs longueurs sont :
toto : 4
blabla : 6
```

1.3 Variables d'environnement

1.3.1 Rappels sur les variables d'environnement

Les variables d'environnement sont des affectations de la forme

```
NOM=VALEUR
```

qui sont disponibles pour tous les processus du système, y compris les shells. Dans un shell, on peut avoir la liste des varialbles d'environnement par la commande `env`. Par exemple, pour la variable d'environnement `PATH` qui contient la liste des répertoires où le shell va chercher les commandes exécutables :

```
$ echo PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11
$ PATH=PATH:.
$ export PATH
$ env | grep PATH
PATH=/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:.
```

la commande `export` permet de transmettre la valeur d'une variable d'environnement aux descendants d'un shell (programmes lancés à partir du shell).

1.3.2 Accéder aux variables d'environnement en C

Dans un programme C, on peut accéder à la liste des variables d'environnement dans la variable `environ`, qui est un tableau de chaînes de caractères (terminé par un pointeur `NULL` pour marquer la fin de la liste).

```
#include <stdio.h>

extern char **environ;

int main(void)
{
   int i;
   for (i=0 ; environ[i]!=NULL ; i++)
      puts(environ[i]);
   return 0;
}
```

Pour accéder à une variable d'environnement particulière à partir de son nom, on utilise la fonction `getenv`, qui prend en paramètre le nom de la variable et qui retourne sa valeur sous forme de chaîne de caractère.

```
#include <stdio.h>
#include <stdlib.h>  /* pour utiliser getenv */

int main(void)
{
   char *valeur;
   valeur = getenv("PATH");
   if (valeur != NULL)
      printf("Le PATH vaut : %s\(\backslash\)n", valeur);
   valeur = getenv("HOME");
   if (valeur != NULL)
```

```
      printf("Le home directory est dans %s\(\backslash\)n", valeur);
   return 0;
}
```

Pour assigner une variable d'environnement, on utilise la fonction `putenv`, qui prend en paramètre une chaîne de caractère. Notons que la modification de la variable ne vaut que pour le programme lui-même et ses descendants (autres programmes lancés par le programme), et ne se transmet pas au shell (ou autre) qui a lancé le programme en cours.

```
#include <stdio.h>
#include <stdlib.h>  /* pour utiliser getenv */

int main(void)
{
   char *path, *home, *nouveaupath;
   char assignation[150];
   path = getenv("PATH");
   home = getenv("HOME");
   printf("ancien PATH : %s\net HOME : %s\n",
          path, home);
   sprintf(assignation, "PATH=%s:%s/bin", path, home);
   putenv(assignation);
   nouveaupath = getenv("PATH");
   printf("nouveau PATH : \n%s\n", nouveaupath);
   return 0;
}
```

Exemple de trace :

```
$ gcc putenv.c -o putenv
 echo PATH
$ /usr/local/bin:/usr/bin:/bin:/usr/bin/X11
$ ./putenv
ancien PATH : /usr/local/bin:/usr/bin:/bin:/usr/bin/X11
et HOME : /home/remy
nouveau PATH :
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/home/remy/bin
 echo PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11
```

Chapitre 2

Processus

2.1 Processus, *PID*, *UID*

2.1.1 Processus et *PID*

Chaque programme (fichier exécutable ou script *shell*,*Perl*) en cours d'exécution dans le système coorespond à un (ou parfois plusieurs) *processus* du système. Chaque processus possède un *numéro de processus* (PID).

Sous unix, on peut voir la liste des processus en cours d'exécution, ainsi que leur *PID*, par la commande `ps`, qui comporte différentes options.

Pour voir ses propres processus en cours d'exécution on peut utiliser le commande

```
$ ps x
```

Pour voir l'ensemble des processus du système, on peut utiliser la commande

```
$ ps -aux
```

Pour voir l'ensemble des attributs des processus, on peut utiliser l'option `-f`. Par exemple, pour l'ensemble des attributs de ses propres processus, on peut utiliser

```
$ ps -f x
```

Un programme C peut accéder au *PID* de son instance en cours d'exécusion par la fonction `getpid`, qui retourne le *PID* :

```
pid_t getpid(\void);
```

Nous allons voir dans ce chapitre comment un programme C en cours d'exécution peut créer un nouveau processus (fonction `fork`), puis au chapitre suivant comment un programme C en cours d'exécution peut se faire remplacer par un autre programme,tout en gardant le même numéro de processus (fonction `exec`). L'ensemble de ces deux fonction permettra à un programme C de lancer un autre programme. Nous verrons ensuite la fonction `system`, qui permet directement de lancer un autre programme, ainsi que les problèmes de sécurité liés à l'utilisation de cette fonction.

2.1.2 Privilèges, *UID*,*Set-UID*

Chaque processus possède aussi un *User ID*, noté *UID*, qui identifie l'utilisateur qui a lancé le processus. C'est en fonction de l'*UID* que le processus se voit accordé ou refuser les droits d'accès en lecture, écritue ou exécution à certains fichiers ou à certaines commandes. On ficxe les droits d'accès d'un fichier avec la commande `chmod`. L'utilisateur `root` possède un *UID* égal à 0. Un programme C peut accéder à l'*UID* de son instance en cours d'exécution par la fonction *getuid* :

```
uid_t getuid(\void);
```

Il existe une permission spéciale, uniquement pour les exécutables binaires, appelée la permission *Set-UID*. Cette permission permet à un ustilisateur ayant les droits en exécution sur le fichier dexécuter le fichier **avec les privilège du propriétaire du fichier**. On met les droits *Set-UID* avec `chmod +s`.

```
$ chmod +x fichier
$ ls -l
-rwxr-xr-x 1 remy remy  7145 Sep  6 14:04 fichier
$ chmod +s fichier
-rwsr-sr-x 1 remy remy  7145 Sep  6 14:05 fichier
```

2.2 La fonction `fork`

La fonction `fork` permet à un programme en cours d'exécution de créer un nouveau processus. Le processus d'origine est appelé *processus père*, et il garde son *PID*, et le nouveau processus créé s'appelle *processus fils*, et possède un nouveau *PID*. Le processus père et le processus fils ont le même code source, mais la valeur retournée par fork permet de savoir si on est dans le processus père ou fils. Ceci permet de faire deux choses différentes dans le processus père et dans le processus fils (en utilisant un `if` et un `else` ou un `switch`), même si les deux processus on le même code source.

La fonction `fork` retourne -1 en cas d'erreur, retourne 0 dans le processus fils, et retourne le *PID* du fils dans le processus père. Ceci permet au père de connaître le *PID* de son fils.

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

int main(void)
{
  pid_t pid_fils;

  pid_fils = fork();
  if (pid_fils == -1)
  {
    puts("Erreur de création du nouveau processus");
    exit (1);
  }
```

```
  if (pid_fils == 0)
  {
    printf("Nous sommes dans le fils\n");
    /* la fonction getpid permet de connaître son propre PID */
    printf("Le PID du fils est \%d\n", getpid());
    /* la fonction getppid permet de connaître le PPID
    (PID de son père) */
    printf("Le PID de mon père (PPID) est \%d", getppid());
  }
  else
  {
    printf("Nous sommes dans le père\n");
    printf("Le PID du fils est \%d\n", pid_fils);
    printf("Le PID du père est \%d\n", getpid());
    printf("PID du grand-père : \%d", getppid());
  }
  return 0;
}
```

2.3 Terminaison d'un processus fils

Lorsque le processus fils se termine (soit en sortant du `main` soit par un appel à `exit`) avant le processus père, le processus fils ne disparaît pas complètement, mais devient un *zombie*. Pour permettre à un processus fils à l'état de zombie de disparaître complètement, le processus père peut appeler l'instruction suivante qui se trouve dans la bibliothèque `sys/wait.h` :

```
wait(NULL);
```

Cependant, il faut prendre garde l'appel de `wait` est bloquant, c'est à dire que lorsque la fonction `wait` est appelée, l'exécution du père est suspendue jusqu'à ce qu'un fils se termine. De plus, **il faut mettre autant d'appels de `wait` qu'il y a de fils**. La fonction `wait` renvoie le code d'erreur -1 dans le cas où le processus n'a pas de fils.

La fonction `wait` est fréquement utilisée pour permettre au processus père d'attendre la fin de ses fils avnt de se terminer lui-même, par exemple pour récupérer le résultat produit par un fils.

Il est possible de mettre le processus père en attente de la fin d'un processus fils particulier par l'instruction

```
waitpid(pid_fils, NULL, 0);
```

Le paramètre de la fonction `wait`, et de deuxième paramètre de `waitpid` est un passage par adresse d'un entier qui donne des informations sur le statut du fils lorsqu'il se termine : terminaison normale, avortement (par exemple par `ctrl-C`) ou processus temporairement stoppé).
Exemple. Voici un exemple où le père récupère le code renvoyé par le fils dans la fonction `exit`.

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

```
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h> /* permet de récupérer les codes d'erreur */

pid_t pid_fils

int main(void)
{
   int status;
   switch (pid_fils=fork())
   {
      case -1 : perror("Problème dans fork()\n");
                exit(errno); /* retour du code d'erreur */
                break;

      case 0 :  puts("Je suis le fils");
                puts("Je retourne le code 3");
                exit(3);

      default : puts("Je suis le père");
                puts("Je récupère le code de retour");
                wait(&status);
                printf("code de sortie du fils %d : %d\n",
                       pid_fils, WEXITSTATUS(status));
                break;
   }
   return 0;
}
```

La trace de ce programme est la suivante :

```
Je suis le fils
Je retourne le code 3
Je suis le père
Je récupère le code de retour
code de sortie : 3
```

2.4 Exercices

Exercice 2.1 (E) crire un programme qui crée un fils. Le père doit afficher “je suis le père” et le fils doit afficher “je suis le fils”.

Exercice 2.2 (E) crire un programme qui crée deux fils appelés fils 1 et fils 2. Le père doit afficher “je suis le père” et le fils 1 doit afficher “je suis le fils 1”, et le fils 2 doit afficher “je suis le fils 2”.

Exercice 2.3 (E) crire un programme qui crée 5 fils en utilisant une boucle `for`. On remarquera que pour que le fils ne crée pas lui-même plusieurs fils, il faut interompre la boucle par un `break` dans le fils.

Exercice 2.4 (E) crire un programme avec un processus père qui engendre 5 fils dans une bouble for. Les fils sont nommés fils 1 à fils 5. Le fils 1 doit afficher "je suis le fils 1" et le fils 2 doit afficher je suis le fils 2, et ainsi de suite.
Indication. on pourra utiliser une variable globale.

Exercice 2.5 (E) crire un programme qui crée deux fils appelés fils 1 et fils 2. Chaque fils doit attendre un nombre de secondes aléatoire entre 1 et 10, en utilisant la fonction `sleep`. Le programme attend que le fils le plus long se termine et affiche la durée totale. On pourra utiliser la fonction `time` de la bibliothèque `time.h`, qui retourne le nombre de secondes depuis le premier janvier 1970 à 0h (en temps universel).

Chapitre 3

Lancement d'un programme : `exec`

3.1 Rappels : Arguments en ligne de commande

La fonction `main` d'un programme peut prendre des arguments en ligne de commande. Par exemple, si un fichier `monprog.c` a permis de générer un exécutable `monprog` à la compilation,

```
$ gcc monprog.c -o monprog
```

on peut invoquer le programme `monprog` avec des arguments

```
$ ./monprog argument1 argment2 argument3
```

Exemple. La commande `cp` du bash prend deux arguments :

```
$ cp nomfichier1 nomfichier2
```

Pour récupérer les arguments dans le programme C, on utilise les paramètres `argc` et `argv` du `main`. L'entier `argc` donne le nombre d'arguments rentrés dans la ligne de commande **plus 1**, et le paramètre `argv` est un tableau de chaînes de caractères qui contient comme éléments :

- Le premier élément `argv[0]` est une chaîne qui contient le nom du fichier executable du programme ;
- Les éléments suivants `argv[1]`, `argv[2]`,etc... sont des chaînes de caractères qui contiennent les arguments passés en ligne de commande.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   int i;
   if (argc == 1)
      puts("Le programme n'a reçu aucun argument");
   if (argc >= 2)
      {
          puts("Le programme a reçu les arguments suivants :");
          for (i=1 ; i<argc ; i++)
             printf("Argument %d = %s\n", i, argv[i]);
      }
   return 0;
}
```

3.2 L'appel système exec

3.2.1 Arguments en liste

L'appel système `exec` permet de remplacer le programme en cours par un autre programme sans changer de numéro de processus (PID). Autrement dit, un programme peut se faire remplacer par un autre code source ou un script shell en faisant appel à `exec`. Il y a en fait plusieurs fonctions de la famille `exec` qui sont légèrement différentes.

La fonction `execl` prend en paramètre une *liste* des arguments à passer au programme (liste terminée par `NULL`).

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

int main()
{
   /* dernier élément NULL, OBLIGATOIRE */
   execl("/usr/bin/emacs", "emacs", "fichier.c", "fichier.h", NULL);

   perror("Problème : cette partie du code ne doit jamais être exécutée");
   return 0;
}
```

Le premier paramètre est une chaîne qui doit contenir le chemin d'accès complet (dans le système de fichiers) au fichier exécutable ou au script shell à exécuter. Les paramètres suivants sont des chaînes de caractère qui représentent les arguments passés en ligne de commande au `main` de ce programme. La chaîne `argv[0]` doit donner le nom du programme (sans chemin d'accès), et les chaînes suivantes `argv[1], argv[2],etc...` donnent les arguments.

Concernant le chemin d'accès, il est donné à partir du répertoire de travail ($\$PWD$), ou à partir du répertoire racine / s'il commence par le caractère /
(exemple : `/home/remy/enseignement/systeme/script1`).

La fonction `execlp` permet de rechercher les exécutables dans les répertoires apparaîssant dans le `PATH`, ce qui évite souvent d'avoir à spécifier le chemin complet.

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

int main()
{
   /* dernier élément NULL, OBLIGATOIRE */
   execlp("emacs", "emacs", "fichier.c", "fichier.h", NULL);

   perror("Problème : cette partie du code ne doit jamais être exécutée");
   return 0;
}
```

3.2.2 Arguments en vecteur

Nous allons étudier l'une d'entre elles (la fonction `execv`). La différence avec `execl` est que l'on n'a pas besoin de conaître la liste des arguments à l'avance (ni même leur nombre). Cette fonction a pour prototype :

```
int execv(const char* application, const char* argv[]);
```

Le mot `const` signifie seulement que la fonction `execv` ne modifie pas ses paramètres. Le premier paramètre est une chaîne qui doit contenir le chemin d'accès (dans le système de fichiers) au fichier exécutable ou au script shell à exécuter. Le deuxième paramètre est un tableau de chaînes de caractères donnant les arguments passés au programme à lancer dans un format similaire au paramètre `argv` du `main` de ce programme. La chaîne `argv[0]` doit donner le nom du programme (sans chemin d'accès), et les chaînes suivants `argv[1], argv[2],etc...` donnent les arguments.

Le dernier élément du tableau de pointeurs argv doit être NULL pour marquer la fin du tableau. Ceci est dû au fait que l'on ne passe pas de paramètre `argc` donnant le nombre d'argument

Concernant le chemin d'accès, il est donné à partir du répertoire de travail ($PWD), ou à partir du répertoire racine / s'il commence par le caractère /
(exemple : `/home/remy/enseignement/systeme/script1`).

Exemple. Le programme suivant édite les fichiers .c et .h du répertoire de travail avec `emacs`.

Dans le programme, le chemin d'accès à la commande `emacs` est donné à partir de la racine `/usr/bin/emacs`.

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

int main()
{
   char * argv[] = {"emacs", "fichier.c", "fichier.h", NULL}
   /* dernier élément NULL, obligatoire */
   execv("/usr/bin/emacs", argv);

   puts("Problème : cette partie du code ne doit jamais être exécutée");
   return 0;
}
```

Remarque 3.2.1 *Pour exécuter un script shell avec* `execv`*, il faut que la première ligne de ce script soit*

```
#! /bin/sh
```

ou quelque chose d'analogue.

En utilisant `fork`, puis en faisant appel à `exec` dans le processus fils, un programme peut lancer un autre programme et continuer à tourner dans le processus père.

Il existe une fonction `execvp` qui lance un programme en le recherchant dans la variable d'environnement PATH. L'utilisation de cette fonction dans un programme *Set-UID* pose des problèmes de sécurité (voir explications plus loin pour la fonction `system`

3.3 La fonction `system`

3.3.1 La variable PATH dans unix

La variable d'environnement PATH sous unix et linux donne un certain nombre de chemins vers des répertoires où se trouve les exécutables et scripts des commandes. Les chmins dans le PATH sont séparés par des ' :'.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games:/home/remy/bin:.
```

Lorsqu'on lance une commande dans une console, le système va cherche l'exécutable ou le script de cette commande dans les répertoires donnés dans le PATH. Chaque utilisateur peut rajouter des chemins dans son PATH (en modifiant son fichier .bashrc sous linux). En particulier, l'utilisateur peut rajouter le répertoire '.' (point) dans le PATH, ce qui signifie que le système va chercher les commandes dans le répertoire de travail donné dans la variable d'environnement PWD. La recherche des commandes dans les répertoires a lieu dans l'ordre dans lequel les répertoires apparaîssent dans le PATH. Par exemple, pour le PATH donné ci-dessus, la commande sera recherchée d'abord dans le répertoire `/usr/local/bin`, puis dans le répertoire `/usr/bin`. Si deux commandes de même nom se trouve dans deux répertoires du PATH, c'est la première commande trouvée qui sera exécutée.

3.3.2 La fonction `system`

La fonction `system` de la bibliothèque stdlib.h permet directement de lancer un programme dans un programme C sans utiliser `fork` et `exec`. Pour celà, on utilise l'instruction :

```
#include <stdlib.h>
...
system("commande");
```

Exemple. La commande unix `clear` permet d'effacer la console. Pour effacer la console dans un programme C avec des entrées-sorties dans la console, on peut ustilser :

```
system("clear");
```

Lorsqu'on utilise la fonction `system`, la commande qu'on exécute est recherchée dans les répertoires du PATH comme si l'on exécutait la commande dans la console.

3.4 Applications suid et problèmes des sécurité liés system, execlp ou exevp

Dans le système unix, les utilisateurs et l'administrateur (utilisateur) on des droits (que l'on appelle privilèges), et l'accès à certaines commandes leur sont interdites. C'est ainsi que, par exemple, si le système est bien administré, un utilisateur ordinaire ne peut pas facilement endomager le système.

Exemple. Imaginons que les utilisateurs aient tous les droits et qu'un utilisateur malintentioné ou distrait tape la commande

```
$ rm -r /
```

Cela supprimerait tous les fichiers du système et des autres utilisateurs et porterait un préjudice important pour tous les utilisateurs du système. En fait, beaucoup de fichiers sont interdits à l'utilisateur en écriture, ce qui fait que la commande `rm` sera ineffective sur ces fichiers.

Pour celà, lorsque l'utilisateur lance une commande ou un script (comme la commande `rm`), les privilèges de cet utilsateur sont pris en compte lors de l'exécution de la commande.

Sous unix, un utilisateur A (par exemple root) peut modifier les permissions sur un fichier exécutable pour que tout autre utilisateur B puisse exécuter ce fichier avec ses propres privilèges (les privilèges de A). Cela s'appelle les permissions suid.

Exemple. Supposons que l'utilisateur `root` tape les commandes suivantes :

```
$ gcc monprog.c -o monprog
$ ls -l
-rwxr-xr-x  1 root root 18687 Sep  7 08:28 monprog
-rw-r--r--  1 root root  3143 Sep  4 15:07 monprog.c
$ chmod +s monprog
$ ls -l
-rwsr-sr-s  1 root root 18687 Sep  7 08:28 monprog
-rw-r--r--  1 root root  3143 Sep  4 15:07 monprog.c
```

Le programme `moprog` est alors suid et n'importe quel utilisateur peut l'exécuter avec les privilèges du propriétaire de monprog, c'est à dire `root`.

Supposons maintenant que dans le fichier `monprog.c` il y ait l'instruction

```
system("clear");
```

Considérons un utilisateur malintentionné `remy`. Cet utilisateur modifie son PATH pour rajouter le répertoire '.', (point) mais met le répertoire '.' au tout début de PATH

```
$ PATH=.:\PATH
$ export PATH
$ echo \PATH
.:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games:/home/remy/bin:.
```

Dans la recherche des commandes dans les répertoires du PATH, le système cherchera d'abord les commandes dans le répertoire de travail '.'. Supposons maintenant que l'utilisateur `remy` crée un script appelé `clear` dans son répertoire de travail. qui contienne la ligne
`rm -r /`

```
$ echo "rm -r /" > clear
$ cat clear
rm -r /
$ chmod +x clear
$ monprog
```

Lorsque l'utilisateur `remy` va lancer l'exécutable `monprog` avec les privilèges de `root`, le programme va exécuter le script `clear` de l'utilisateur (au lieu de la commande `/usr/bin/clear`) avec les privilèges de `root`, et va supprimer tous les fichiers du système.

Il ne faut jamais utiliser la fonction `system` ou la fonction `execvp` dans une application suid, car un utilisateur malintentioné pourrait exécuter n'importe quel script avec vos privilèges.

3.5 Exercices

Exercice 3.1 $(*)$ Écrire un programme qui prend deux arguments en ligne de commande en supposant qu ce sont des nombres entiers, et qui affiche l'addition de ces deux nombres.

Exercice 3.2 $(*)$ Ecrire un programme qui prend en argument un chemin vers un répertoire R, et copie le répertoire courant dans ce répertoire R.

Exercice 3.3 $(*)$ Ecrire un programme qui saisit un nom de fichier texte au clavier et ouvre ce fichier dans l'éditeur `emacs`, dont le fichier exécutable se trouve à l'emplacement `/usr/bin/emacs`.

Exercice 3.4 $(**)$ Ecrire un programme qui saisit des noms de répertoires au clavier et copie le répertoire courant dans tous ces répertoires. Le programme doit se poursuivre jusqu'à ce que l'utilisateur demande de quitter le programme.

Exercice 3.5 $(**)$ Ecrire un programme qui saisit des nom de fichiers texte au clavier et ouvre tous ces fichiers dans l'éditeur `emacs`. Le programme doit se poursuivre jusqu'à ce que l'utilisateur demande de quitter.

Exercice 3.6 $(***)$ Considérons les coefficients binômiaux C_n^k tels que

$$C_i^0 = 1 \text{ et } C_i^i = 1 \text{ pour tout } i$$

$$C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$$

Écrire un programme pour calculerC_n^k qui n'utilise aucune boucle (ni `while` ni `for`), et qui n'ait comme seule fonction que la fonction `main`. La fonction `main` ne doit contenir aucun appel à elle-même.On pourra utiliser des fichiers textes temporaires dans le répertoire `/tmp`.

Chapitre 4

Communication entre processus

Dans ce chapitre, nous voyons comment faire communiquer des processus entre eux par des tubes. Pour le moment, les processus qui communiquent doivent être des processus de la même machine. Cependant, le principe de communication avec les fonctions `read` et `write` sera réutilisé par la suite lorsque nous aborderons la programmation réseau, qui permet de faire communiquer des processus se trouvant sur des stations de travail distinctes.

4.1 Tubes et `fork`

Un tube de communication est un tuyau (en anglais *pipe*) dans lequel un processus peut écrire des données et un autre processus peut lire. On crée un tube par un appel à la fonction `pipe`, déclarée dans `unistd.h` :

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

La fonction renvoie 0 si elle réussit, et elle crée alors un nouveau tube. La fonction `pipe` remplit le tableau `descripteur` passé en paramètre, avec :

- `descripteur[0]` désigne la sortie du tube (dans laquelle on peut lire des données) ;
- `descripteur[1]` désigne l'entrée du tube (dans laquelle on peut écrire des données) ;

Le principe est qu'un processus va écrire dans `descripteur[1]` et qu'un autre processus va lire les mêmes données dans `descripteur[0]`. Le problème est qu'on ne crée le tube dans un seul processus, et un autre processus ne peut pas deviner les valeurs du tableau `descripteur`. Pour faire communiquer plusieurs processus entre eux, il faut appeler la fonction `pipe` avant d'appeler la fonction `fork`. Ensuite, le processus père et le processus fils auront les mêmes descripteurs de tubes, et pourront donc communiquer entre eux. De plus, un tube ne permet de communiquer que dans un seul sens. Si l'on souhaite que les processus communiquent dans les deux sens, il faut créer deux pipes.

Pour écrire dans un tube, on utilise la fonction `write` :

```
ssize_t write(int descripteur1, const void *bloc, size_t taille);
```

Le descripteur doit crrespondre à l'entrée d'un tube. La taille est le nombre d'octets qu'on souhaite écrire, et le bloc est un pointeur vers la mémoire contenant ces octets.

Pour lire dans un tube, on utilise la fonction `read` :

```
ssize_t read(int descripteur0, void *bloc, size_t taille);
```

Le descripteur doit correspondre à la sortie d'un tube, le bloc pointe vers la mémoire destinée à recevoir les octets, et la taille donne le nombre d'octets qu'on souhaite lire. La fonction renvoie le nombre d'octets effectivement lus. Si cette valeur est inférieure à `taille`, c'est qu'une erreur s'est produite en cours de lecture (par exemple la fermeture de l'entrée du tube suite à la terminaison du processus qui écrit).

Dans la pratique, on peut transmettre un buffer qui a une taille fixe (256 octets dans l'exemple ci-dessous). L'essentiel est qu'il y ait exactement le même nombre d'octets en lecture et en écriture de part et d'autre du pipe. La partie significative du buffer est terminée par un $'\backslash 0'$ comme pour n'importe quelle chaîne de caractère.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

#define BUFFER_SIZE 256

int main(void)
{
  pid_t pid_fils;
  int tube[2];
  unsigned char bufferR[256], bufferW[256];

  puts("Cr ation d'un tube");
  if (pipe(tube) != 0)   {/* pipe */
    {
      fprintf(stderr, "Erreur dans pipe\(\backslash\)n");
      exit(1);
    }
  pid_fils = fork();     {/* fork */
  if (pid_fils == -1)
    {
      fprintf(stderr, "Erreur dans fork\(\backslash\)n");
      exit(1);
    }
  if (pid_fils == 0) {/* processus fils */
    {
     printf("Fermeture entr e dans le fils (pid = %d)\(\backslash\)n", getpid());
      close(tube[1]);
      read(tube[0], bufferR, BUFFER_SIZE);
      printf("Le fils (%d) a lu : %s\(\backslash\)n", getpid(), bufferR);
    }
  else               {/* processus p re */
    {
     printf("Fermeture sortie dans le p re (pid = %d)\(\backslash\)n", getpid());
```

```
      close(tube[0]);
      sprintf(bufferW, "Message du p re (%d) au fils", getpid());
      write(tube[1], bufferW, BUFFER_SIZE);
      wait(NULL);
    }
  return 0;
}
```

La sortie de ce programme est :

```
Création d'un tube
Fermeture entrée dans le fils (pid = 12756)
Fermeture sortie dans le père (pid = 12755)
Ecriture de 31 octets du tube dans le père
Lecture de 31 octets du tube dans le fils
Le fils (12756) a lu : Message du père (12755) au fils
```

Il faut noter que les fonctions `read` et `write` permettent de transmettre uniquement des tableaux des octets. Toute donnée (nombre ou texte) doit être convertie en tableau de caractère pour être transmise, et la taille des ces données doit être connue dans les deux processus communiquants.

4.2 Transmettre des données binaires

Voici un exemple de programme qui saisit une valeur x au clavier dans le processus père, et transmet le sinus de ce nombre, **en tant que `double`** au processus fils.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>
#include <math.h>

int main(\void)
{
  pid_t pid_fils;
  int tube[2];
  double x, valeurW, valeurR;

  puts("Création d'un tube");
  if (pipe(tube) != 0)   /* pipe */
    {
      fprintf(stderr, "Erreur dans pipe\n");
      exit(1);
    }
  switch(pid_fils = fork())     /* fork */
```

```
  {
     case -1 :
        perror("Erreur dans fork\n");
        exit(errno);
     case 0 : /* processus fils */
        close(tube[1]);
        read(tube[0], &valeurR, sizeof(double));
        printf("Le fils (%d) a lu : %.2f\(\backslash\)n", getpid(), valeurR);
        break;
     default :    /* processus père */
      printf("Fermeture sortie dans le père (pid = %d)\n", getpid());
      close(tube[0]);
      puts("Entrez x :");
      scanf("%lf", &x);
      valeurW = sin(x);
      write(tube[1], &valeurW, \sizeof(\double));
      wait(NULL);
      break;
  }
  return 0;
}
```

Compléments

√ D'une manière générale, l'adresse d'une variable peut toujours être considérée comme un tableau dont le nombre d'octet est égal à la taille de cette variable. Ceci est dû au fait qu'un tableau est seulement une adresse qui pointe vers une zone mémoire réservée pour le programme (statiquement ou dynamiquement).

4.3 Rediriger les flots d'entrées-sorties vers des tubes

On peut lier la sortie `tube[0]` du tube à `stdin`. Par la suite, tout ce qui sort du tube arrive sur le flot d'entrée standard `stdin`, et peut être lu avec `scanf`, `fgets`, etc... Pour celà, il suffit de mettre l'instruction :

```
dup2(tube[0], STDIN_FILENO);
```

De même, on peut lier l'entrée `tube[1]` du tube à `stdout`. Par la suite, tout ce qui sort sur le flot de sortie standard `stdout` entre ans le tube, et on peut écrire dans le tube avec `printf`, `puts`, etc... Pour celà, il suffit de mettre l'instruction :

```
dup2(tube[1], STDOUT_FILENO);
```

Compléments

√ Plus généralement, la fonction `dup2` copie le descripteur de fichier passé en premier argument dans le descripteur passé en deuxième argument.

4.4 Tubes nommés

On peut faire communiquer deux processus à traver un tube nommé. Le tube nommé passe par un fichier sur le disque. L'intérêt est que **les deux processus n'ont pas besoin d'avoir un lien de parenté**. Pour créer un tube nommé, on utilise la fonction `mkfifo` de la bibliothèque `sys/stat.h`.
Exemple. Dans le code suivant, le premier programme transmet le mot "coucou" au deuxième programme. Les deux programmes n'ont pas besoin d'être liés par un lien de parenté.

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

int main()
{
   int fd;
   FILE *fp;
   char *nomfich="/tmp/test.txt";   /* nom du fichier */
   if(mkfifo(nomfich, 0644) != 0)  /* création du fichier */
   {
      perror("Problème de création du noeud de tube");
      exit(1);
   }
   fd = open(nomfich, O_WRONLY);  /* ouverture en écriture */
   fp=fdopen(fd, "w");  /* ouverture du flot */
   fprintf(fp, "coucou\(\backslash\)n"); /* écriture dans le flot */
   unlink(nomfich); /* fermeture du tube */
   return 0;
}
```

La fonction `mkfifo` prend en paramètre, outre le chemin vers le fichier, le masque des permissions (lecture, écriture) sur la structure *fifo*.

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

int main()
{
   int fd;
   FILE *fp;
   char *nomfich="/tmp/test.txt", chaine[50];
   fd = open(nomfich, O_RDONLY);  /* ouverture du tube */
   fp=fdopen(fd, "r");  /* ouverture du flot */
```

```
   fscanf(fp, "%s", chaine);  /* lecture dans le flot */
   puts(chaine);  /* affichage */
   unlink(nomfich);  /* fermeture du flot */
   return 0;
}
```

4.5 Exercices

Exercice 4.1 ($exo_basepipe$) Écrire un programme qui crée deux processus. Le processus père ouvre un fichier texte en lecture. On suppose que le fichier est composé de mots formés de caractères alphabétiques séparés par des espaces. Le processus fils saisit un mot au clavier. Le processus père recherche le mot dans le fichier, et transmet au fils la valeur 1 si le mot est dans le fichier, et 0 sinon. Le fils affiche le résultat.

Exercice 4.2 (R) eprendre les programmes de l'exercice **??**. Nous allons faire un programme qui fait la même chose, mais transmet les données différement. Dans le programme père, on liera les flots `stdout` et `stdin` à un tube.

Exercice 4.3 (exo_tube_exec) Écrire un programme qui crée un tube, crée un processus fils, puis, dans le fils, lance par `execv` un autre programme, appelé programme fils. Le programme père transmets les descripteurs de tubes au programmes fils en argument, et transmet un message au fils par le tube. Le programme fils affiche le message.

Exercice 4.4 (M) ême question qu'à l'exercice **??** mais en passant les descripteurs de tube comme variables d'environnement.

Chapitre 5

Threads Posix

5.1 Pointeurs de fonction

Un pointeur de fonctions en C est une variable qui permet de désigner une fonction C. Comme n'importe quelle variable, on peut mettre un pointeur de fonctions soit en variable dans une fonction, soit en paramètre dans une fonction.

On déclare un pointeur de fonction comme un prototype de fonction, mais on ajoute une étoile ($*$) devant le nom de la fonction. Dans l'exemple suivant, on déclare dans le main un pointeur sur des fonctions qui prennent en paramètre un `int`, et un pointeur sur des fonctions qui retournent un `int`.

```
#include <stdio.h>

int SaisisEntier(void)
{
   int n;
   printf("Veuillez entrer un entier : ");
   scanf("%d", &n);
   return n;
}

void AfficheEntier(int n)
{
   printf("L'entier n vaut %d\(\backslash\)n", n);
}

int main(void)
{
  void (*foncAff)(int); {/* d claration d'un pointeur foncAff */
  int (*foncSais)(void); {/*d claration d'un pointeur foncSais */
  int entier;

  foncSais = SaisisEntier;  {/* affectation d'une fonction */
  foncAff = AfficheEntier; {/* affectation d'une fonction */
```

```
  entier = foncSais(); {/* on ex cute la fonction */
  foncAff(entier);   {/* on ex cute la fonction */
  return 0;
}
```

Dans l'exemple suivant, la fonction est passée en paramètre à une autre fonction, puis exécutée.

```
#include <stdio.h>

int SaisisEntier(void)
{
   int n;
   printf("Veuillez entrer un entier : ");
   scanf("%d", &n);
   getchar();
   return n;
}

void AfficheDecimal(int n)
{
   printf("L'entier n vaut %d\(\backslash\)n", n);
}

void AfficheHexa(int n)
{
   printf("L'entier n vaut %x\(\backslash\)n", n);
}

void ExecAffiche(void (*foncAff)(int), int n)
{
  foncAff(n);  {/* ex cution du param tre */
}

int main(void)
{
  int (*foncSais)(void); {/*d claration d'un pointeur foncSais */
  int entier;
  char rep;

  foncSais = SaisisEntier;  {/* affectation d'une fonction */
  entier = foncSais(); {/* on ex cute la fonction */

  puts("Voulez-vous afficher l'entier n en d cimal (d) ou en hexa (x) ?");
  rep = getchar();
  {/* passage de la fonction en param tre : */
  if (rep == 'd')
```

```
    ExecAffiche(AfficheDecimal, entier);
  if (rep == 'x')
    ExecAffiche(AfficheHexa, entier);

  return 0;
}
```

Pour prévoir une utilisation plus générale de la fonction `ExecAffiche`, on peut utiliser des fonctions qui prennent en paramètre un `void*` au lieu d'un `int`. Le `void*` peut être ensuite reconverti en d'autres types par un *cast*.

```
void AfficheEntierDecimal(void *arg)
{
   inte n = (int)arg; /* un void* et un int sont sur 4 octets */
   printf("L'entier n vaut \%d\n", n);
}

void ExecFonction(void (*foncAff)(void* arg), void *arg)
{
  foncAff(arg);  /* exécution du paramètre */
}

int main(void)
{
   int n;
   ...
   ExecFonction(AfficheEntierDecimal, (void*)n);
   ...
}
```

On peut utiliser la même fonction `ExecFonction` pour afficher tout autre chose que des entiers, par exemple un tableau de `float`.

```
typedef struct
   {
      int n;  /* nombre d'éléments du tableau */
      double *tab;  /* tableau de double */
   }TypeTableau;

void AfficheTableau(void *arg)
{
   inte i;
   TypeTableau *T = (TypeTableau*)arg; /* cast de pointeurs */
   for (i=0 ; i<T->n ; i++)
      {
          printf("%.2f", T->tab[i]);
      }
}
```

```
void ExecFonction(void (*foncAff)(void* arg), void *arg)
{
  foncAff(arg);  /* exécution du paramètre */
}

int main(void)
{
   TypeTableau tt;
   ...
   ExecFonction(AfficheTableau, (void*)&tt);
   ...
}
```

5.2 Thread Posix (sous linux)

5.2.1 Qu'est-ce qu'un thread ?

Un *thread* (ou *fil d'exécution* en français) est une parie du code d'un programme (une fonction), qui se déroule parallèlement à d'autre parties du programme. Un premier interêt peut être d'effectuer un calcul qui dure un peu de temps (plusieurs secondes, minutes, ou heures) sans que l'interface soit bloquée (le programme continue à répondre aux signaux). L'utilisateur peut alors intervenir et interrompre le calcul sans taper un `ctrl-C` brutal. Un autre intérêt est d'effectuer un calcul parallèle sur les machines multi-processeur. Les fonctions liées aux thread sont dans la bibliothèque `pthread.h`, et il faut compiler avec la librairie `libpthread.a` :

```
$ gcc -lpthread monprog.c -o monprog
```

5.2.2 Création d'un thread et attente de terminaison

Pour créer un thread, il faut créer une fonction qui va s'exécuter dans le thread, qui a pour prototype :

```
void *ma_fonction_thread(void *arg);
```

Dans cette fonction, on met le code qui doit être exécuté dans le thread. On crée ensuite le thread par un appel à la fonction `pthread_create`, et on lui passe en argument la fonction `ma_fonction_thread` dans un pointeurs de fonction (et son argument `arg`). La fonction `pthread_create` a pour prototype :

```
int pthread_create(pthread_t *thread, pthread_attr_t *attributs,
                   void * (*fonction)(void *arg), void *arg);
```

Le premier argument est un passage par adresse de l'identifiant du thread (de type `pthread_t`). La fonction `pthread_create` nous retourne ainsi l'identifiant du thread, qui l'on utilise ensuite pour désigner le thread. Le deuxième argument `attributs` désigne les attributs du thread, et on peut mettre `NULL` pour avoir les attibuts par défaut. Le troisième argument est un pointeur

sur la fonstion à exécuter dans le thread (par exemple `ma_fonction_thread`, et le quatrième argument est l'argument de la fonction de thread.

Le processus qui exécute le `main` (l'équivalent du processus père) est aussi un thread et s'appelle le *thread principal*. Le thread principal peut attendre la fon de l'exécution d'un autre thread par la fonction `pthread_join` (similaire à la fonction `wait` dans le fork. Cette fonction permet aussi de récupérer la valeur retournée par la fonction `ma_fonction_thread` du thread. Le prototype de la fonction `pthread_join` est le suivant :

```
int pthread_join(pthread_t thread, void **retour);
```

Le premier paramètre est l'dentifiant du thread (que l'on obtient dans `pthread_create`), et le second paramètre est un `passage par adresse d'un pointeur` qui permet de récupérer la valeur retournée par `ma_fonction_thread`.

5.2.3 Exemples

Le premier exemple crée un thread qui dort un nombre de secondes passé en argument, pendant que le thread principal attend qu'il se termine.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

void *ma_fonction_thread(void *arg)
{
  int nbsec = (int)arg;
  printf("Je suis un thread et j'attends %d secondes\(\backslash\)n", nbsec);
  sleep(nbsec);
  puts("Je suis un thread et je me termine");
  pthread_exit(NULL);  {/* termine le thread proprement */
}

int main(void)
{
  int ret;
  pthread_t my_thread;
  int nbsec;
  time_t t1;
  srand(time(NULL));
  t1 = time(NULL);
  nbsec = rand()%10; {/* on attend entre 0 et 9 secondes */
  {/* on cr e le thread */
  ret = pthread_create(&my_thread, NULL,
                       ma_fonction_thread, (void*)nbsec);
  if (ret != 0)
    {
```

```
      fprintf(stderr, "Erreur de cr ation du thread");
      exit (1);
    }
  pthread_join(my_thread, NULL); {/* on attend la fin du thread */
  printf("Dans le main, nbsec = %d\(\backslash\)n", nbsec);
  printf("Duree de l'operation = %d\(\backslash\)n", time(NULL)-t1);
  return 0;
}
```

Le deuxième exemple crée un thread qui lit une valeur entière et la retourne au `main`.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

void *ma_fonction_thread(void *arg)
{
  int resultat;
  printf("Je suis un thread. Veuillez entrer un entier\(\backslash\)n");
  scanf("%d", &resultat);
  pthread_exit((void*)resultat);  {/* termine le thread proprement */
}

int main(void)
{
  int ret;
  pthread_t my_thread;
  {/* on cr e le thread */
  ret = pthread_create(&my_thread, NULL,
                       ma_fonction_thread, (void*)NULL);
  if (ret != 0)
    {
      fprintf(stderr, "Erreur de cr ation du thread");
      exit (1);
    }
  pthread_join(my_thread, (void*)&ret); {/* on attend la fin du thread */
  printf("Dans le main, ret = %d\(\backslash\)n", ret);
  return 0;
}
```

5.3 Donnée partagées et exclusion mutuelle

Lorsqu'un nouveau processus est créé par un `fork`, toutes les données (variables globales, variables locales, mémoire allouée dynamiquement), sont dupliquées et copiées, et le processus père et le processus fils travaillent ensuite sur des variables différentes.

Dans le cas de threads, la mémoire est *partagée*, c'est à dire que les variables globales sont partagées entre les différents threads qui s'exécutent en parallèle. Cela pose des problèmes lorsque deux threads différents essaient d'écrire et de lire une même donnée.

Deux types de problèmes peuvent se poser :

- Deux threads concurrents essaient en même temps de modifier une variable globale ;
- Un thread modifie une structure de donnée tandis qu'un autre thread essaie de la lire. Il est alors possible que le thread lecteur lise la structure alors que le thread écrivain a écrit la donnée à moitié. La donnée est alors incohérente.

Pour accéder à des données globales, il faut donc avoir recours à un mécanisme d'exclusion mutuelle, qui fait que les threads ne peuvent pas accéder en même temps à une donnée. Pour celà, on introduit des données appelés *mutex*, de type `pthread_mutex_t`.

Un thread peut verrouiller un `mutex`, avec la fonction `pthread_mutex_lock()`, pour pouvoir accéder à une donnée globale ou à un flot (par exemple pour écrire sur la sortie `stdout`). Une fois l'accès terminé, le thread dévérouille le mutex, avec la fonction `pthread_mutex_unlock()`. Si un thread A essaie de vérouiller le un mutex alors qu'il est déjà verrouillé par un autre thread B, le thread A reste bloqué sur l'appel de `pthread_mutex_lock()` jusqu'à ce que le thread B dévérouille le mutex. Une fois le mutex dévérouillé par B, le thread A verrouille immédiatement le mutex et son exécution se poursuit. Cela permet au thread B d'accéder tranquillement à des variables globales pendant que le thread A attend pour accéder aux mêmes variables.

Pour déclarer et initialiser un mutex, on le déclare en variable globale (pour qu'il soit accessible à tous les threads) :

```
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
```

La fonction `pthread_mutex_lock()`, qui permet de verrouiller un mutex, a pour prototype :

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Il faut éviter de verrouiller deux fois un même mutex dans le même thread sans le déverrouiller entre temps. Il y a un risque de blocage définitif du thread. Certaines versions du système gèrent ce problème mais leur comportement n'est pas portable.

La fonction `pthread_mutex_unlock()`, qui permet de déverrouiller un mutex, a pour prototype :

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Dans l'exemple suivant, différents threads font un travail d'une durée aléatoire. Ce travail est fait alors qu'un mutex est verrouillé.

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

```
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;

void* ma_fonction_thread(void *arg);

int main(void)
{
  int i;
  pthread_t thread[10];
  srand(time(NULL));

  for (i=0 ; i<10 ; i++)
      pthread_create(&thread[i], NULL, ma_fonction_thread, (void*)i);

  for (i=0 ; i<10 ; i++)
      pthread_join(thread[i], NULL);
  return 0;
}

void* ma_fonction_thread(void *arg)
{
  int num_thread = (int)arg;
  int nombre_iterations, i, j, k, n;
  nombre_iterations = rand()%8;
  for (i=0 ; i<nombre_iterations ; i++)
    {
      n = rand()%10000;
      pthread_mutex_lock(&my_mutex);
    printf("Le thread num ro %d commence son calcul\(\backslash\)n", num_thread);
      for (j=0 ; j<n ; j++)
        for (k=0 ; k<n ; k++)
          {}
     printf("Le thread numero %d a fini son calcul\(\backslash\)n", num_thread);
      pthread_mutex_unlock(&my_mutex);
    }
  pthread_exit(NULL);
}
```

Voici un extrait de la sortie du programme. On voit qu'un thread peut travailler tranquillement sans que les autres n'écrivent.

```
...
Le thread numéro 9 commence son calcul
Le thread numero 9 a fini son calcul
Le thread numéro 4 commence son calcul
Le thread numero 4 a fini son calcul
Le thread numéro 1 commence son calcul
```

```
Le thread numero 1 a fini son calcul
Le thread numéro 7 commence son calcul
Le thread numero 7 a fini son calcul
Le thread numéro 1 commence son calcul
Le thread numero 1 a fini son calcul
Le thread numéro 1 commence son calcul
Le thread numero 1 a fini son calcul
Le thread numéro 9 commence son calcul
Le thread numero 9 a fini son calcul
Le thread numéro 4 commence son calcul
Le thread numero 4 a fini son calcul
...
```

En mettant en commentaire les lignes avec `pthread_mutex_lock()` et `pthread_mutex_unlock()`, on obtient :

```
...
Le thread numéro 9 commence son calcul
Le thread numero 0 a fini son calcul
Le thread numéro 0 commence son calcul
Le thread numero 1 a fini son calcul
Le thread numéro 1 commence son calcul
Le thread numero 4 a fini son calcul
Le thread numero 8 a fini son calcul
Le thread numéro 8 commence son calcul
Le thread numero 8 a fini son calcul
Le thread numéro 8 commence son calcul
Le thread numero 1 a fini son calcul
Le thread numéro 1 commence son calcul
Le thread numero 3 a fini son calcul
Le thread numéro 3 commence son calcul
Le thread numero 3 a fini son calcul
Le thread numéro 3 commence son calcul
Le thread numero 5 a fini son calcul
Le thread numero 9 a fini son calcul
...
```

On voit que plusieurs threads interviennent pendant le calcul du thread numéro 9 et 4.

5.4 Sémaphores

En général, une section critique est une partie du code où un processus ou un thread ne peut rentrer qu'à une certaine condition. Lorsque le processus (ou un thread) entre dans la section critique, il modifie la condition pour les autres processus/threads.

Par exemple, si une section du code ne doit pas être exécutée simultanément par plus de n threads. Avant de rentrer dans la section critique, un thread doit vérifier qu'au plus n-1 threads y sont déjà. Lorsqu'un thread entre dans la section critique, il modifie la conditions

sur le nombre de threads qui se trouvent dans la section critique. Ainsi, un autre thread peut se trouver empêché d'entrer dans la section critique.

La difficulté est qu'on ne peut pas utiliser une simple variable comme compteur. En effet, si le test sur le nombre de thread et la modification du nombre de threads lors de l'entrée dans la section critique se font séquentiellement par deux instructions, si l'on joue de malchance un autre thread pourrait tester le condition sur le nombre de threads justement entre l'exécution de ces deux instructions, et deux threads passeraient en même temps dans la section critiques. Il y a donc nécessité de tester et modifier la condition de manière atomique, c'est à dire qu'aucun autre processus/thread de peut rien exécuter entre le test et la modification. C'est une opération atomique appelée *Test and Set Lock*.

Les sémaphores sont un type `sem_t` et une ensemble de primitives de base qui permettent d'implémenter des conditions assez générales sur les sections critiques. Un sémaphore possède un compteur dont la valeur est un entier positif ou nul. On entre dans une section critique si la valeur du compteur est strictement positive.

Pour utiliser une sémaphore, on doit le déclarer et l'initialiser à une certaine valeur avec la fonction `sem_init`.

```
\inte sem_init(sem_t *semaphore, \inte partage, {\bf unsigned} \inte valeur)
```

Le premier argument est un passage par adresse du sémaphore, le deuxième argument indique si le sémaphore peut être partagé par plusieurs processus, ou seulement par les threads du processus appelant (`partage` égale 0). Enfin, le troisième argument est la valeur initiale du sémaphore.

Après utilisation, il faut systématiquement libérer le sémaphore avec la fonction `sem_destroy`.

```
int sem_destroy (sem_t *semaphore)
```

Les primitives de bases sur les sémaphores sont :

- `sem_wait` : Reste bloquée si le sémaphore est nul et sinon décrémente le compteur (opération atomique) ;
- `sem_post` : incrémente le compteur ;
- `sem_getvalue` : récupère la valeur du compteur dans une variable passée par adresse ;
- `sem_trywait` : teste si le sémaphore est non nul et décrémente le sémaphore, mais sans bloquer. Provoque une erreur en cas de valeur nulle du sémaphore. Il faut utiliser cette fonction avec précaution car elle est prompte à générer des bogues.

Les prototypes des fonctions `sem_wait`, `sem_post` et `sem_getvalue` sont :

```
\inte sem_wait (sem_t * semaphore)
\inte sem_post(sem_t *semaphore)
\inte sem_getvalue(sem_t *semaphore, \inte *valeur)
```

Exemple. Le programme suivant permet au plus n sémaphores dans la section critique, où n en passé en argument.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

sem_t semaphore; {/* variable globale : s maphore */

void* ma_fonction_thread(void *arg);

int main(int argc, char **argv)
{
  int i;
  pthread_t thread[10];
  srand(time(NULL));

  if (argc != 2)
    {
      printf("Usage : %s nbthreadmax\(\backslash\)n", argv[0]);
      exit(0);
    }
  sem_init(&semaphore, 0, atoi(argv[1]));  {/* initialisation */

  {/* cr ation des threads */
  for (i=0 ; i<10 ; i++)
      pthread_create(&thread[i], NULL, ma_fonction_thread, (void*)i);

  {/* attente */
  for (i=0 ; i<10 ; i++)
      pthread_join(thread[i], NULL);
  sem_destroy(&semaphore);
  return 0;
}

void* ma_fonction_thread(void *arg)
{
  int num_thread = (int)arg;  {/* num ro du thread */
  int nombre_iterations, i, j, k, n;
  nombre_iterations = rand()%8+1;
  for (i=0 ; i<nombre_iterations ; i++)
    {
      sem_wait(&semaphore);
      printf("Le thread %d entre dans la section critique\(\backslash\)n",
              num_thread);
      sleep(rand()%9+1);
      printf("Le thread %d sort de la section critique\(\backslash\)n",
              num_thread);
```

```
      sem_post(&semaphore);
      sleep(rand()%9+1);
    }
  pthread_exit(NULL);
}
```

Exemples de trace :

```
\ gcc -lpthread semaphore.c -o semaphore
\  ./semaphore 2
Le thread 0 entre dans la section critique
Le thread 1 entre dans la section critique
Le thread 0 sort de la section critique
Le thread 2 entre dans la section critique
Le thread 1 sort de la section critique
Le thread 3 entre dans la section critique
Le thread 2 sort de la section critique
Le thread 4 entre dans la section critique
Le thread 3 sort de la section critique
Le thread 5 entre dans la section critique
Le thread 4 sort de la section critique
Le thread 6 entre dans la section critique
Le thread 6 sort de la section critique
Le thread 7 entre dans la section critique
Le thread 7 sort de la section critique
Le thread 8 entre dans la section critique
...
```

Autre exemple avec trois threads dans la section critique :

```
\ ./semaphore 3
Le thread 0 entre dans la section critique
Le thread 1 entre dans la section critique
Le thread 2 entre dans la section critique
Le thread 1 sort de la section critique
Le thread 3 entre dans la section critique
Le thread 0 sort de la section critique
Le thread 4 entre dans la section critique
Le thread 2 sort de la section critique
Le thread 5 entre dans la section critique
Le thread 3 sort de la section critique
Le thread 6 entre dans la section critique
Le thread 6 sort de la section critique
Le thread 7 entre dans la section critique
Le thread 5 sort de la section critique
Le thread 8 entre dans la section critique
```

```
Le thread 4 sort de la section critique
Le thread 9 entre dans la section critique
Le thread 9 sort de la section critique
Le thread 1 entre dans la section critique
Le thread 1 sort de la section critique
Le thread 2 entre dans la section critique
Le thread 7 sort de la section critique
Le thread 0 entre dans la section critique
Le thread 8 sort de la section critique
Le thread 6 entre dans la section critique
Le thread 2 sort de la section critique
Le thread 3 entre dans la section critique
Le thread 3 sort de la section critique
```

5.5 Exercices

Exercice 5.1 (∗) Écrire un programme qui crée un thread qui prend en paramètre un tableau d'entiers et l'affiche dans la console.

Exercice 5.2 (∗) Écrire un programme qui crée un thread qui alloue un tableau d'entiers, initialise les éléments par des entiers aléatoires entre 0 et 99, et retourne le tableau d'entiers.

Exercice 5.3 (∗∗) Créer une structure `TypeTableau` qui contient :

- Un tableau d'entiers ;
- Le nombre d'éléments du tableau ;
- Un entier x.

Écrire un programme qui crée un thread qui initialise un `TypeTableau` avec des valeurs aléatoires entre 0 et 99. Le nombre d'éléments du tableau est passé en paramètre. Dans le même temps, le thread principal lit un entiers x au clavier. Lorsque le tableau est fini de générer, le programme crée un thread qui renvoie 1 si l'élément x est dans le tableau, et 0 sinon.

Exercice 5.4 (∗∗) **a)** Reprendre la fonction de thread de génération d'un tableau aléatoire du 1. Le thread principal crée en parallèle deux tableaux T1 et T2, avec le nombre d'éléments de T1 plus petit que le nombre d'élemnts de T2.
b) Lorsque les tableaux sont finis de générer, lancer un thread qui détermine si le tableau T1 est inclus dans le tableau T2. Quelle est la complexité de l'algorithme ?
c) Modifier le programme précédent pour qu'un autre thread puisse terminer le programme si l'utilsateur appuie sur la touche 'A' (par exit(0)). Le programme doit afficher un message en cas d'annulation, et doit afficher le résultat du calcul sinon.

Exercice 5.5 (∗∗) Écrire un programme, avec un compteur global `compt`, et qui crée deux threads :

- Le premier thread itère l'opération suivante : on incrémente le compteur et attend un temps alléatoire entre 1 et 5 secondes.
- Le deuxième thread affiche la valeur du compteur toutes les deux secondes.

Les accès au compteur seront bien sûr protégés par un mutex. Les deux threads se terminent lorsque le compteur atteint une valeur limite passée en argument (en ligne de commande) au programme.

Exercice 5.6 (∗∗) Créer un programme qui a en variable globale un tableau de N `double`, avec N=100.

Dans le `main`, le tableau sera initialisé avec des valeurs réelles aléatoires entre 0 et 100, sauf les valeurs `tableau[0]` et `tableau[99]` qui vallent 0.

Le programme crée deux threads :

- Le premier thread remplace chaque valeur `tableau[i]`, avec i = 1,2,…,98 par la moyenne
  `(tableau[i-1]+tableau[i]+tableau[i+1])/3`
  Il attend ensuite un temps alléatoire entre 1 et 3 secondes ;
- Le deuxième thread affiche le tableau toutes les 4 secondes.

Exercice 5.7 (∗∗) Dans un programme prévu pour utiliser des threads, créer un compteur global pour compter le nombre d'itérations, et une variable globale réelle u. Dans le `main`, on initialisera u à la valeur 1

Le programme crée deux threads T1 et T2. Dans chaque thread Ti, on incrémente le compteur du nombre d'itération, et on applique une affectation :

```
u = f_i(u);
```

pour une fonction `f_i` qui dépend du thread.

$$f\_1(x) = \frac{1}{4}(x-1)^2 \text{ et } f\_2(x) = \frac{1}{6}(x-2)^2$$

De plus, le thread affiche la valeur de u et attend un temps aléatoire (entre 1 et 5 secondes) entre deux itérations.

Exercice 5.8 (∗∗) **(Problème du rendez-vous)**
a) Les sémaphores permettent de réaliser simplement des rendez-vous Deux threads T1 et T2 itèrent un traitement 10 fois. On souhaite qu'à chaque itération le thread T1 attende à la fin de son traitement qui dure 2 secondes le thread T2 réalisant un traitement d'une durée aléatoire entre 4 et 9 secondes. Écrire le programme principal qui crée les deux threads, ainsi que les fonctions de threads en organisant le rendez-vous avec des sémaphores.

b) Dans cette version N threads doivent se donner rendez-vous, N étant passé en argument au programme. Les threads ont tous une durée aléatoire entre 1 et 5 secondes.

Exercice 5.9 (∗ ∗ ∗) (Problème de l'émetteur et du récepteur) Un thread émetteur dépose , à intervalle variable entre 1 et 3 secondes, un octet dans une variable globale à destination d'un processus récepteur. Le récepteur lit cet octet à intervalle variable aussi entre 1 et 3 secondes. Quelle solution proposez-vous pour que l'émetteur ne dépose pas un nouvel octet alors que le récepteur n'a pas encore lu le précédent et que le récepteur ne lise pas deux fois le même octet ?

Exercice 5.10 (∗ ∗ ∗) (Problème des producteurs et des consommateurs) Des processus producteurs produisent des objets et les insère un par un dans un tampon de n places. Bien entendu des processus consommateurs retirent, de temps en temps les objets (un par un).

Résolvez le problème pour qu'aucun objet ne soit ni perdu ni consommé plusieurs fois. Écrire une programme avec N threads producteurs et M threads consomateurs, les nombres N et M étant saisis au clavier. Les producteurs et les consomateurs attendent un temps aléatoire entre 1 et 3 secondes entre deux produits. Les produits sont des octets que l'on stocke dans un tableau de 10 octets avec gestion `LIFO`. S'il n'y a plus de place, les producteurs restent bloqués en attendant que des places se libèrent.

Exercice 5.11 (∗ ∗ ∗) (Problème des lecteurs et des rédacteurs) Ce problème modélise les accès à une base de données. On peut accepter que plusieurs processus lisent la base en même temps mais si un processus est en train de la modifier, aucun processus, pas même un lecteur, ne doit être autorisé à y accéder. Comment programmer les lecteurs et les rédacteurs ? Proposer une solution avec famine des écrivains : les écrivains attendent qu'il n'y ait aucun lecteur. Écrire une programme avec N threads lecteurs et M threads rédacteurs, les nombres N et M étant saisis au clavier. La base de donnée est représentée par un tableau de 15 octets initialisés à 0. À chaque lecture/écriture, le lecteur/écrivain lit/modifie l'octet à un emplacement aléatoire. Entre deux lectures, les lecteurs attendent un temps aléatoire entre 1 et 3 secondes. Entre deux écritures, les écrivains attendent un temps aléatoire entre 1 et 10 secondes.

Chapitre 6

Gestion du disque dûr et des fichiers

6.1 Organisation du disque dur

6.1.1 Plateaux, cylindres, secteurs

Un disque dur possède plusieurs plateaux, chaque plateau possède deux faces, chaque face possède plusieurs secteurs et plusieurs cylindres (voir ficgure ci-dessous).

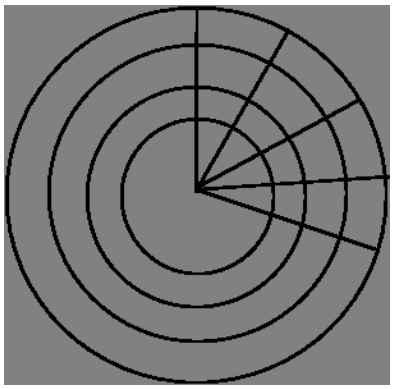

FIGURE 6.1 : Organisation d'une face de plateau

Le disque possède un secteur de boot, qui contient des informations sur les partitions bootables et le boot-loader, qui permet de choisir le système sous lequel on souhaite booter. Un disque est divisé en partitions (voir la figure 6.2) Les données sur les partitions (telles que les cylindre de début et de fin ou les types de partition) sont stockées dans une *table des partitions*. Le schéma d'organisation d'une partition *UNIX* est montré sur la figure 6.3

6.1.2 Gérer les partitions et périphériques de stockage

L'outil `fdisk` permet de modifier les partitions sur un disque dur. Par exemple, pour voir et modifier les partitions sur le disque dur *SATA* `/dev/sda`, on lance `fdisk` :

```
# fdisk /dev/sda

The number of cylinders for this disk is set to 12161.
There is nothing wrong with that, but this is larger than 1024,
```

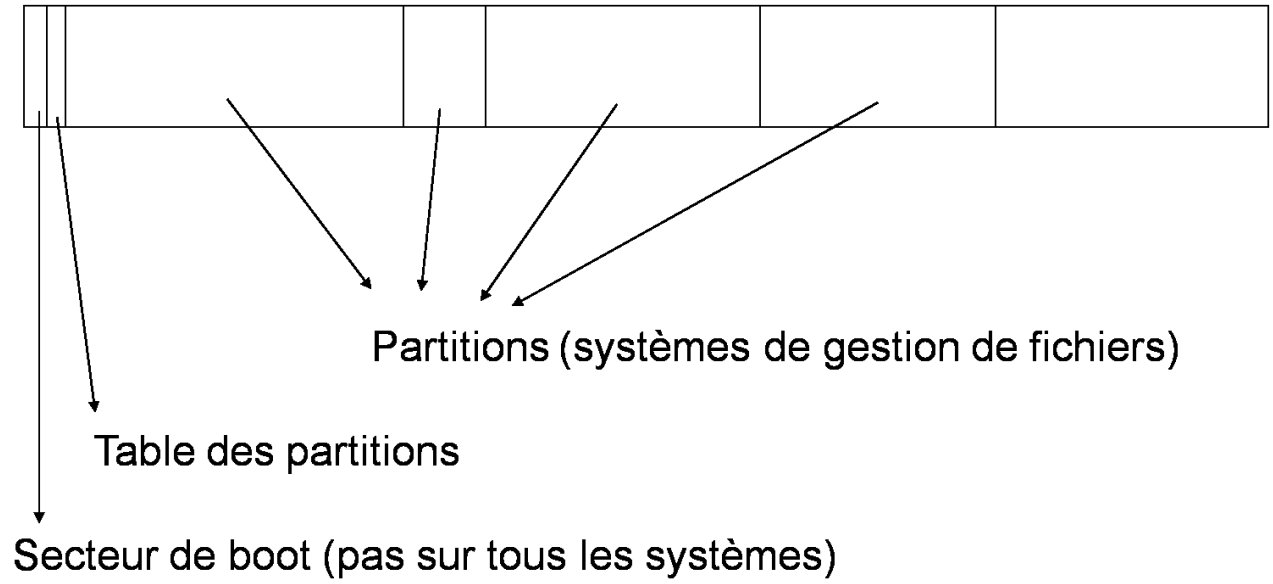

FIGURE 6.2 : Organisation d'un disque dur

```
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): m
Command action
   a   toggle a bootable flag
   b   edit bsd disklabel
   c   toggle the dos compatibility flag
   d   delete a partition
   l   list known partition types
   m   print this menu
   n   add a new partition
   o   create a new empty DOS partition table
   p   print the partition table
   q   quit without saving changes
   s   create a new empty Sun disklabel
   t   change a partition's system id
   u   change display/entry units
   v   verify the partition table
   w   write table to disk and exit
   x   extra functionality (experts only)

Command (m for help): p

Disk /dev/sda: 100.0 GB, 100030242816 bytes
255 heads, 63 sectors/track, 12161 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
```

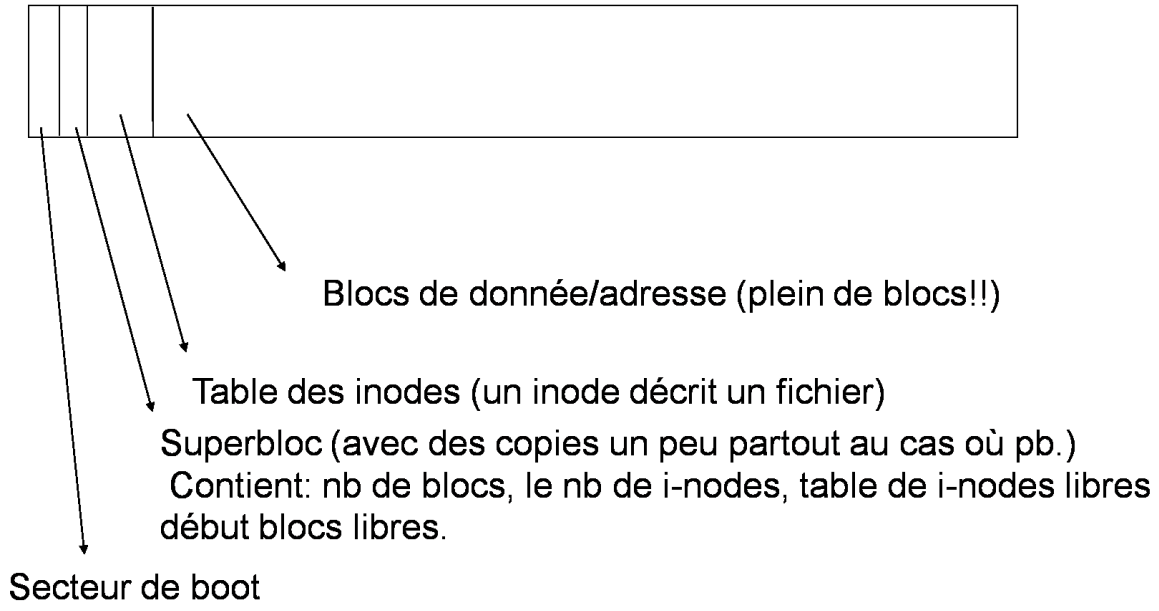

FIGURE 6.3 : Organisation d'une partition

```
   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1        1459    11719386    7  HPFS/NTFS
/dev/sda2            1460        2638     9470317+  83  Linux
/dev/sda3            2639       12161    76493497+   5  Extended
/dev/sda5           11796       12161     2939863+  82  Linux swap / Solaris
/dev/sda6            2639       11795    73553571   83  Linux

Partition table entries are not in disk order

Command (m for help): q

#
```

On voit en particulier le nombre total des cylindres, la taille de chaque cylindre, et pour chaque partition, le cylindre de début et de fin. Ici, on voit que `/dev/sda1` est une partition *NTFS* (type de partition pour windows), les partitions `/dev/sda2` et `/dev/sda6` sont de type *ext3* (linux), et la partition `/dev/sda5` est un espace de swap (utilisée par les programmes lors d'un dépassement de la *RAM*.

Les types de partition qu'il est (actuellement) possible de créer avec `fdisc` sont :

```
 0  Empty           1e  Hidden W95 FAT1 80  Old Minix       be  Solaris boot
 1  FAT12           24  NEC DOS         81  Minix / old Lin bf  Solaris
 2  XENIX root      39  Plan 9          82  Linux swap / So c1  DRDOS/sec (FAT-
 3  XENIX usr       3c  PartitionMagic  83  Linux           c4  DRDOS/sec (FAT-
 4  FAT16 <32M      40  Venix 80286     84  OS/2 hidden C:  c6  DRDOS/sec (FAT-
 5  Extended        41  PPC PReP Boot   85  Linux extended  c7  Syrinx
 6  FAT16           42  SFS             86  NTFS volume set da  Non-FS data
 7  HPFS/NTFS       4d  QNX4.x          87  NTFS volume set db  CP/M / CTOS / .
```

```
 8  AIX             4e  QNX4.x 2nd part 88  Linux plaintext de  Dell Utility
 9  AIX bootable    4f  QNX4.x 3rd part 8e  Linux LVM       df  BootIt
 a  OS/2 Boot Manag 50  OnTrack DM      93  Amoeba          e1  DOS access
 b  W95 FAT32       51  OnTrack DM6 Aux 94  Amoeba BBT      e3  DOS R/O
 c  W95 FAT32 (LBA) 52  CP/M            9f  BSD/OS          e4  SpeedStor
 e  W95 FAT16 (LBA) 53  OnTrack DM6 Aux a0  IBM Thinkpad hi eb  BeOS fs
 f  W95 Ext'd (LBA) 54  OnTrackDM6      a5  FreeBSD         ee  EFI GPT
10  OPUS            55  EZ-Drive        a6  OpenBSD         ef  EFI (FAT-12/16/
11  Hidden FAT12    56  Golden Bow      a7  NeXTSTEP        f0  Linux/PA-RISC b
12  Compaq diagnost 5c  Priam Edisk     a8  Darwin UFS      f1  SpeedStor
14  Hidden FAT16 <3 61  SpeedStor       a9  NetBSD          f4  SpeedStor
16  Hidden FAT16    63  GNU HURD or Sys ab  Darwin boot     f2  DOS secondary
17  Hidden HPFS/NTF 64  Novell Netware  b7  BSDI fs         fd  Linux raid auto
18  AST SmartSleep  65  Novell Netware  b8  BSDI swap       fe  LANstep
1b  Hidden W95 FAT3 70  DiskSecure Mult bb  Boot Wizard hid ff  BBT
1c  Hidden W95 FAT3 75  PC/IX
```

Pour formater une partition, par exemple après avoir créé la partition, on utilise `mkfs`.
Exemple.

```
pour une partition windows :
# mkfs -t ntfs /dev/sda1
pour une clef usb sur /dev/sdb1 :
# mkfs -t vfat /dev/sdb1
pour une partition linux ext3 (comme le /home) :
# mkfs -t ext3 /dev/sda6
```

Il faut ensuite monter la partition, c'est à dire associer la partition à un répertoire dans l'arborescence du système de fichiers. **Exemple.**

```
Pour monter une clef usb qui est sur /dev/sdb1
# mkdir /mnt/usb
# mount -t vfat /dev/sdb1 /mnt/usb
# ls /mnt/usb/
etc...
ou encore, pour accéder à une partition windows
à partir de linux :
# mkdir /mnt/windows
# mount -t ntfs /dev/sda1 /mnt/windows
# ls -l /mnt/windows
```

Pour démonter un volume, utiliser `umount`. On peut monter automatiquement un périphérique en rajoutant une ligne dans le fichier `fstab` :

```
# cat /etc/fstab
# /etc/fstab: static file system information.
#
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
proc            /proc           proc    defaults        0       0
```

```
/dev/sda2       /               ext3    defaults,errors=remount-ro 0       1
/dev/sda6       /home           ext3    defaults        0       2
/dev/sda5       none            swap    sw              0       0
/dev/scd0       /media/cdrom0   udf,iso9660 user,noauto 0       0
/dev/sdb1 /mnt/usb         vfat user,noauto        0        0
/dev/mmcblk0p1  /mnt/sdcard         vfat user,noauto        0        0
/dev/sda1       /mnt/windows     ntfs    uid=1001,auto
```

Par exemple ici le lecteur de *CDROM*, la clef *USB* et la caret `SD` peuvent être montée par tous les utilisateurs (option `user`). Par contre, l'accès à la partition windows est réservé à l'utilisateur d'*UID* 1001 (voir `/etc/passwd` pour trouver l'*IUD* d'un utilisateur).

On peut voir la liste des partitions qui sont montées, ainsi que l'espace libre dans chacune d'elles, par la commande `df` :

```
# df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda2              9322492   8362396    486584  95% /
tmpfs                   517544         0    517544   0% /lib/init/rw
udev                     10240        72     10168   1% /dev
tmpfs                   517544         0    517544   0% /dev/shm
/dev/sda6             72397784  34359196  34360912  50% /home
/dev/sda1             11719384   6920456   4798928  60% /mnt/windows
/dev/sdb1              3991136    126880   3864256   4% /mnt/usb
/dev/scd0               713064    713064         0 100% /media/cdrom0
/dev/mmcblk0p1         2010752    528224   1482528  27% /mnt/sdcard
```

6.1.3 Fichiers, inodes et liens

Un *inode* identifie un fichier et décrit ses propritées, telles qu'in peut les vois par `ls -l` :

- données sur le propriétaire : *UID* et *GID* ;
- Droits d'accès ;
- Dates de création, de dernière modification, de dernier accès ;
- Nombre de fichier ayant cet inode (en cas de liens durs) ;
- Taille en octets ;
- **Adresse** d'un block de données.

Il existe dans un disque dur des liens, lorsque plusieurs noms de fichiers conduisent aux mêmes données. Ces liens sont de deux types :

1. On parle d'un **lien dur** lorsque deux noms de fichiers sont associés au même inode (les différents liens durs a exactement les mêmes propriétés mais des noms différents). En particulier, l'adresse des données est la même dans tous les liens durs. La commande `rm` décrémente le nombre de liens durs. La suppression n'est effective que lorsque le nombre de lien durs (visible par `ls -l` ou `stat`) devient nul. Les liens durs sont créés par la commande `ln`.

2. On parle d'un **lien symbolique** lorsque le block de données d'un fichier contient l'adresse du bloc de données d'un autre fichier. Les liens symboliques sont créés par la commande `ln -s` (optin `-s`).

6.2 Obtenir les informations sur un fichier en C

On peut obtenir les informations sur un fichier ou un répertoire (ID du propriétaire, taille inode,...) avec la fonction `stat`.
Exemple. L'exemple suivant affiche des informations sur le fichier dont le nom est passé en argument.

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

int main(int argc, char**argv)
{
  struct stat st;  /* pour récupérer les informations sur un fichier */
  struct tm *temps; /* pour traduire les  dates (voir ctime(3)) */

  if (argc != 2)
    {
      fprintf(stderr, "Usage : %s nom_de_fichier\n", argv[0]);
      exit(1);
    }
  if (stat(argv[1], &st)!= 0)
    {
      perror("Erreur d'accès au fichier\n");
      exit(1);
    }
  if (S_ISDIR(st.st_mode))
    printf("Le nom %s correspond à un répertoire\n", argv[1]);
  if (S_ISREG(st.st_mode))
    {
      printf("%s est un fichier ordinaire\n", argv[1]);
      printf("La taille du fichier en octets est %d\n", st.st_size);
      temps = localtime(&st.st_mtime);
      printf("Le jour de dernière mofification est %d/%d/%d\n",
             temps->tm_mday, temps->tm_mon+1, temps->tm_year+1900);
    }
  return 0;
}
```

Exemple de trace d'exécution de ce programme :

```
$ gcc testStat.c -o testStat
$ ls -l
-rwxr-xr-x 1 remy remy 8277 Feb  1 13:03 testStat
-rw-r--r-- 1 remy remy  935 Feb  1 13:03 testStat.c
$ ./testStat testStat.c
testStat.c est un fichier ordinaire
La taille du fichier en octets est 935
Le jour de dernière mofification est 1/2/2008
```

Toutes les informatione renvoyées par `stat` sont stockées dans la structure `stat`, dont la déclaration est la suivante (voir `stat(2)`)

```
struct stat {
      dev_t     st_dev;     /* ID of device containing file */
      ino_t     st_ino;     /* inode number */
      mode_t    st_mode;    /* protection */
      nlink_t   st_nlink;   /* number of hard links */
      uid_t     st_uid;     /* user ID of owner */
      gid_t     st_gid;     /* group ID of owner */
      dev_t     st_rdev;    /* device ID (if special file) */
      off_t     st_size;    /* total size, in bytes */
      blksize_t st_blksize; /* blocksize for filesystem I/O */
      blkcnt_t  st_blocks;  /* number of blocks allocated */
      time_t    st_atime;   /* time of last access */
      time_t    st_mtime;   /* time of last modification */
      time_t    st_ctime;   /* time of last status change */
};
```

6.3 Parcourir les répertoires en *C*

La fonction `opendir` permet d'ouvrir un répertoire et retourne un pointeur de répertoire (type `DIR*`). On peut alors parcourir la liste des éléments (fichiers, liens ou répertoires) qui sont contenus dans ce répertoire (y compris les répertoires `"."` et `".."`).

```
#include <stdio.h>
#include <dirent.h>
#include <sys/types.h>
#include <stdlib.h>

int main(int argc, char**argv)
{
   DIR *dir;
   struct dirent *ent;
   int i;

   for (i=1 ; i<argc ; i++)
```

```
    {
      dir = opendir(argv[i]); /* ouverture du répertoire */
      if (dir==NULL)
        {
         fprintf(stderr, "Erreur d'ouverture du réperoire %s\(\backslash\)n",
                 argv[i]);
          fprintf(stderr, "Droits inssufisant ou répertoire incorrect\n");
          exit(1);
        }
      printf("Répertoire %s\n", argv[i]);
      while ((ent=readdir(dir)) != NULL) /* on parcourt la liste */
        printf("%s ", ent->d_name);
    }
  printf("\n");
  retu 0;
}
```

Exemple de trace d'exécution de ce programme :

```
$ gcc parcoursRep.c -o parcoursRep
$ ls
parcoursRep  parcoursRep.c
$ ./parcoursRep .
Répertoire .
parcoursRep .. parcoursRep.c .
```

6.4 Descripteurs de fichiers

Un descripteur de fichier est un entier qui identifie un fichier dans un programme C. Ne pas confondre un descripteur de fichier avec un pointeur de fichier. La fonction `fdopen` permet d'obtenir un pointeur de fichier à partir d'un descripteur.

6.4.1 Ouverture et création d'un fichier

La fonction `open` permet d'obtenir un descripteur de fichier à partir du nom de fichier sur le disque, de la même façon que `fopen` permet d'obtenir un pointeur de fichier. Une grande différence est que la fonction `open` offre beaucoup plus d'options pour tester les permissions. Le prototype de la fonction `open` est le suivant :

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

La fonction retourne une valeur strictement négative en cas d'erreur.

Le paramètre `flags` permet de préciser si, pour le programme, le fichier est en lecture seule (masque `O_RDONLY`), écriture seule (masque `O_WRONLY`), ou lecture-écriture (masque `O_RDWR`). Par un ou bit à bit (`|`), on peut aussi préciser si le fichier est ouvert en mode ajout (masque `O_APPEND`), ou si le fichier doit être écrasé (masque `O_TRUNC`) ou ouvert en mode création (si le fichier n'existe pas il sera créé, masque `O_CREAT`).

Le mode permet de fixer les permissions lors de la création du fichier (le cas échéant), lorsque le paramètre `flag` est spécifié avec `O_CREAT`. Les permissions peuvent être définies par des chiffres octaux (7 pour rwx, 0 pour aucune permission) pour l'utilisateur, le groupe et les autres ustilisateur. Cependant, un et bit à bit est effectué avec la négation bit à bit de variable d'environnement `umask` (`mode & ũmask`)). (voir `man umask` et `man open(2)`)

6.4.2 Lecture et écriture via un descripteur

Pour écrire des octets via descripteur de fichier, on utilise la fonction `write` :

```
ssize_t write(int descripteur1, const void *bloc, size_t taille);
```

Le fichier (ou tube ou socket...) doit être ouvert en écriture (options `O_WRONLY, O_RDWR`) La taille est le nombre d'octets qu'on souhaite écrire, et le bloc est un pointeur vers la mémoire contenant ces octets.

Pour lire des octets via un descripteur de fichiers, on utilise la fonction `read` :

```
ssize_t read(int descripteur0, void *bloc, size_t taille);
```

Le fichier (ou tube ou socket...) doit être ouvert en lecture (options `O_RDONLY, O_RDWR`)

On peut aussi utiliser `fdopen` pour obtenir un `FILE*` ce qui permet d'utiliser des fonctions plus haut niveau telles que `fprintf` et `fscanf` ou `fread` et `fwrite`.

6.5 Exercices

Exercice 6.1 ([) sortez vos calculettes !] Soit un dique du ayant 2 plateaux, chaque face ayant 1000 cylindres et 60 secteurs, chaque secteur ayant 1024 octets.

a) Calculez la capacité totale du disque.

b) Caluculez la position (le numéro) de d'octet 300 su secteur 45 du cylindre 350 de la face 2 du premier plateau.

c) Sur quel secteur et en quelle position se trouve l'octet numéro 78000000 ?

Exercice 6.2 (**)** Ecrire un programme qui prend en argument des noms de répertoire et affiche la liste des fichiers de ces répertoires qui ont une taille supérieure à (à peu près) $1Mo$ avec l'*UID* du propriétaire du fichier.

Exercice 6.3 (**a**)) Écrire un programme qui saisit au clavier un tableau d'entiers et sauvegarde ce tableau au format binaire dans un fichier ayant permission en écriture pour le groupe du fichier et en lecture seule pour les autres utilisateurs.

b) Écrire une programme qui charge en mémoire un tableau d'entiers tel que généré au a). Le fichier d'entiers ne contient pas le nombre d'éléments. Le programme doit fonctionner pour un nombre quelconque de données entières dans le fichier.

Chapitre 7

Signaux

7.1 Préliminaire : Pointeurs de fonctions

Un pointeur de fonctions en C est une variable qui permet de désigner une fonction C. Comme nimporte quelle variable, on peut mettre un pointeur de fonctions soit en variable dans une fonction, soit en paramètre dans une fonction.

On déclare un pointeur de fonction comme un prototype de fonction, mais on ajoute une étoile ($*$) devant le nom de la fonction. Dans l'exemple suivant, on déclare dans le main un pointeur sur des fonctions qui prennent en paramètre un `int`, et un pointeur sur des fonctions qui retournent un `int`.

```
#include <stdio.h>

int SaisisEntier(void)
{
   int n;
   printf("Veuillez entrer un entier : ");
   scanf("%d", &n);
   return n;
}

void AfficheEntier(int n)
{
   printf("L'entier n vaut %d\(\backslash\)n", n);
}

int main(void)
{
  void (*foncAff)(int); /* déclaration d'un pointeur foncAff */
  int (*foncSais)(void); \em{}/*déclaration d'un pointeur foncSais */
  inte entier;

  foncSais = SaisisEntier;  {/* affectation d'une fonction */
  foncAff = AfficheEntier; /* affectation d'une fonction */
```

```
  entier = foncSais(); /* on exécute la fonction */
  foncAff(entier);   /* on exécute la fonction */
  \retu 0;
}
```

Dans l'exemple suivant, la fonction est passée en paramètre à une autre fonction, puis exécutée.

```
#include <stdio.h>

int SaisisEntier(void)
{
   int n;
   printf("Veuillez entrer un entier : ");
   scanf("%d", &n);
   getchar();
   return n;
}

void AfficheDecimal(int n)
{
   printf("L'entier n vaut %d\n", n);
}

void AfficheHexa(int n)
}
   printf("L'entier n vaut %x\(\backslash\)n", n);
}

void ExecAffiche(void (*foncAff)(int), int n)
{
  foncAff(n);  /* exécution du paramètre */
}

int main(void)
{
  int (*foncSais)(\void); /*déclaration d'un pointeur foncSais */
  int entier;
  char rep;

  foncSais = SaisisEntier;  /* affectation d'une fonction */
  entier = foncSais(); /* on exécute la fonction */

  puts("Voulez-vous afficher l'entier n en décimal (d) ou en hexa (x)?");
  rep = getchar();
  /* passage de la fonction en paramètre : */
  if (rep == 'd')
```

```
    ExecAffiche(AfficheDecimal, entier);
  if (rep == 'x')
    ExecAffiche(AfficheHexa, entier);

  return 0;
}
```

7.2 Les principaux signaux

Un signal permet de prévenir un processus qu'un évennement particulier c'est produit dans le système pour dans un (éventuellement autre) processus. Certains signaux sont envoyés par le noyau (comme en cas d'erreur de violation mémoire ou division par 0), mais un programme utilisateur peut envoyer un signal avec la fonction ou la commande `kill`, ou encore par certaines combinaison de touches au clavier (comme `Ctrl-C`). Un utilisateur (à l'exception de `root`) ne peut envoyer un signal qu'à un processus dont il est propriétaire.

Les principaux signaux (décrits dans la norme `POSIX.1-1990`) (faire `man 7 signal` pour des compléments) sont expliqués sur le table 7.1.

7.3 Envoyer un signal

La méthode la plus générale pour envoyer un signal est d'utiliser soit la commande sheel `kill(1)`, soit la fonction C `kill(2)`.

7.3.1 La commande `kill`

La commande `kill` prend une option `-signal` et un `pid`.
Exemple.

```
$ kill -SIGINT 14764 {\em# interromp processus de pid 14764}
$ kill -SIGSTOP 22765 {\em# stoppe temporairement le process 22765}
$ kill -SIGCONT 22765 {\em# reprend l'exécution du prcessus 22765}
```

Compléments

- √ Le signal par défaut, utilisé en cas est `SIGTERM`, qui termine le processus.
- √ On peut utiliser un *PID* négatif pour indiquer un groupe de processus, tel qu'il est indiqué par le *PGID* en utilisant l'option `-j` de la commande `ps`. Cela permet d'envoyer un signal à tout un groupe de processus.

7.3.2 La fonction `kill`

La fonction C `kill` est similaire à la commande `kill` du *shell*. Elle a pour prototype :

```
int kill(pid_t pid, int signal);
```

Signal	Valeur	Action	Commentaire
SIGHUP	1	Term	Terminaison du leader de session (exemple : terminaison du terminal qui a lancé le programme ou logout)
SIGINT	2	Term	Interruption au clavier (par Ctrl-C par défaut)
SIGQUIT	3	Core	Quit par frappe au clavier (par Ctr − AltGr − \ par défaut)
SIGILL	4	Core	Détection d'une instruction illégale
SIGABRT	6	Core	Avortement du processus par la fonction abort(3) (généralement appelée par le programmeur en cas de détection d'une erreur)
SIGFPE	8	Core	Exception de calcul flottant (division par 0 racine carrées d'un nombre négatif, etc...)
SIGKILL	9	Term	Processus tué (kill)
SIGSEGV	11	Core	Violation mémoire. Le comportement par défaut termine le processus sur une erreur de segmentation
SIGPIPE	13	Term	Erreur de tube : tentative d'écrire dans un tube qui n'a pas de sortie
SIGALRM	14	Term	Signal de timer suite à un appel de alarm(2) qui permet d'envoyer un signal à une certaine date
SIGTERM	15	Term	Signal de terminaison
SIGUSR1	30,10,16	Term	Signal utilisateur 1 : permet au programmeur de définir son propre signal pour une utilisation libre
SIGUSR2	31,12,17	Term	Signal utilisateur 23 : permet au programmeur de définir son propre signal pour une utilisation libre
SIGCHLD	20,17,18	Ign	L'un des processus fils est stoppé (par SIGSTOP ou terminé)
SIGSTOP	17,19,23	Stop	Stoppe temporairement le processus. Le processus se fige jusqu'à recevoir un signal SIGCONT
SIGCONT	19,18,25	Cont	Reprend l'exécution d'un processus stoppé.
SIGTSTP	18,20,24	Stop	Processus stoppé à partir d'un terminal (tty) (par Ctrl-S par défaut)
SIGTTIN	21,21,26	Stop	saisie dans un terminal (tty) pour un processus en tâche de fond (lancé avec &)
SIGTTOU	22,22,27	Stop	Affichage dans un terminal (tty) pour un processus en tâche de fond (lancé avec &)

Table 7.1 : Liste des principaux signaux

Exemple. Le programme suivant tue le processus dont le *PID* est passé en argument seulement si l'utilisateur confirme.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <signal.h>

int main(int argc, char **argv)
{
  pid_t pidToSend;
  char rep;
  if (argc != 2)
    {
      fprintf(stderr, "Usage %s pid\n", argv[0]);
      exit(1);
    }
  pidToSend = atoi(argv[1]);
  printf("Etes-vous sûr de vouloir tuer le processus %d? (o/n)",
         pidToSend);
  rep = getchar();
  if (rep == 'o')
     kill(pidToSend, SIGTERM);
  return 0;
}
```

7.3.3 Envoi d'un signal par combinaison de touches du clavier

Un certain nombre de signaux peuvent être envoyé à partir du terminal par une combinaison de touche. On peut voir ces combinaisons de touches par `stty -a` :

```
 stty -a
speed 38400 baud; rows 48; columns 83; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?; eol2 = M-^?;
swtch = M-^?; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V;
flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff -iuclc
ixany imaxbel iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt echoctlechoke
```

7.4 Capturer un signal

7.4.1 Créer un gestionnaire de signal (*signal handler*)

Un gestionnaire de signal (*signal handler*) permet de changer le comportement du processus lors de la réception du signal (par exemple, se terminer).

Voici un exemple de programme qui sauvegarde des données avant de se terminer lors d'une interruption par `Ctrl-C` dans le terminal. Le principe est de modifier le comportement lors de la réception du signal `SIGINT`.

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

int donnees[5];

void gestionnaire(int numero)
{
  FILE *fp;
  int i;
  if (numero == SIGINT)
    {
      printf("\nSignal d'interruption, sauvegarde...\n");
      fp = fopen("/tmp/sauve.txt", "w");
      for (i=0 ; i<5 ; i++)
        {
          fprintf(fp, "%d ", donnees[i]);
        }
      fclose(fp);
      printf("Sauvegarde terminée, terminaison du processus\n");
      exit(0);
    }
}

int main(void)
{
  int i;
  char continuer='o';
  struct sigaction action;
  action.sa_handler = gestionnaire; /* pointeur de fonction */
  sigemptyset(&action.sa_mask); /* ensemble de signaux vide */
  action.sa_flags = 0; /* options par défaut */
  if (sigaction(SIGINT, &action, NULL) != 0)
    {
      fprintf(stderr, "Erreur sigaction\(\backslash\)n");
      exit(1);
    }
```

```
  for (i=0 ; i<5 ; i++)
    {
      printf("donnees[%d] = ", i);
      scanf("%d", &donnees[i]); getchar();
    }

  while (continuer == 'o')
    {
      puts("zzz...");
      sleep(3);
      for (i=0 ; i<5 ; i++)
          printf("donnees[%d] = %d ", i, donnees[i]);
      printf("\(\backslash\)nVoules-vous continuer ? (o/n) ");
      continuer = getchar(); getchar();
    }
}
```

Voici le trace de ce programme, que l'on interrompt avec `Ctrl-C` :

```
 gcc sigint.c -o sigint
 ./sigint
donnees[0] = 5
donnees[1] = 8
donnees[2] = 2
donnees[3] = 9
donnees[4] = 7
zzz...
donnees[0] = 5 donnees[1] = 8 donnees[2] = 2 donnees[3] = 9 donnees[4] = 7
Voules-vous continuer ? (o/n)
Signal d'interruption, sauvegarde...
Sauvegarde terminée, terminaison du processus
 cat /tmp/sauve.txt
5 8 2 9 7
```

7.4.2 Gérer une exception de division par zéro

Voici un programme qui fait une division entre deux entiers y/x saisis au clavier. Si le dénominateur x vaut 0 un signal `SIGFP` est reçu et le gestionnaire de signal fait resaisir x. Une instruction `siglongjmp` permet de revenir à la ligne d'avant l'erreur de division par 0.

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <setjmp.h>

int x,y, z;
sigjmp_buf env;
```

```
void gestionnaire(int numero)
{
  FILE *fp;
  int i;
  if (numero == SIGFPE)
    {
      printf("Signal d'erreur de calcul flottant.\n");
      printf("Entrez x différent de zéro : ");
      scanf("%d", &x);
      siglongjmp(env, 1); /* retour au sigsetjmp */
    }
}

int main(void)
{
  int i;
  char continuer='o';
  struct sigaction action;
  action.sa_handler = gestionnaire; /* pointeur de fonction */
  sigemptyset(&action.sa_mask); /* ensemble de signaux vide */
  action.sa_flags = 0; /* options par défaut */
  if (sigaction(SIGFPE, &action, NULL) != 0)
    {
      fprintf(stderr, "Erreur sigaction\n");
      exit(1);
   }

    printf("Veuillez entrer x et y : ");
    scanf("%d %d", &x, &y); getchar();
    sigsetjmp(env, 1); /* on mémorise la ligne */
    z = y/x;  /* opération qui risque de provoquer une erreur */
    printf("%d/%d=%d\(\backslash\)n", y, x, z);
    return 0;
}
```

Voici la trace de ce programme lorsqu'on saisit un dénominateur x égal à 0 :

```
Veuillez entrer x et y : 0 8
Signal d'erreur de calcul flottant.
Entrez x différent de zéro : 0
Signal d'erreur de calcul flottant.
Entrez x différent de zéro : 2
8/2=4
```

7.5 Exercices

Exercice 7.1 () Ecrire un programme qui crée un fils qui fait un calcul sans fin. Le processus père propose alors un menu :

- Lorsque l'utilisateur appuie sur la touche `'s'`, le processus père endort son fils.
- Lorsque l'utilisateur appuie sur la touche `'r'`, le processus père redémare son fils.
- Lorsque l'utilisateur appuie sur la touche `'q'`, le processus près tue son fils avant de se terminer.

Exercice 7.2 () Ecrire un programme `saisit.c` qui saisit un int au clavier, et l'enregistre dans un fichier `/tmp/entier.txt`. Écrire un programme `affiche.c` qui attend (avec `sleep`) un signal utilisateur du programme `saisit.c`. Lorsque l'entier a été saisi, le programme `affiche.c` affiche la valeur de l'entier.

Exercice 7.3 (a)) Écrire un programme qui crée 5 processus fils qui font une boucle `while 1`. Le processus père proposera dans une boucle sans fin un menu à l'utilisateur :

- Endormir un fils ;
- Réveiller un fils ;
- Terminer un fils ;

b) Modifier les fils pour qu'ils affichent un message lorsqu'ils sont tués. Le processus père affichera un autre message s'il est tué.

Exercice 7.4 () Ecrire un programme qui saisit les valeurs d'un tableau d'entier `tab` de n éléments alloué dynamiquement. L'entier n sera saisi au clavier. Le programme affiche la valeur d'un élément `tab[i]` où i est saisi au clavier. En cas d'erreur de segmentation le programme fait resaisir la valeur de i avant de l'afficher.

Chapitre 8

Programmation réseaux

Le but de la programmation réseau est de permettre à des programmes de dialoguer (d'échanger des données) avec d'autres programmes qui se trouvent sur des ordinateurs distants, connectés par un réseau. Nous verrons tout d'abord des notions générales telles que les adresse IP ou le protocole TCP, avant d'étudier les sockets unix/linux qui permettent à des programmes d'établir une communication et de dialoguer.

8.1 Adresses IP et MAC

Chaque interface de chaque ordinateur sera identifié par

- Son adresse IP : une adresse IP (version 4, protocole IPV4) permet d'identifier un hôte et un sous-réseau. L'adresse IP est codée sur 4 octets. (les adresses IPV6, ou IP next generation seront codées sur 6 octets).
- L'adresse mac de sa carte réseau (carte ethernet ou carte wifi) ;

Une adresse IP permet d'identifier un hôte. Une passerelle est un ordinateur qui possède plusieurs interfaces et qui transmet les paquets d'une interface à l'autre. La passerelle peut ainsi faire communiquer différents réseaux. Chaque carte réseau possède une adresse MAC unique garantie par le constructeur. Lorsqu'un ordinateur a plusieurs plusieurs interfaces, chacune possède sa propre adresse MAC et son adresse IP. On peut voir sa configuration réseau par `ifconfig`.

```
$ /sbin/ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:B2:3A:24:F3:C4
          inet addr:192.168.0.2  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::2c0:9fff:fef9:95b0/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:16 errors:0 dropped:0 overruns:0 carrier:5
          collisions:0 txqueuelen:1000
          RX bytes:1520 (1.4 KiB)  TX bytes:2024 (1.9 KiB)
          Interrupt:10
```

On voit l'adresse MAC `00:B2:3A:24:F3:C4` et l'adresse IP 192.168.0.2. Cela signifie que le premier octet de l'adresse IP est égal à 192, le deuxième 168, le troisème octet est nul, et le quatrième vaut 2.

Dans un programme C, les 4 octets d'une adresse IP peuvent être stockés dans un `unsigned int`. On peut stocker toutes les données d'adresse dans une structure `in_addr`. On peut traduire l'adresse IP en une chaîne de caractère (avec les octets écrits en décimal et séparés par des points, exemple : "192.168.0.2") par la fonction `inet_ntoa` :

```
char * inet_ntoa(struct in_addr adresse);
```

Inversement, on peut traduire une chaîne de caractère représentant une adresse IP en `struct in_addr`, en passant la structure par adresse à la fonction `inet_aton` :

```
int inet_aton(const char *chaine, struct in_addr *adresse);
```

8.2 Protocoles

Un paquet de données à transmettre dans une application va se voir ajouter, suivant le protocole, les données nécessaires pour

- le routage (détermination du chemin parcouru par les données jusqu'à destination) ;
- la vérification de l'intégrité des données (c'est à dire la vérification qu'il n'y a pas eu d'erreur dans la transmission).

Pour le routage, les données sont par exemple l'adresse IP de la machine de destiation ou l'adresse MAC de la carte d'une passerelle. Ces données sont rajoutées a paquet à transmettre à travers différentes **couches**, jusqu'à la couche physique (câbles) qui transmet effectivement les données d'un ordinateur à l'autre.

8.2.1 La listes des protocles connus du systèmes

Un protocole (IP, TCP, UDP,...) est un mode de communication réseau, c'est à dire une manière d'établir le contact entre machine et de transférer les données. Sous linux, la liste des protocoles reconnus par le système se trouve dans le fichier `/etc/protocols`.

```
$ cat /etc/protocols
# Internet (IP) protocols
ip      0       IP              # internet protocol, pseudo protocol number
#hopopt 0       HOPOPT          # IPv6 Hop-by-Hop Option [RFC1883]
icmp    1       ICMP            # internet control message protocol
igmp    2       IGMP            # Internet Group Management
ggp     3       GGP             # gateway-gateway protocol
ipencap 4       IP-ENCAP        # IP encapsulated in IP (officially ``IP'')
st      5       ST              # ST datagram mode
tcp     6       TCP             # transmission control protocol
egp     8       EGP             # exterior gateway protocol
igp     9       IGP             # any private interior gateway (Cisco)
```

```
pup     12      PUP             # PARC universal packet protocol
udp     17      UDP             # user datagram protocol
hmp     20      HMP             # host monitoring protocol
xns-idp 22      XNS-IDP         # Xerox NS IDP
rdp     27      RDP             # "reliable datagram" protocol
etc...          etc...
```

A chaque protocole est associé un numéro d'identification standard. Le protocle IP est rarement utilisé directement dans une application et on utilise le plus couramment les protocoles TCP et UDP.

8.2.2 Le protocole TCP

Le protocole TCP sert à établir une communication fiable entre deux hôtes. Pour cela, il assure les fonctionnalités suivantes :

- Connexion. L'émetteur et le récepteur se mettent d'accord pour établir un connexion. La connexion reste ouverte jusqu'à ce qu'on la referme.
- Fiabilité. Suite au transfert de données, des tests sont faits pour vérifier qu'il n'y a pas eu d'erreur dans la transmission. Ces tests utilisent la redondance des données, c'est à dire qu'un partie des données est envoyée plusieurs fois. De plus, les données arrivent dans l'ordre où elles ont été émises.
- Possiblilité de communiquer sous forme de flot de données, comme dans un tube (par exemple avec les fonctions `read` et `write`). Les paquets arrivent à destination dans l'ordre où ils ont été envoyés.

8.2.3 Le protocole `UDP`

Le protocole `UDP` permet seulement de transmettre les paquets sans assurer la fiabilité :

- Pas de connexion préalable ;
- Pas de controle d'intégrité des données. Les données ne sont envoyées qu'un fois ;
- Les paquets arrivent à destination dans le désordre.

8.3 Services et ports

Il peuty avoir de nombreuses applications réseau qui tournent sur la même machine. Les numéros de port permettent de préciser avec quel programme nous souhaitons dialoguer par le réseau. Chaque application qui souhaite utiliser les services de la couche IP se voit attribuer un uméro de port. Un numéro de port est un entier sur 16 bits (deux octets). Dans un programme C, on peut stocker un numéro de port dans un `unsigned short`. Il y a un certain nombre de ports qui sont réservés à des services standards. Pour connaître le numéro de port correspondant à un service tel que `ssh`, on peut regarder dans le fichier `/etc/services`.

```
# Network services, Internet style
tcpmux          1/tcp                           # TCP port service multiplexer
echo            7/tcp
echo            7/udp
discard         9/tcp           sink null
discard         9/udp           sink null
systat          11/tcp          users
daytime         13/tcp
daytime         13/udp
netstat         15/tcp
qotd            17/tcp          quote
msp             18/tcp                          # message send protocol
msp             18/udp
chargen         19/tcp          ttytst source
chargen         19/udp          ttytst source
ftp-data        20/tcp
ftp             21/tcp
fsp             21/udp          fspd
ssh             22/tcp                          # SSH Remote Login Protocol
ssh             22/udp
telnet          23/tcp
smtp            25/tcp          mail
time            37/tcp          timserver
time            37/udp          timserver
rlp             39/udp          resource        # resource location
nameserver      42/tcp          name            # IEN 116
whois           43/tcp          nicname
tacacs          49/tcp                          # Login Host Protocol (TACACS)
tacacs          49/udp
re-mail-ck      50/tcp                         # Remote Mail Checking Protocol
re-mail-ck      50/udp
domain          53/tcp          nameserver      # name-domain server
domain          53/udp          nameserver
mtp             57/tcp                          # deprecated
etc...
```

L'administrateur du système peut définir un nouveau service en l'ajoutant dans `/etc/services` et en précisant le numéro de port. Les numéros de port inférieurs à 1024 sont réservés aux serveurs et démons lancés par root (éventuellement au démarage de l'ordinateur), tels que le serveur d'impression `/usr/sbin/cupsd` sur le port ou le serveur ssh `/usr/sbin/sshd` sur le port 22.

8.4 Sockets TCP

Dans cette partie, nous nous limitons aux sockets avec protocole TCP/IP, c'est à dire un protocole TCP (avec connexion préalable et vérification des données), fondé sur IP (c'est à dire utilisant la couche IP). Pour utiliser d'autres protocoles (tel que UDP), il faudrait mettre

d'autres options dans les fonctions telles que `socket`, et utiliser d'autres fonctions que `read` et `write` pour transmettre des données.

8.4.1 Création d'une socket

Pour créer une socket, on utilise la fonction `socket`, qui nous retourne un identifiant (de type `int`) pour la socket. Cet identifiant servira ensuite à désigner la socket dans la suite du programme (comme un pointeur de fichiers de type `FILE*` sert à désigner un fichier). Par exemple, pour une socket destinée à être utilisée avec un protocole `TCP/IP` (avec connexion TCP) fondé sur IP (`AF_INET`), on utilise

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
```

Cette socket est destinée à permettre à une autre machine de dialoguer avec le programme.

On précise éventuellement l'adresse IP admissible (si l'on souhiate faire un contrôle sur l'adresse IP) de la machine distante, ainsi que le port utilisé. On lie ensuite la socket `sock` à l'adresse IP et au port en question avec la fonction `bind`. On passe par adresse l'addresse de la socket (de type `struct sockaddr_in`) avec un cast, et la taille en octets de cette structure (revoyée par `sizeof`).

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


#include <unistd.h>

#include <arpa/inet.h>
#include <netdb.h>
#include <netinet/in.h>

#include <sys/types.h>
#include <sys/socket.h>

#define BUFFER_SIZE 256

int cree_socket_tcp_ip()
{
  int sock;
  struct sockaddr_in adresse;

  if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
      fprintf(stderr, "Erreur socket\(\backslash\)n");
      return -1;
    }

  memset(&adresse, 0, sizeof(struct sockaddr_in));
```

```
  adresse.sin_family = AF_INET;
  // donner un num ro de port disponible quelconque
  adresse.sin_port = htons(0);
  // aucun contr le sur l'adresse IP :
  adresse.sin_addr.s_addr = htons(INADDR_ANY);

  // Autre exemple :
  // connexion sur le port 33016 fix
  // adresse.sin_port = htons(33016);
  // depuis localhost seulement :
  // inet_aton("127.0.0.1", &adresse.sin_addr);

  if (bind(sock, (struct sockaddr*) &adresse,
           sizeof(struct sockaddr_in)) < 0)
  {
    close(sock);
    fprintf(stderr, "Erreur bind\(\backslash\)n");
    return -1;
  }

  return sock;
}
```

8.4.2 Affichage de l'adresse d'une socket

Après un appel à `bind`, on peut retrouver les données d'adresse et de port par la fonction `getsockname`. Les paramètres dont pratiquement les mêmes que pour `bind` sauf que le nombre d'octets est passé par adresse. On peut utiliser les fonctions `ntoa` et `ntohs` pour afficher l'adresse IP et le port de manière compréhensible par l'utilisateur.

```
int affiche_adresse_socket(int sock)
{
  struct sockaddr_in adresse;
  socklen_t longueur;

  longueur = sizeof(struct sockaddr_in);
  if (getsockname(sock, (struct sockaddr*)&adresse, &longueur) < 0)
    {
      fprintf(stderr, "Erreur getsockname\(\backslash\)n");
      return -1;
    }
  printf("IP = %s, Port = %u\(\backslash\)n", inet_ntoa(adresse.sin_addr),
         ntohs(adresse.sin_port));
  return 0;
}
```

8.4.3 Implémentation d'un serveur TCP/IP

Un serveur (réseau) est une application qui va attendre que d'autres programmes (sur des machines distantes), appelés clients, entrent en contact avec lui, et dialoguent avec lui. Pour créer un serveur TCP/IP avec des sockets, on crée d'abord la socket avec `socket` et `bind`. On indique ensuite au noyau `linux/unix` que l'on attend une connection sur cette socket. Pour cela, on utilise la fonction `listen` qui prend en paramètre l'identifiant de la socket et la taille de la file d'attente (en général 5 et au plus 128) au cas ou plusieurs clients se présenteraient au même moment.

Le serveur va ensuite boucler dans l'attente de clients, et attendre une connexion avec l'appel système `accept`. La fonction `accept` **crée une nouvelle socket pour le dialogue avec le client**. En effet, la socket initiale doit rester ouverte et en attente pour la connection d'autres clients. Le dialogue avec le client se fera donc avec une nouvelle socket, qui est retournée par `accept`.

Ensuite, le serveur appelle `fork` et crée un processus fils qui va traîter le client, tandis que le processus père var boucler à nouveau sur `accept` dans l'attente du client suivant.

```
int main(void)
{
  int sock_contact;
  int sock_connectee;
  struct sockaddr_in adresse;
  socklen_t longueur;
  pid_t pid_fils;

  sock_contact = cree_socket_tcp_ip();
  if (sock_contact < 0)
    return -1;
  listen(sock_contact, 5);
  printf("Mon adresse (sock contact) -> ");
  affiche_adresse_socket(sock_contact);
  while (1)
    {
      longueur = sizeof(struct sockaddr_in);
      sock_connectee = accept(sock_contact,
                              (struct sockaddr*)&adresse,
                              &longueur);
      if (sock_connectee < 0)
        {
          fprintf(stderr, "Erreur accept\(\backslash\)n");
          return -1;
        }
      pid_fils = fork();
      if (pid_fils == -1)
        {
          fprintf(stderr, "Erreur fork\(\backslash\)n");
          return -1;
        }
```

```
      if (pid_fils == 0)  {/* fils */
        {
          close(sock_contact);
          traite_connection(sock_connectee);
          exit(0);
        }
      else
        close(sock_connectee);
    }
  return 0;
}
```

8.4.4 Traîtement d'une connexion

Une fois la connexion établie, le serveur (ou son fils) peut connaître les données d'adresse IP et de port du client par la fonction `getpeername`, qui fonction comme `getsockname` (vue plus haut). Le programme dialogue ensuite avec le client avec les fonctions `read` et `write` comme dans le cas d'un tube.

```
void traite_connection(int sock)
{
  struct sockaddr_in adresse;
  socklen_t  longueur;
  char bufferR[BUFFER_SIZE];
  char bufferW[BUFFER_SIZE];
  int nb;

  longueur = sizeof(struct sockaddr_in);
  if (getpeername(sock, (struct sockaddr*) &adresse, &longueur) < 0)
    {
      fprintf(stderr, "Erreur getpeername\(\backslash\)n");
      return;
    }
  sprintf(bufferW, "IP = %s, Port = %u\(\backslash\)n",
          inet_ntoa(adresse.sin_addr),
          ntohs(adresse.sin_port));
  printf("Connexion : locale (sock_connectee) ");
  affiche_adresse_socket(sock);
  printf("   Machine distante : %s", bufferW);
  write(sock, "Votre adresse : ", 16);
  write(sock, bufferW, strlen(bufferW)+1);
  strcpy(bufferW, "Veuillez entrer une phrase : ");
  write(sock, bufferW, strlen(bufferW)+1);
  nb= read(sock, bufferR, BUFFER_SIZE);
  bufferR[nb-2] = '\(\backslash\)0';
  printf("L'utilsateur distant a tap  : %s\(\backslash\)n", bufferR);
```

```
  sprintf(bufferW, "Vous avez tap  : %s\(\backslash\)n", bufferR);
  strcat(bufferW, "Appuyez sur entree pour terminer\(\backslash\)n");
  write(sock, bufferW, strlen(bufferW)+1);
  read(sock, bufferR, BUFFER_SIZE);
}
```

8.4.5 Le client telnet

Un exemple classique de client est le programme `telnet`, qui affiche les données reçues sur sa sortie standard et envoie les données saisies dans son entrée standard dans la socket. Cela permet de faire un système client-serveur avec une interface en mode texte pour le client.

Ci-dessous un exemple, avec à gauche le côté, et à droite le côté client (connecté localement).

```
$ ./serveur
Mon adresse (sock contact) -> IP = 0.0.0.0, Port = 33140
                                        $   telnet localhost 33140
                                        Trying 127.0.0.1...
                                        Connected to portable1.
                                        Escape character is '^]'.
                                        Votre adresse : IP = 127.0.0.1, Port = 33141
                                        Veuillez entrer une phrase :
Connexion : locale (sock_connectee) IP = 127.0.0.1, Port = 33140
  Machine distante : IP = 127.0.0.1, Port = 33141
                                        Veuillez entrer une phrase : coucou
                                        Vous avez tapé : coucou

                                        Connection closed by foreign host.
L'utilsateur distant a tapé : coucou
```

Ci-dessous un autre exemple, avec à gauche le côté, et à droite le côté client (connecté à distance).

```
$ ./serveur
Mon adresse (sock contact) -> IP = 0.0.0.0, Port = 33140
                                        $   telnet 192.168.0.2 33140
                                        Trying 192.168.0.2...
                                        Connected to 192.168.0.2.
                                        Escape character is '^]'.
                                        Votre adresse : IP = 192.168.0.5, Port = 34353
                                        Veuillez entrer une phrase :

Connexion : locale (sock_connectee) IP = 127.0.0.1, Port = 33140
  Machine distante : IP = 192.168.0.5, Port = 33141
                                        Veuillez entrer une phrase : test
                                        Vous avez tapé : test

                                        Connection closed by foreign host.
```

```
L'utilsateur distant a tapé : test
```

8.5 Créer une connection client

Jusqu'à maintenant, nous n'avons pas programmé de client. Nous avons simplement tuilisé le client telnet. L'exemple ci-dessous est une client qui, alternativement :

1. lit une chaîne dans la socket et l'affiche sur sa sortie standard ;
2. saisit une chaîne sur son entrée standard et l'envoie dans la socket.

C'est un moyen simple de créer une inetrface client.

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

#include <arpa/inet.h>
#include <netdb.h>
#include <netinet/in.h>

#include <sys/types.h>
#include <sys/socket.h>

#define BUFFER_SIZE 256

int cree_socket_tcp_client(int argc, char** argv)
{
  struct sockaddr_in adresse;
  int sock;

  if (argc != 3)
    {
      fprintf(stderr, "Usage : %s adresse port\(\backslash\)n", argv[0]);
      exit(0);
    }
  if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
      fprintf(stderr, "Erreur socket\(\backslash\)n");
      return -1;
    }

  memset(&adresse, 0, sizeof(struct sockaddr_in));
  adresse.sin_family = AF_INET;
  adresse.sin_port = htons(atoi(argv[2]));
  inet_aton(argv[1], &adresse.sin_addr);
```

```
  if (connect(sock, (struct sockaddr*) &adresse,
          sizeof(struct sockaddr_in)) < 0)
  {
    close(sock);
    fprintf(stderr, "Erreur connect\(\backslash\)n");
    return -1;
  }
  return sock;
}

int affiche_adresse_socket(int sock)
{
  struct sockaddr_in adresse;
  socklen_t longueur;

  longueur = sizeof(struct sockaddr_in);
  if (getsockname(sock, (struct sockaddr*)&adresse, &longueur) < 0)
    {
      fprintf(stderr, "Erreur getsockname\(\backslash\)n");
      return -1;
    }
  printf("IP = %s, Port = %u\(\backslash\)n", inet_ntoa(adresse.sin_addr),
         ntohs(adresse.sin_port));
  return 0;
}

int main(int argc, char**argv)
{
  int sock;
  char buffer[BUFFER_SIZE];
  sock = cree_socket_tcp_client(argc, argv);
  if (sock < 0)
    {
      puts("Erreur connection socket client");
      exit(1);
    }
  affiche_adresse_socket(sock);
  while(1)
    {
      if (read(sock, buffer, BUFFER_SIZE)==0)
        break;
      puts(buffer);
      if (fgets(buffer, BUFFER_SIZE, stdin) == NULL)
        break;
      buffer[strlen(buffer)-1] = '\(\backslash\)0';
      write(sock, buffer, BUFFER_SIZE);
    }
```

```
  return 0;
}
```

Le serveur correspondant est programmé comme le serveur TCP précédent, sauf que la fonction `traite_connection` doit être modifiée pour tenir compte du fonctionnement du client (alternance des lectures et écritures dans la socket).

```
void traite_connection(int sock)
{
  struct sockaddr_in adresse;
  socklen_t  longueur;
  char bufferR[BUFFER_SIZE];
  char bufferW[BUFFER_SIZE];
  char tmp[50];
  int nb;

  longueur = sizeof(struct sockaddr_in);
  if (getpeername(sock, (struct sockaddr*) &adresse, &longueur) < 0)
    {
      fprintf(stderr, "Erreur getpeername\(\backslash\)n");
      return;
    }
  sprintf(bufferW, "IP = %s, Port = %u\(\backslash\)n",
          inet_ntoa(adresse.sin_addr),
          ntohs(adresse.sin_port));
  printf("Connexion : locale (sock_connectee) ");
  affiche_adresse_socket(sock, tmp);
  printf(tmp);
  printf("   Machine distante : %s", bufferW);
  strcat(bufferW, "Votre adresse : ");
  affiche_adresse_socket(sock, tmp);
  strcat(bufferW, tmp);

  strcat(bufferW, "Veuillez entrer une phrase : ");
  write(sock, bufferW, BUFFER_SIZE);
  nb= read(sock, bufferR, BUFFER_SIZE);
  printf("L'utilsateur distant a tap  : %s\(\backslash\)n", bufferR);
  sprintf(bufferW, "Vous avez tap  : %s\(\backslash\)n", bufferR);
  write(sock, bufferW, BUFFER_SIZE);
}
```

8.6 Exercices

Exercice 8.1 (**) Le but de l'exercice est d'écrire un serveur TCP/IP avec client `telnet` qui gère une base de données de produits et des clients qui font des commandes. Chaque client se connecte au serveur, entre le nom du (ou des) produit(s) commandé(s), les quantités, et son nom. Le serveur affiche le prix de la commande, et crée un fichier dont le nom est unique (par

exemple créé en fonction de la date) qui contient les données de la commande. La base de données est stockée dans un fichier texte dont chaque ligne contient un nom de produit (sans espace), et un prix unitaire.

a) Définir une structure produit contenant les données d'un produit.

b) Écrire une fonction de chargement de la base de données en mémoire dans un tableau de structures.

c) Écrire une fonction qui renvoie un pointeur sur la structure (dans le tableau) correspondant à un produit dont le nom est passé en paramètre.

d) Écrire le serveur qui va gérer les commandes de un clients. Le serveur saisit le nom d'un produit et les quantités via une socket, recherche le prix du produit, et affiche le prix de la commande dans la console du client connecté par telnet.

e) Même question en supposant que le client peut commander plusieurs produits dans une même commande.

f) Modifier le serveur pour qu'il enregistre les données de la commande dans un fichier. Le serveur crée un fichier dont le nom est unique (par exemple créé en fonction de la date).

g) Quel reproche peut-on faire à ce programme concernant sa consommation mémoire ? Que faudrait-il faire pour gérer les informations sur les stocks disponibles des produits ?

Exercice 8.2 (∗∗) **a)** Écrire un serveur TCP/IP qui vérifie que l'adresse IP du client se trouve dans un fichier `add_autoris.txt`. Dans le fichier, les adresse IP autorisées sont écrites lignes par lignes.

b) Modifier le programme précédent pour que le serveur souhaite automatiquement la bienvenue au client en l'appelant par son nom (écrit dans le fichier sur la même ligne que l'adresse IP.

Exercice 8.3 (∗) Ecrire un système client serveur qui réalise la chose suivante :

- Le client prend en argument un chemin vers un fichier texte local ;
- Le serveur copie ce fichier texte dans un répertoire `/home/save/` sous un nom qui comprend l'adresse IP du client et la date au format `aaaa_mm_jj`.

Pour la gestion des dates, on utilisera les fonctions suivantes de la bibliothèque `time.h`

```
// la fonction suivante renvoie la date
time_t time(time_t *t);

// la fonction suivante traduit la date
struct tm *localtime(const time_t *t);

// la structure tm contenant les données de date
// est la suivante :
struct tm \{
```

```
  int tm_sec;   /* Secondes */
  int tm_min;   /* Minutes */
  int tm_hour;  /* Heures (0 - 23) */
  int tm_mday;  /* Quantième du mois (1 - 31) */
  int tm_mon;   /* Mois (0 - 11) */
  int tm_year;  /* An (année calendaire - 1900) */
  int tm_wday;  /* Jour de semaine (0 - 6  Dimanche = 0) */
  int tm_yday;  /* Jour dans l'année (0 - 365) */
  int tm_isdst; /* 1 si "daylight saving time" */
\};
```

Exercice 8.4 (∗∗) Même question qu'à l'exercice 8.3, mais cette fois le client donne le chemin vers un répertoire contenant des fichiers texte ou binaires. On pourra créer une archive correspondant au répertoire dans `/tmp` :

```
# coté client :
$ tar zcvf /tmp/rep.tgz /chemin/vers/le/repertoire/client/

# coté serveur :
$ cd /chemin/vers/le/repertoire/serveur/ ; tar zxvf rep.tgz ; rm rep.tgz
```

Exercice 8.5 (∗∗) Créez un serveur de messagerie. Le serveur met en relation les clients deux à deux à mesure qu'ils se connectent. Une fois que les clients sont en relation, chaque client peut alternativement saisir une phrase et lire une phrase écrite par l'autre client. Le serveur affiche chez les client :

```
L'autre client dit : coucou
Saisissez la réponse :
```

Exercice 8.6 (∗ ∗ ∗) Créez un forum de chat. Le serveur met tous les clients connectés en relation. Chaque client demande à parler en tapant 1. Une seul client peut parler à la fois. Lorsque le client envoie une message, le serveur affiche l'adresse *IP* du client et le message chez tous les autres clients. Les clients sont traités par des threads et l'accès aux sockets en écriture est protégé par un mutex.

Annexe A

Compilation séparée

A.1 Variables globales

Une variable globale est une variable qui est définie en dehors de toute fonction. Une variable globale déclarée au début d'un fichier source peut être utilisée dans toutes les fonctions du fichier. La variable n'existe qu'en un seul exemplaire et la modification de la variable globale dans une fonction change la valeur de cette variable dans les autres fonctions.

```
#include <stdio.h>

inte ;  /* déclaration en dehors de toute fonction */

\void ModifieDonneeGlobale(\void)/* pas de paramètre */
{
   x = x+1;
}

void AfficheDonneGlobale(void)  /* pas de paramètre */
{
   printf("%d\n", x);
}

int main(void)
{
   x = 1;
   ModifieDonneeGlobale();
   AfficheDonneGlobale();   /* affiche 2 */
   return 0;
}
```

Dans le cas d'un projet avec programmation multifichiers, on peut utiliser dans un fichier source une variable globale définie dans un autre fichier source en déclarant cette variable avec le mot clef `extern` (qui signifie que la variable globale est définie ailleurs).

```
extern int x;  /* déclaration d'une variable externe
```

A.2 Mettre du code dans plusieurs fichiers

`Exemple.` Supposons qu'un `TypeArticle` regroupe les données d'un produit dans un magasin. La fonction `main`, dans le fichier `main.c`, appelle les fonctions `SaisitProduit` et `AfficheProduit`, qui sont définies dans le fichier `routines.c`. Les deux fichiers `.c` incluent le fichier `typeproduit.h`

```
/************************************************
|******* HEADER FILE  typeproduit.h ***************|
************************************************/

/* 1) Protection contre les inclusions multiples */

#ifndef MY_HEADER_FILE  /* Evite la définition multiple */
#define MY_HEADER_FILE

/* 2) Définition des constantes et variables globales exportées */

#define STRING_LENGTH 100
extern int erreur;

/* 3) Définition des structures et types */

typedef struct {
           int code;   /* code article */
           char denomination[STRING_LENGTH];  /* nom du produit */
           float prix; /* prix unitaire du produit */
           int stock;  /* stock disponible */
                }TypeArticle;

/* 4) Prototypes des fonctions qui sont */
/* définies dans un fichier mais utilisées */
/* dans un autre */

void SaisitProduit(TypeArticle *adr_prod);
void AfficheProduit(TypeArticle prod);

#endif  /* fin de la protection */
```

La protection `#ifndef` permet d'éviter que le code du fichier header ne soit compilé plusieurs fois, provoquant des erreurs de définitions multiples, si plusieurs fichiers sources incluent le header.

```
/************************************************\
|********** SOURCE FILE  routines.c ***************|
\************************************************/

#include <stdio.h>
```

```
#include "typeproduit.h"  /* attention aux guillemets */

int max_stock=150;

/* La variable globale erreur est définie ici */
int erreur;

/* La fonction suivante est statique */
/* LitChaine n'est pas accessible dans main.c */

static LitChaine(const char *chaine)
{
   fgets(chaine, STRING_LENGTH, stdin);
}

void SaisitProduit(TypeArticle *adr_prod)
{
   printf("Code produit : ");
   scanf("%d", &adr_prod->code)
   printf("Dénomination : ");
   LitChaine(adr_prod->denomination);
   printf("Prix : ");
   scanf("%f", &adr_prod->prix);
   printf("Stock disponible : ");
   scanf("%d", &adr_prod->stock);
   if(adr_prod->stock > max_stock)
       {
 fprintf(stderr, "Erreur, stock trop grand\n");
         erreur=1;
       }
}

void AfficheProduit(TypeArticle prod)
{
   printf("Code : %d\n", prod.code);
   printf("Dénomination : %s\n", prod.denomination);
   printf("Prix : %f\n", prod.prix);
   printf("Stock disponible : %d\n", prod.stock);
}

/************************************************\
|************* SOURCE FILE  main.c ***************|
\************************************************/

#include "typeproduit.h"  /* attention aux guillemets */

int main(void)
```

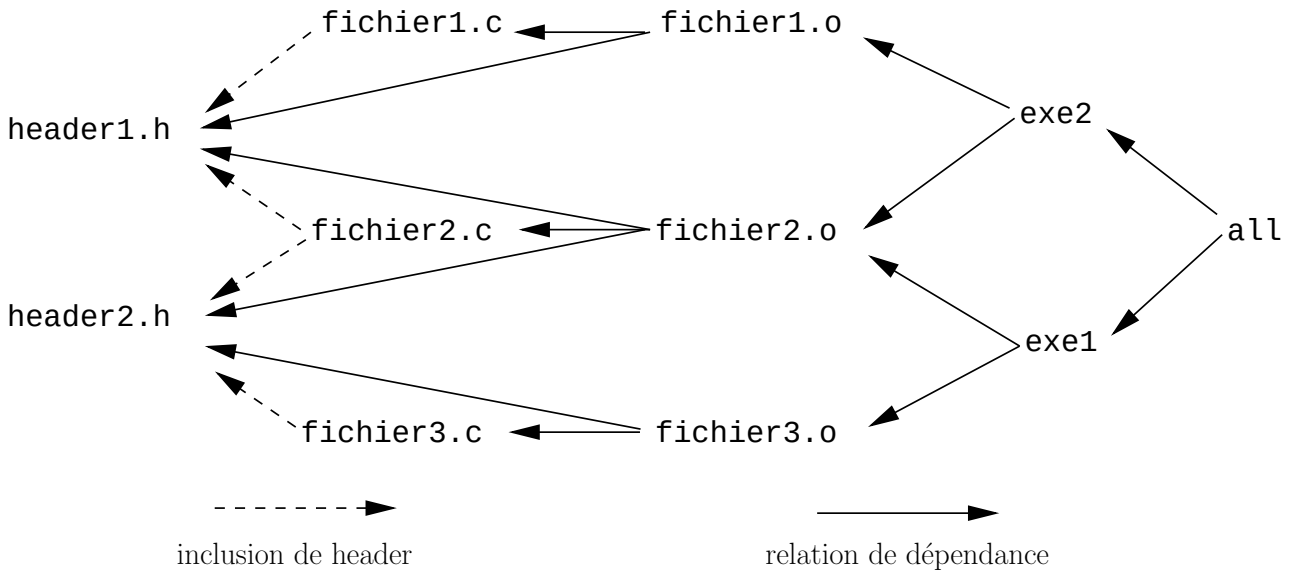

FIGURE A.1 : Exemple de compilation séparée de deux fichiers ecécutables

```
{
   TypeProduit prod;
   erreur=0;  /* variable définie dans un autre fichier */
   SaisitProduit(&prod); /* fonction définie dans un autre fichier */
   if (erreur==0)
      AfficheProduit(prod);
}
```

A.3 Compiler un projet multifichiers

A.3.1 Sans makefile

Pour compiler l'exemple précédent sans `makefile` sous *Linux*, c'est à dire pour créer un fichier exécutable, il faut d'abord créer un fichier *objet* pour chaque fichier source.

```
$ gcc -c routines.c
$ gcc -c main.c
```

Ceci doit générer deux fichiers objets `routines.o` et `main.o`. On crée ensuite l'exécutable (par exemple appelé `produit.exe`) en réalisant l'édition des liens (*link*) par l'instruction suivante :

```
$ gcc routines.o main.o -o produit.exe
```

C'est lors de l'édition des liens que les liens entre les fonctions s'appelant d'un fichier à l'autre, et les variables globales `extern` sont faits. On peut schématiser ce processus de compilation séparée des fichiers sources par la figure A.1, dans laquelle deux fichiers exécutables `exe1` et mttexe2 dépendent respectivement de certains fichiers sources, qui eux même incluent des header.

Les règles de dépendances indiquent que lorsqu'un certain fichier est modifié, certains autres fichiers doivent être recompilés.

Exemple. Sur la figure A.1, si le fichier `header1.h` est modifié, il faut reconstruire `fichier1.o` et `fichier2.o`. Ceci entraine que `exe1.o` et `exe2.o` doivent aussi être reconstruits. En cas de modification de `header1.h`, on doit donc exécuter les commandes suivantes :

```
$ gcc -c fichier1.c
$ gcc -c fichier2.c
$ gcc fichier1.o fihier2.o -o exe1
$ gcc fichier2.o fihier3.o -o exe2
```

A.3.2 Avec `makefile`

Un `makefile` est un moyen qui permet d'automatiser la compilation d'un projet multifichier. Grace au `makefile`, la mise à jours des fichiers objets et du fichier exécutable suite à une modification d'un source se fait en utilisant simplement la commande :

```
$ make
```

Pour cela, il faut spécifier au système les dépendances entre les différents fichiers du projet, en créant un *fichier makefile*.
Exemple 1. Pour l'exemple des fichiers `main.c,routines.c` et `typeproduit.h` ci-dessus, on crée un fichier texte de nom `makefile` contenant le code suivant :

```
produit.exe : routines.o main.o
     gcc routines.o main.o -o produit.exe
routines.o : routines.c typeproduit.h
     gcc -c routines.c
main.o: main.c typeproduit.h
     gcc -c main.c
```

Ce fichier comprend trois parties. Chaque partie exprime une règle de dépendance et une règle de reconstruction. Les règles de reconstruction (lignes 2, 4 et 6) commencent obligatoirement par une tabulation.

Les règles de dépendance sont les suivantes :

- Le fichier exécutable `produit.exe` dépend de tous les fichiers objets. Si l'un des fichier objets est modifié, il faut utiliser la règle de reconstruction pour faire l'édition des liens.
- Le fichier objet `routine.o` dépend du fichier `routine.c` et du fichier `typearticle.h`. Si l'un de ces deux fichiers est modifié, il faut utiliser la règle de reconstruction pour reconstruire `routine.o`
- De même, `main.o` dépend de `main.c` et `typearticle.h`.

Exemple 2. Pour les fichiers de la figure A.1, un makefile simple ressemblerait à ceci :

```
CC=gcc  # variable donnant le compilateur
CFLAGS= -Wall -pedantic -g # options de compilation
all: exe1 exe2 # règle pour tout reconstruire
fichier1.o : fichier1.c header1.h
      $(CC) $(CFLAGS) -c fichier1.c
```

```
fichier2.o : fichier2.c header1.h header2.h
      $(CC) $(CFLAGS) -c fichier2.c
fichier3.o : fichier3.c header2.h
      $(CC) $(CFLAGS) -c fichier3.c
exe1 : fichier1.o fichier2.o
      $(CC) $(CFLAGS) fichier1.o fichier2.o -o exe1
exe2 : fichier2.o fichier3.o
      $(CC) $(CFLAGS) fichier3.o fichier3.o -o exe2
clean:
      rm *.o exe1 exe2
```

Utilisation :

```
$ make exe1  # reconstruit la cible exe1
$ make all   # reconstruit tout
$ make clean # fait le ménage
$ make  # reconstruit ma première cible (en l'occurence all)
```