Programmation système et réseaux

Rémy Malgouyres
LAIC, IUT, département info
B.P. 86
63172 AUBIERE cedex
http ://laic.u-clermont1.fr/~mr

Table des matières

1 Création d'un processus : `fork` 2
- 1.1 Processus, *PID*, *UID* . . . 2
- 1.2 La fonction `fork` . . . 3
- 1.3 Exercices . . . 4

2 Lancement d'un programme : `exec` 6
- 2.1 Arguments en ligne de commande . . . 6
- 2.2 La fonction `execv` . . . 7
- 2.3 La fonction `system` . . . 8
- 2.4 Exercices . . . 10

3 Threads Posix 12
- 3.1 Pointeurs de fonction . . . 12
- 3.2 Thread Posix (sous linux) . . . 15
- 3.3 Donnée partagées et exclusion mutuelle . . . 18
- 3.4 Exercices . . . 20

4 Communication entre processus 23
- 4.1 Tubes et `fork` . . . 23
- 4.2 Rediriger les flots d'entrées-sorties vers des tubes . . . 25
- 4.3 Exercices . . . 25

5 Programmation réseaux 27
- 5.1 Adresses *IP* et *MAC* . . . 27
- 5.2 Protocoles . . . 28
- 5.3 Services et ports . . . 29
- 5.4 Sockets *TCP* . . . 30
- 5.5 Exercices . . . 35

A Rappels sur le langage *C* 37
- A.1 Le générateur aléatoire . . . 37
- A.2 `atoi`, `sprinf` et `sscanf` . . . 38
- A.3 Variables globales . . . 39

Chapitre 1

Création d'un processus : `fork`

1.1 Processus, *PID*, *UID*

1.1.1 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.

1.1.2 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
```

1.2 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;
}
```

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 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**.

1.3 Exercices

Exercice 1.1 (∗) Ecrire un programme qui crée un fils. Le père doit afficher "je suis le père" et le fils doit afficher "je suis le fils".

Exercice 1.2 (∗) Ecrire 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 1.3 (∗) Ecrire 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 1.4 (∗∗) Ecrire 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 1.5 (∗∗) Ecrire 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 2

Lancement d'un programme : exec

2.1 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;
}
```

2.2 La fonction execv

La fonction execv 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 execv. Il y a en fait plusieurs fonctions exec qui sont légèrement différentes. Nous allons étudier l'une d'entre elles (la fonction execv). 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 affiche la liste des fichiers *.c* et *.h* du répertoire de travail, équivalent à la commande :

```
$ ls *.c *.h
```

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

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

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

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

```
    return 0;
}
```

Remarque 2.2.1 *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`

2.3 La fonction `system`

2.3.1 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.

2.3.2 La commande `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.

2.3.3 Applications suid et problèmes des sécurité liés `system`

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.

2.4 Exercices

Exercice 2.1 (∗) 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 2.2 (∗) Ecrire un programme qui prend en argument un chemin vers un répertoire R, et copie le répertoire courant dans ce répertoire R.

Exercice 2.3 (∗) 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 2.4 (∗∗) 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 2.5 (∗∗) 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 2.6 $(***)$ 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 calculer C_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 3

Threads Posix

3.1 Pointeurs de fonction

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\n", n);
}

int main(void)
{
  void (*foncAff)(int); /* déclaration d'un pointeur foncAff */
  int (*foncSais)(void); /*déclaration d'un pointeur foncSais */
  int 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 */
  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\n", n);
}

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

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)
{
   int 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)
{
   int 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);
   ...
}
```

3.2 Thread Posix (sous linux)

3.2.1 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. Sous linux, chaque thread donne lieu à un processus indépendant. 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
```

3.2.2 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 $\texttt{pthread}_\texttt{t}$). 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`$_{\texttt{f}}$`onction`$_{\texttt{t}}$`hread`, 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 `passageparadressed'unpointeur` qui permet de récupérer la valeur retournée par `ma_fonction_thread`.

3.2.3 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\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 crée le thread */
  ret = pthread_create(&my_thread, NULL,
```

```
                        ma_fonction_thread, (void*)nbsec);
  if (ret != 0)
    {
      fprintf(stderr, "Erreur de création du thread");
      exit (1);
    }
  pthread_join(my_thread, NULL); /* on attend la fin du thread */
  printf("Dans le main, nbsec = %d\n", nbsec);
  printf("Duree de l'operation = %d\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\n");
  scanf("%d", &resultat);
  pthread_exit((void*)resultat);  /* termine le thread proprement */
}

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

3.3 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 numéro %d commence son calcul\n", num_thread);
      for (j=0 ; j<n ; j++)
        for (k=0 ; k<n ; k++)
          {}
      printf("Le thread numero %d a fini son calcul\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.

3.4 Exercices

Exercice 3.1 (∗) Écrire un programme qui crée un thread qui prend en paramètre un tableau d'entiers et l'affiche dans la console.

Exercice 3.2 (∗) 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 3.3 (∗∗) 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 3.4 (∗∗) **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 3.5 (∗∗) 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 3.6 (∗∗) Créer un programme qui a en vaiable 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 $i = 1, 2, \ldots, 98$ par la moyenne
$$(\texttt{tableau}[\texttt{i}-1] + \texttt{tableau}[\texttt{i}] + \texttt{tableau}[\texttt{i}+1])/3$$
Il attend ensuite un temps alléatoire entre 1 et 3 secondes ;
- Le deuxième thread affiche le tableau toutes les 4 secondes.

Exercice 3.7 (∗∗) 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 T_1 et T_2. Dans chaque thread T_i, 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.

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 '\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("Création d'un tube");
  if (pipe(tube) != 0)   /* pipe */
    {
      fprintf(stderr, "Erreur dans pipe\n");
      exit(1);
    }
  pid_fils = fork();     /* fork */
  if (pid_fils == -1)
    {
      fprintf(stderr, "Erreur dans fork\n");
      exit(1);
    }
  if (pid_fils == 0) /* processus fils */
    {
      printf("Fermeture entrée dans le fils (pid = %d)\n", getpid());
      close(tube[1]);
      read(tube[0], bufferR, BUFFER_SIZE);
      printf("Le fils (%d) a lu : %s\n", getpid(), bufferR);
    }
  else               /* processus père */
    {
      printf("Fermeture sortie dans le père (pid = %d)\n", getpid());
      close(tube[0]);
      sprintf(bufferW, "Message du père (%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 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);
```

4.3 Exercices

Exercice 4.1 (∗) Écrire un programme qui crée deux processus. Le processus père saisit une valeur de type `double` au clavier, calcule son sinus, et le transmets au processus fils, qui affiche la valeur du sinus.

Exercice 4.2 (∗∗) 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.

Exercice 4.3 (∗∗) 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, et transmet un message au fils par le tube. Le programme fils affiche le message.

Exercice 4.4 (∗ ∗ ∗) Reprendre les programmes de l'exercice 4.3. Nous allons faire un programme qui fait la même chose, mais transmet les données différement. Dans le programme père, on liera `stdout` à l'entrée du tube. Dans le programme fils, on liera `stdin` à la sortie du tube.

Chapitre 5

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.

5.1 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);
```

5.2 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.

5.2.1 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.

5.2.2 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.

5.2.3 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.

5.3 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 `unsignedshort`. 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.

5.4 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.

5.4.1 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 <unistd.h>

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

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

#define BUFFER_SIZE 1000

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

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

  memset(&adresse, 0, sizeof(struct sockaddr_in));
  adresse.sin_family = AF_INET;
  // donner un numéro de port disponible quelconque
  adresse.sin_port = htons(0);
  // aucun contrôle sur l'adresse IP :
  adresse.sin_addr.s_addr = htons(INADDR_ANY);

  // Autre exemple :
  // connexion sur le port 33016 fixé
  // 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\n");
    return -1;
  }

  return sock;
}
```

5.4.2 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\n");
      return -1;
    }
  printf("IP = %s, Port = %u\n", inet_ntoa(adresse.sin_addr),
         ntohs(adresse.sin_port));
  return 0;
}
```

5.4.3 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\n");
          return -1;
        }
      pid_fils = fork();
      if (pid_fils == -1)
        {
          fprintf(stderr, "Erreur fork\n");
          return -1;
        }
      if (pid_fils == 0)  /* fils */
        {
          close(sock_contact);
          traite_connection(sock_connectee);
          exit(0);
        }
      else
        close(sock_connectee);
    }
  return 0;
}
```

5.4.4 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\n");
      return;
    }
  sprintf(bufferW, "IP = %s, Port = %u\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] = '\0';
  printf("L'utilsateur distant a tapé : %s\n", bufferR);
  sprintf(bufferW, "Vous avez tapé : %s\n", bufferR);
  write(sock, bufferW, strlen(bufferW)+1);
}
```

5.4.5 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
```

5.5 Exercices

Exercice 5.1 (**) 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 5.2 (∗∗) **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.

Annexe A

Rappels sur le langage C

A.1 Le générateur aléatoire

On tire un nombre aléatoire avec la fonction `rand` de la bibliothèsque `stdlib.h`, qui retourne un nombre aléatoire entre 0 et `RAND_MAX` (défini comme égal à 2147483647 dans `stdlib.h`). On peut faire appel à un modulo ou à un facteur d'échelle pour avoir un nombre dans une fourchette donnée.

Cepedant, la fonction `rand` est implémentée par un algorithme déterministe, et si l'on souhaite obtenir toujours des nombres différentes, le générateur aléatoire doit être initialisé en fonction de l'heure. Pour celà, on utilise la fonction `srand`, qui permet d'initialiser le générateur aléatoire à une certaine valeur, et on peut utiliser la bibliothèque `time.h`, et par exemple la fonction `time`, qui retourne le nombre de secondes depuis le premier janvier 1970 à $0h$ (en temps universel).

Exemple. Le programme suivant affiche uns série de 10 nombres réels aléatoires entre 0 et 1 :

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

int main(void)
{
  int i;
  double x;
  srand(time(NULL));
  for (i=0 ; i<10 ; i++)
    {
      x = rand()/((double)RAND_MAX);
      printf("%.4f\n", x);
    }
  return 0;
}
```

A.2 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);  /* saisie d'une chaîne de caractères */
   a = atoi(s);  /* 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;
}
```

A.3 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>

int x ;  /* 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 */
```