Programmation sous Linux

Thomas VANTROYS
Centre de Recherche en Informatique, Signal et Automatique de Lille / Équipe 2XS
École Polytechnique Universitaire de Lille
Université de Lille
59655 Villeneuve d'Ascq, France

email : **thomas.vantroys@univ-lille.fr**

13 janvier 2020

Table des matières

Chapitre 1

Développement sous Linux

Real Users hate Real Programmers

1.1 Présentation

Dans ce chapitre, nous allons voir quelques outils de base nous permettant de développer en C sous Linux. Nous commencerons par voir les principes des bibliothèques de programmation et les utilitaires associés, puis nous parlerons de `GCC` le compilateur C que nous utiliserons en TP et nous terminerons par l'utilitaire `make` qui permet l'automatisation d'un grand nombre de tâches lors de la réalisation d'un projet.

1.2 Outils de développement

1.2.1 Gestion des bibliothèques

Une bibliothèque est un fichier unique regroupant une collection de fichiers objets dans une structure qui rend possible de retrouver les fichiers originaux (appelés *membre* d'une archive). Une bibliothèque simplifie la programmation et l'édition de liens en regroupant ensemble, par exemple, des sous-programmes d'intérêt général ou des sous-programmes spécifiques à un projet. A chaque bibliothèque correspond un fichier d'en-tête dans lequel se trouve l'interface de toutes les fonctions utilisables ainsi que certaines définitions de types et de constantes.
Il existe deux types de bibliothèques :

- Bibliothèque statique : Le contenu de la bibliothèque est intégré à l'exécutable lors de l'édition de liens.
- Bibliothèque dynamique : Les fonctions de la bibliothèques sont chargées lors de l'exécution du programme.

L'utilisation d'une bibliothèque statique lors de la compilation permet l'obtention d'un exécutable qui s'exécutera partout. Il ne sera pas nécessaire pour l'utilisateur d'installer en plus des bibliothèques pour faire fonctionner le programme. Néanmoins, la taille de l'exécutable ainsi créé est nettement plus importante et consomme plus de ressources lors de son exécution. De plus le changement d'une bibliothèque dynamique (changement de l'implémentation pas de l'interface) est transparent, cela ne nécessite pas la recompilation de l'exécutable.

1.2.1.1 Conventions

Par conventions, les librairies sont installées dans le répertoire */usr/lib/* et le répertoire */lib/*. C'est par défaut à cet endroit que le compilateur va chercher les bibliothèques à utiliser. Le nom d'une bibliothèque commence toujours par *lib* et à pour extension (dans le cas d'une bibliothèque statique) *.a* comme par exemple `libpthread.a` (bibliothèque pour la programmation multithreadée).

1.2.1.2 Recherche de fonctions : l'utilitaire nm

Il est souvent intéressant de savoir si telle ou telle fonction fait partie d'une certaine bibliothèque, ne serait-ce que pour indiquer au compilateur quelles sont les bibliothèques à utiliser. Pour cela, nous utilisons l'utilitaire `nm`. Il liste les différents symboles présents dans une biliothèque. Dans l'exemple suivant, nous cherchons à vérifier que la fonction `pthread_create()` est bien incluse dans la librairie *libpthread.a*.

L'option **-P** permet de formater la sortie selon la norme POSIX. Le résultat est affiché sur trois colonnes :

- La première colonne indique le nom du symbole
- La deuxième colonne indique l'état du symbole
  - **U** : Symbole non défini
  - **T** : Symbole défini normalement
- La troisième colonne indique l'adresse à laquelle se trouve la fonction dans la bibliothèque

1.2.1.3 Création de bibliothèques : l'utilitaire ar

Nous allons maintenant développer nos propres librairies, pour cela nous utilisons `ar`. Cet utilitaire permet la création, la modification et l'extraction de bibliothèques.
Dans l'exemple suivant, nous créons une bibliothèque nommée `libsck` à partir des fichiers `aff.o` et `lib_socket.o`.

Les options utilisées sont les suivantes :

- **c** : Création de l'archive
- **r** : Insertion des fichiers dans la bibliothèque avec remplacement si les symboles sont déjà existants
- **u** : Insertion uniquement si les fichiers sont plus récents
- **s** : Création d'un index (utilisé par `nm`)

1.2.1.4 Création de bibliothèque "dynamique"

Pour générer un objet partagé (*Shared Object*), appelé également bibliothèque dynamique, il faut :

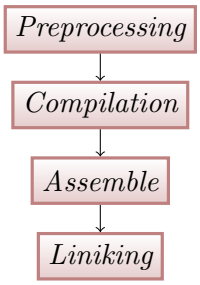
1. création d'un code objet
2. création de la bibliothèque

```
gcc -Wall -fPIC -c *.c
gcc -shared -Wl,-soname,libctest.so.1 -o libctest.so.1.0 *.o
mv libctest.so.1.0 /usr/lib
ln -sf /usr/lib/libctest.so.1.0 /usr/lib/libctest.so.1
ln -sf /usr/lib/libctest.so.1.0 /usr/lib/libctest.so
```

1.2.2 Compilateur C : gcc

Pour la programmation en C, nous utilisons le compilateur **gcc** (**G**nu **C** **C**ompiler). Il enchaîne automatiquement l'appel des différents outils réalisant la traduction d'un fichier source en un exécutable. Pour rappel, ces étapes sont :

— La précompilation (source $->$ source étendu)
— La compilation (source étendu $->$ assembleur)
— L'assemblage (assembleur $->$ binaire)
— L'édition de liens (binaire + fonctions de la bibliothèque $->$ exécutable)



La syntaxe est :

gcc [options] source.c

Les options peuvent être classées dans différentes catégories :

— Option de gcc :

option	usage
-E	Arrêt après la précompilation
-S	Arrêt après la compilation
-c	Arrêt après l'assemblage
-M	Permet d'obtenir la liste des dépendances

  — **-pipe** : Enchaînement des étapes sans fichiers temporaires. Cela accélère le fonctionnement mais nécessite plus de mémoire ;
  — **On** : Niveau d'optimisation ($0 -> 3$) ;
  — **-Wall** : Affiche tous les warnings ;
  — **-o nom** : permet de spécifier le nom du fichier de sortie (par défaut `a.out`) ;
  — **-g** : Ajout de symbole pour le débuggage (utilisation de l'utilitaire `gdb`) ;

— Option du préprocesseur :
  — **-DMACRO** : Définition de la macro-instruction **MACRO** (semblable à `#define MACRO`) ;
  — **-DMACRO=valeur** : Définition de la macro-instruction **MACRO** avec la valeur **valeur** (semblable à `#define MACRO valeur`) ;
  — **-Irepertoire** : Répertoire où sont placés des fichiers d'en-têtes (par défaut */usr/include*) ;

— Option de l'éditeur de liens :
  — **-Lrepertoire** : Répertoire où sont placées des bibliothèques ;
  — **-lbibliotheque** : Inclusion de la bibliothèque. Par exemple `-lpthread` pour la bibliothèque `libpthread` ;

1.2.3 Utilitaire de compilation : make

Le but de l'utilitaire `make` est de déterminer automatiquement quelles sont les parties d'un programme qui doivent être recompilées et réaliser cette compilation en fonction de règles situées dans un fichier de dépendances.

1.2.3.1 La commande

La commande est la suivante :

```
make [-f fichier_make] [options] [ref]
```

Les éléments optionnels sont :
— fichier_make : Référence du fichier de dépendance (par défaut les règles sont placés dans un fichier nommé `Makefile` ou `makefile`) ;
— ref : Règle à utiliser (par défaut, `make` utilise la première règle du fichier de dépendances).

1.2.3.2 Le fichier de dépendances

Le fichier de dépendance permet de spécifier à `make` les actions à effectuer. Voici deux exemples de fichiers de dépendances. Le premier est un fichier simple, le second est un peu plus évolué.

```
1  #Exemple simple de fichier Makefile
2
3  #
4  # Constantes liees au projet
5  #
6
7  OBJS = aff.o lib_socket.o
8
9
10 #
11 # La cible generale
12 #
13
14 all: $(OBJS)
15         ar crus libsck.a $(OBJS)
16
17 aff.o: aff.c
18         gcc -c aff.c
19
20 lib_socket.o: lib_socket.c lib_socket.h
21         gcc -c lib_socket.c
```

Le deuxième fichier comporte un plus grand nombre de constantes et utilise la commande `makedepend` pour calculer automatiquement les dépendances entre les différents fichiers.

```
1   # Exemple plus élaboré de fichier Makefile
2
3   #
4   # Constantes liees au projet
5   #
6
7   SHELL = /bin/sh
8   CC      = gcc                 # Compilateur
9   CDEBUG = -g -DDEBUG       # Options de déverminage
10  CFLAGS += -pipe -Wall     # Options du compilateur
11  RM = rm -f -v             # Constante de nettoyage
12  AR = ar crus              # Constante de création de bibliothèque
13  MKDEP = makedepend        # Programme de calcul de dépendances
14
15  SRC = aff.c lib_socket.c # Fichiers sources utilisés
16  OBJS = $(SRC:.c=.o)       # Fichiers objets utilisés
17  LIBNAME = libsck.a        # Nom de la bibliothèque générée
18
19
20  #
21  # La cible generale
22  #
23
24  all: depend $(OBJS)
25          @echo
26          @echo Début de la création de la bibliothèque libsck.a
27          $(AR) $(LIBNAME) $(OBJS)
28          @echo Fin de la création de la bibliothèque libsck.a
29          @echo
30
31  #
32  # La cible de vérification des dépendances
33  #
34
35  depend:
36          @echo
37          @echo Début du calcul des dépendances
38          $(MKDEP) $(SRC)
39          @echo Fin du calcul des dépendances
40          @echo
41
42
43  #
44  # La cible de déverminage
45  #
```

```
46
47  debug:
48          @echo
49          @echo Déverminage
50          $(MAKE) CFLAGS="$(CFLAGS) $(CDEBUG)"
51          @echo
52
53
54  #
55  # La cible de nettoyage
56  #
57
58  clean:
59          @echo
60          @echo Début du nettoyage
61          $(RM) *.o
62          @echo Fin du nettoyage
63          @echo
```

La première partie de ces fichiers de dépendances est constituée de déclarations de constantes dont la syntaxe est <identificateur>=<chaîne>. Lors de l'exécution de `make`, $<identificateur> est remplacé par <chaîne> (semblable aux variables manipulées par le shell).
Viennent ensuite les différentes relations de dépendance dont la syntaxe est la suivante :

```
cible: [ref]
    [Actions] (la ligne commence après une tabulation)
```

`ref` peut être une référence à une autre cible ou une référence à un fichier. De plus `ref` n'est pas obligatoire si la cible ne dépend de personne (comme par exemple la cible `clean`).
Tout ce qui suit le caractère `#` est considéré par `make` comme un commentaire.
`make` dispose également de variables internes que nous ne sommes pas obligés de redéfinir, comme par exemple `$(MAKE)` qui représente l'exécutable `make` lui-même. Ainsi, la ligne `$(MAKE) CFLAGS=" $(CFLAGS) $(CDEBUG) "` signifie que nous faisons appel à `make` en lui passant comme paramètre la macro-définition `CFLAGS` initialisée avec l'ancienne valeur de `CFLAGS` et avec la valeur de `CDEBUG`.

Chapitre 2

Processus

2.1 Présentation

Dans ce chapitre, nous allons faire un rapide rappel théorique sur les processus puis nous verrons quelques fonctions permettant la manipulation de ces processus.

2.1.1 Définition

On appelle processus (ou tâche) toute exécution d'un programme à un instant donné.
Un processus est caractérisé par :

- Une identification, le PID (**P**rocess **Id**entification) qui est un nombre entier unique ;
- Un processus père, le PPID ;
- Un propriétaire ;
- Un groupe propriètaire ;
- Un état.

2.1.2 Cycle de vie d'un processus

Le cycle de vie d'un processus est représenté à la figure 2.1.

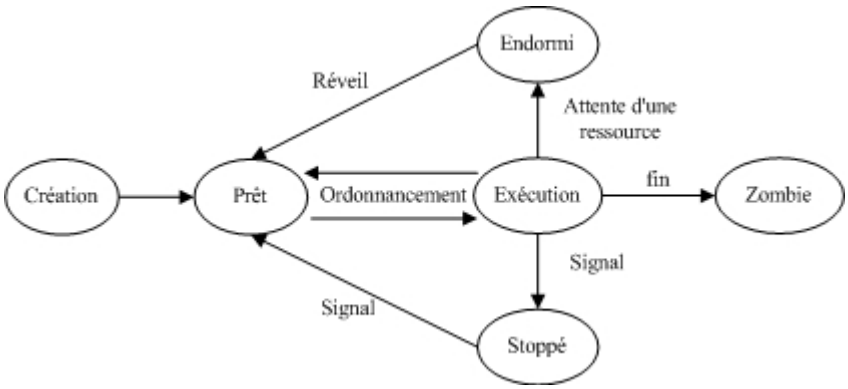


FIGURE 2.1 – Cycle de vie d'un processus

2.2 Création et terminaison d'un processus

2.2.1 Création

Il est possible de créer dynamiquement un nouveau processus grâce à la fonction `fork()` décrite ci-dessous. Le processus qui exécute cette fonction est le père du nouveau processus. Le processus créé est une copie exacte du père et les deux processus s'exécutent de manière concurrente.

```
#include <unistd.h>

pid_t fork(void);
```

Le fils hérite de son père :

- Du même code ;
- D'une copie de la zone de données ;
- D'une copie de la zone de pile ;
- De son environnement ;
- De son propriétaire ;
- Des descripteurs de fichiers ouverts ;
- Des traitements des signaux.

La distinction entre le père et le fils est faite grâce à la valeur de retour du `fork`. Si cette valeur est égale à 0, nous sommes dans le fils, sinon la valeur correspond au PID du fils et nous sommes dans le père.
Le petit exemple suivant montre l'utilisation de `fork()` :

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <sys/types.h>
 4 #include <sys/wait.h>
 5
 6 int main(void)
 7 {
 8     pid_t pid;
 9
10     pid = fork();
11     printf("Nous sommes 2\n");
12     fork();
13     printf("Nous sommes 4\n");
14
15     return 0;
16 }
```

2.2.2 Suspension

Il est possible de suspendre un processus pendant un certain nombre de secondes grâce à la fonction décrite ci-dessous.

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Cette fonction prend en argument le nombre de seconde pendant lesquelles le processus passe dans l'état *endormi*.

2.2.3 Terminaison

Le processus peut s'arrêter de manière naturel après un `return` à la fin de la fonction `main`, mais il peut également provoquer sa fin explicitement en faisant appel à la fonction `exit` décrite ci-dessous.

```
#include <unistd.h>

void exit(int status);
```

Cette fonction prend comme argument la valeur de retour du processus. Cette valeur, comprise entre 0 et 255, est par convention égale à 0 lorsque le processus se termine correctement.

2.2.4 Synchronisation avec les processus fils

Un processus père peut attendre la fin de ces fils grâce aux deux fonctions *wait* et *waitpid* décrites ci-dessous :

```
#include <sys/types.h>
#include <sys/wait.h>

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

Les fonctions renvoient le `PID` du fils. La variable `status` contient la valeur de retour du fils, cela permet au père de savoir si son fils a fonctionné correctement.
La fonction `wait` est bloquante jusqu'à la terminaison de n'importe lequel des fils. Si un fils est déjà mort, cette fonction renvoie immédiatement le résultat.
La fonction `waitpid` attend la fin d'un fils dont le `PID` est passé en argument. Si cet valeur est égal à -1, cette fonction aura le même comportement que la fonction `wait`. L'option peut être initialisé en utilisant deux constantes qui sont :

- **WNOHANG** : cela provoque le retour immédiat de la fonction si aucun fils ne s'est déjà terminé. La valeur de retour est égale à 0.
- **WUNTRACED** : cela provoque le retour pour des fils dont l'état change.

2.3 Exemple

Ce petit exemple montre l'utilisation de `fork` pour détacher automatiquement un processus du terminal.

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <sys/types.h>
 4
 5 int main(void)
 6 {
 7     pid_t pid;
 8     pid = fork();
 9
10     /* Vérification de la bonne exécution du fork() */
11     if(pid == -1)
12     {
13         perror("Erreur lors du fork ");
14         exit(-1);
15     }
16
17     /* Si la valeur de retour est différente de 0  */
18     /* Nous sommes dans le père                    */
19     /* Comme nous voulons fonctionner en tant que  */
20     /* démon et nous détacher du terminal, nous    */
21     /* provoquons la fin du père. Le fils continue */
22     /* de fonctionner normalement                  */
23     if(pid != 0)
24         exit(0);
25
26     /* A partir de maintenant, tout le code est */
27     /* exécuté par le fils                      */
28     printf("Je suis le fils mon pid est %d\n", getpid());
29                     .
30                     .
31                     .
32                     .
33                     .
34
35     return 0;
36 }
```

2.4 Attributs d'un processus

Les différentes fonctions présentées ici servent à récupérer des informations sur un processus. L'utilisation est suffisamment explicite, nous ne nous attarderons donc pas à décrire en détail ces fonctions.

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

pid_t   getpid(void);    /* Récupération le PID du processus courant         */
pid_t   getppid(void);   /* Récupération du PID du père du processus courant */

uid_t   getuid(void);    /* Récupération de l'utilisateur réel     */
uid_t   geteuid(void);   /* Récupération de l'utilisateur effectif */

gid_t   getgid(void);    /* Récupération du groupe réel     */
gid_t   getegid(void);   /* Récupération du groupe effectif */
```

2.5 Modification des attributs

Les différentes fonctions présentées ici permettent de modifier les informations relatives à un processus.

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

int   setuid(uid_t uid);                  /* Changement de l'utilisateur effectif */
int   setreuid(uid_t ruid, uid_t euid);   /* Changement de l'utilisateur réel
                                                                  et effectif */
int   seteuid(uid_t euid);                /* Changement de l'utilisateur effectif */

int   setgid(gid_t gid);                  /* Changement du groupe effectif         */
int   setregid(gid_t rgid, gid_t egid);   /* Changement du groupe réel et effectif */
int   setegid(gid_t egid);                /* Changement du groupe effectif         */
```

La valeur de retour est égale à 0 en cas de succès et à -1 en cas d'erreur.

2.6 Exécution de programmes

Les différentes fonctions présentées ici permettent l'exécution d'un nouveau programme. Ce nouveau programme remplace le code du programme appelant.

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ... , const char *argn, NULL);
int execlp(const char *file, const char *arg0, ... , const char *argn, NULL);

int execle(const char *path, const char *argv0, ... ,const char *argn, NULL,
              char *const envp[]);
```

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

Après l'appel à une fonction `exec*`, le processus conserve toutes ses caractéristiques d'origine (PID, PPID, propriétaire, ...).
L'argument `path` correspond au nom de l'exécutable à lancer. Les arguments `arg0` à `argn` correspondent aux différents paramètres passés à la commande. Ces éléments sont ceux que nous récupérons dans la fonction `main` avec le tableau `*argv`, cela signifie que `arg0` correspond au nom de l'exécutable lancé. La liste des arguments se termine par `NULL`.
La variable `envp` permet de passer des variables d'environnement pour l'exécutable.
Lors de l'exécution de `execl` et `execv`, si la variable `path` est un nom relatif, la recherche de l'exécutable aura lieu uniquement dans le répertoire de travail.
Lors de l'exécution de `execlp` et `execvp`, si la variable `path` est un nom relatif, la recherche de l'exécutable aura lieu dans le répertoire de travail et dans les répertoires spécifiés dans la variable d'environnement `PATH`.

2.6.1 Exemples d'utilisation

Nous présentons ci-dessous deux exemples d'utilisation de la commande `execl`.
Le premier exemple permet de constater que l'appel à une commande de type `exec*` réalise bien le remplacement complet du code du programme appelant.

```
 1 #include <stdio.h>
 2 #include <errno.h>
 3 #include <unistd.h>
 4
 5 int main(void)
 6 {
 7     if(execl("/bin/ls", "ls", "-l", NULL) == -1)
 8     {
 9         perror("Echec execl\n");
10         exit(1);
11     }
12     else
13         printf("Cette ligne n'est jamais affichée\n");
14
15     exit(0);
16 }
```

Le Deuxième exemple montre l'utilisation de `execl` dans un fils. Cela peut vaguement ressembler à un shell primitif (lancement d'une application dans un autre processus puis retour au processus d'origine).

```
1 #include <stdio.h>
2 #include <errno.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
```

```
 6
 7 int main(void)
 8 {
 9     int etat;
10     pid_t pid,n;
11
12     if((n=fork()) == -1)
13     {
14         perror("Erreur du fork\n");
15         exit(1);
16     }
17
18     if(n == 0)  /* Fils */
19     {
20         if(execl("/bin/date", "date", NULL) == -1)
21         {
22             perror("Erreur du execl\n");
23             exit(2);
24         }
25     }
26
27     /* On attend la fin du fils */
28     if((pid = waitpid(n,&etat,NULL)) == -1)
29     {
30         perror("Erreur waitpid\n");
31         exit(3);
32     }
33
34     /* Vérification de la valeur de retour du fils */
35     if(WIFEXITED(etat) != 0)
36     {
37         printf("Arrêt normal du fils\n");
38     }
39
40     exit(0);
41
42 }
```

Chapitre 3

IPC System V

3.1 Présentation

Dans ce chapitre, nous allons étudier différents moyens permettant à des processus de communiquer entre eux. Ces fonctions fondamentales ont pour nom IPC (**I***nter* **P***rocess* **C***ommunication*).

3.1.1 Types de communication

Les IPC définissent trois types de communications :

- **Files de messages** : Une file de message est assimilable à une boîte aux lettres. Les processus peuvent y déposer ou récupérer des messages (entier, chaîne de caractères, ...) ;
- **Sémaphores** : Les sémaphores permettent la synchronisation de processus en mettant des verrous sur des ressources partagées ;
- **Mémoire partagée** : Mise en commun d'un espace mémoire entre plusieurs applications.

Nous ne verrons que les files de messages, pour les sémaphores et la mémoire partagée, se référer à [**?**].

3.2 Eléments de base

3.2.1 Structure de base

La structure `struct ipc_perm` représente un objet de type IPC dans le système. Cette structure permet le contrôle et l'accès à l'objet représenté.

```
#include <sys/ipc.h>

Struct ipc_perm {
      ushort   uid;      /* Propriétaire                      */
      ushort   gid;      /* Groupe propriétaire               */
      ushort   cuid;     /* Créateur                          */
      ushort   cgid;     /* Groupe créateur                   */
      ushort   mode;     /* Droits d'accès                    */
      ushort   seq;      /* Nombre d'utilisation de l'entrée */
```

```
      key_t    key;      /* Clé                                  */
};
```

3.2.2 Commandes shell

Il existe deux commandes shell pour gérer les IPC.

3.2.2.1 La commande `ipcs`

la commande `ipcs` permet de lister l'ensemble des objets IPC existant. Son exécution donne le résultat suivant :
Les objets sont classés par catégories (mémoire partagée, sémaphores, files de messages). Dans chaque catégorie, nous trouvons la clé, l'identification, le propriétaire et les droits sur l'objet (comme pour les fichiers).

3.2.2.2 La commande `ipcrm`

La commande `ipcrm` permet de supprimer un objet IPC existant, comme dans l'exemple ci-dessous.
Cette commande prend en premier argument le type d'objet (`shm, msg, sem`) et en deuxième argument l'identification de l'objet.

3.2.3 Création d'une clé

La fonction `ftok`, décrite ci-dessous, permet de générer une clé à partir d'un nom de fichier et d'un caractère.

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(char *pathname, char proj);
```

L'utilisation de cette fonction n'est toutefois pas obligatoire pour initialiser une clé. Nous pouvons déclarer une variable de type `key_t` et l'initialiser avec un `int` quelconque.

3.3 Les files de messages

Dans cette section, nous allons étudier l'ensemble des primitives nous permettant l'utilisation des files de messages.

3.3.1 structure de base

La structure `struct msgbuf` représente le message de base géré par une file de messages.

```
#include <sys/msg.h>

struct msgbuf {
      mtyp_t   m_type;        /* type de message (entier positif) */
```

```
    char    mtext[1];      /* contenu du message              */
};
```

Nous pouvons tout de suite voir que cette structure n'est pas toujours utilisable directement car le taille pour le message est trop restreinte. La solution consiste donc à redéfinir une structure comportant un `long` pour représenter le type et un tableau de caractères de taille adaptés aux besoins.

3.3.2 Création d'une file de message

La fonction `msgget`, décrite ci-dessous, permet soit de créer une file de messages soit de retrouver l'identification d'une file existante.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t clef, int option);
```

Le premier argument correspond à la clé de la file de messages existante ou que nous voulons créer. Les options vont nous permettre de choisir l'action à réaliser.
Si nous plaçons l'option `IPC_CREAT` et que la clé n'est pas encore utilisée alors la file de messages sera créée et son identification serra retournée par fonction.
Si nous plaçons l'option `IPC_CREAT` ou `IPC_EXCL` et que la clé est déjà utilisée alors la fonction renvoie l'identification de la file de messages.
Généralement, nous pouvons utiliser `msgget(clé, IPC_EXCL)` pour obtenir l'identification d'une file existante et `msgget(clé, IPC_CREAT|IPC_EXCL|0666)` pour créer une nouvelle file de messages. `0666` permet de spécifier les droits d'accès de la file de la même manière que pour un fichier normal.
Si la fonction échoue, la valeur de retour est égale à -1.

3.3.3 Contrôle d'une file de message

La fonction `msgctl`, décrite ci-dessous, permet de contrôler une file de messages.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Le premier argument correspond à l'identification de la file de messages que l'on souhaite contrôler. Le second argument indique le type d'opération que nous souhaitons réaliser :

- `IPC_RMID` : Pour détruire la file de messages (`buf` est `NULL`) ;
- `IPC_SET` ; Pour fixer certains membres de la structure `struct msqid_ds` qui caractérise la file de messages ;
- `MSG_STAT` ; Pour copier la table associée à la file de message dans `buf`.

3.3.4 Envoi de messages

La fonction `msgsnd`, décrite ci-dessous, permet l'envoi de messages.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, struct msgbuf *msgp, int msgtaille, int msgopt);
```

Elle prend comme arguments :
- L'identification de la file de messages ;
- Un pointeur représentant le message à envoyer ;
- La taille du message envoyé ;
- Des options : Si `IPC_NOWAIT` est passé comme option, l'appel de la fonction ne sera plus bloquant. Cela a un intérêt uniquement lorsque la file de messages est pleine.

3.3.5 réception de messages

La fonction `msgrcv`, décrite ci-dessous permet la réception d'un message.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv(int msqid, struct msgbuf *msgp, int lgmax, long type, int option);
```

Elle prend comme arguments :
- L'identification de la file de messages
- Un pointeur pour stocker le message
- La taille maximale du message à recevoir
- Le type du message
- Des options : Si `IPC_NOWAIT` est passé comme option, l'appel de la fonction ne sera plus bloquant. Cela évite l'attente active.

3.4 Exemple

Pour montrer l'utilisation des différentes fonctions présentées précédemment, voici un petit exemple de communication entre un processus père et un processus fils. Le premier fichier, nommé *communication_ipc.h* contient la définition des constantes et des types utilisés. Le deuxième fichier, nommé *ipc.c* contient le programme.
Fichier *communication_ipc.h*

```
1 #ifndef __COMMUNICATION_IPC_H
2 #define __COMMUNICATION_IPC_H
3
4 /* Définition des fichiers d'inclusions */
5 #include <sys/types.h>
```

```
 6 #include <sys/ipc.h>
 7 #include <sys/msg.h>
 8
 9 /* Constantes utilisées pour obtenir les clés */
10 #define FICHIER_COMMANDE "/etc/passwd"
11 #define PROJ_FTOK 'a'
12
13 /* Définition des types de messages */
14 #define MESSAGE                 1
15 #define RECU                    2
16 #define FIN                     3
17
18 #define SIZE_BUF 256
19
20 /* Structure d'un message */
21 struct data_s {
22     long type;      /* Type du message */
23     char buf[SIZE_BUF]; /* Message */
24 };
25
26 typedef struct data_s data_t;
27
28 #endif
```

Fichier *ipc.c*

```
 1 /*********************************************************************/
 2 /* Envoi de messages entre deux processus via une boite aux lettre */
 3 /* Le père envoie un message de type MESSAGE au fils et attend un  */
 4 /* message de type RECU, affiche son contenu et attend la mort de  */
 5 /* son fils et détruit la file de message                          */
 6 /* Le fils attend le message de type MESSAGE, affiche son contenu, */
 7 /* envoie un message de type RECU et se termine                    */
 8 /*********************************************************************/
 9
10 #include <stdio.h>
11 #include <sys/types.h>
12 #include <sys/wait.h>
13 #include <sys/ipc.h>
14 #include <sys/msg.h>
15 #include <errno.h>
16 #include <unistd.h>
17
18 #include "./communication_ipc.h"
19
20
21 int main(void)
22 {
23     key_t cle;
```

```
24      int pid;
25      int identificateur;
26      data_t message;
27
28      /* Création de la clé */
29      if((cle = ftok(FICHIER_COMMANDE, PROJ_FTOK)) == -1)
30      {
31          perror("Erreur création clé\n");
32          exit(1);
33      }
34
35      /* Création de la file de messages */
36      if((identificateur = msgget(cle, IPC_CREAT|0666)) == -1)
37      {
38          perror("Erreur création file de messages\n");
39          exit(2);
40      }
41
42      /* Création d'un processus fils */
43      if((pid = fork()) == -1)
44      {
45          perror("Erreur création fils\n");
46          exit(3);
47      }
48
49      if(pid == 0)
50      {
51              /* Nous sommes dans le fils */
52
53          /* Lecture du message du père */
54          msgrcv(identificateur, (struct msgbuf *) &message,
sizeof(data_t),
55                  MESSAGE, 0);
56
57          /* Affichage du message */
58          printf("message reçu par le fils : %s", message.buf);
59
60          /* Préparation du message à envoyer */
61          message.type = RECU;
62          strcpy(message.buf, "message fils\n");
63
64          /* Envoi du message au père */
65          msgsnd(identificateur, (struct msgbuf *) &message,
sizeof(data_t),
66                  0);
67          /* Fin du fils */
68          exit(0);
69
```

```
70      }
71      else
72      {
73          /* Nous sommes dans le père */
74
75          /* Préparation du message à envoyer */
76          message.type = MESSAGE;
77          strcpy(message.buf, "message pere\n");
78
79          /* Envoi du message au père */
80          msgsnd(identificateur, (struct msgbuf *) &message,
sizeof(data_t),
81                  0);
82          /* Lecture du message du fils */
83          msgrcv(identificateur, (struct msgbuf *) &message,
sizeof(data_t),
84                  RECU, 0);
85
86          /* Affichage du message */
87          printf("message reçu par le pére : %s", message.buf);
88
89          /* Attente de la mort du fils */
90          wait(NULL);
91
92          /* Destruction de la file de message */
93          msgctl(identificateur, IPC_RMID, 0);
94
95          /* Fin du père */
96          exit(0);
97      }
98 }
```

Chapitre 4

Programmation multithreadée

4.1 Présentation

Les threads, parfois appelés *processus léger*, sont les cousins des processus Unix standards. Un thread s'exécute au sein d'un processus. Il existe un partage des ressources entre les threads d'un même processus, comme par exemple les fichiers ouverts.

4.1.1 Cycle de vie d'un thread

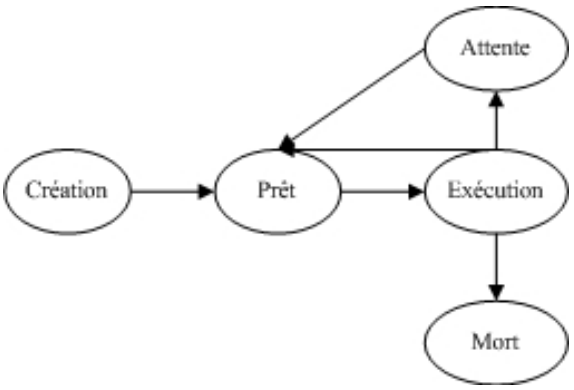


FIGURE 4.1 – Cycle de vie d'un thread

4.1.2 Contraintes de programmation

l'ensemble des threads d'un processus partagent les mêmes variables globales. Il faut donc gérer l'accès concurrent à ces ressources pour éviter par exemple les incohérences. De plus il n'existe pas de mécanisme de protection entre les threads. Un thread peut effacer des données manipulées par un autre thread.

4.1.3 Utilisation des threads

L'utilisation des threads se retrouve dans la plupart des logiciels informatique :

— Dans les systèmes d'exploitation : prise en charge des appels systèmes (comme pour Solaris de Sun) ;

- Dans les programmes serveurs : Diminue le temps de réponse en autorisant le traitement de plusieurs requêtes simultanées (surtout sur les machine multi-processeurs) ;
- Dans les applications :
  - Des les interfaces utilisateur : Chaque élément graphique est associé à un thread pour plus de réactivité ;
  - Dans le cadre de la simulation : Observer le comportement collectif d'activités complexes ;
  - Dans les jeux : Faire évoluer plusieurs entités en parallèle et augmentation de la réactivité.

4.1.4 Généralités sur les threads Posix (*pthread*)

4.1.4.1 Noms de fonctions

Les noms de fonctions respectent la norme suivante :

`pthread[_objet]_operation[_np]`

dans laquelle
- `objet` représente le type de l'objet auquel la fonction s'applique (*mutex* pour les sémaphores d'exclusion mutuelle, *cond* pour les variables de conditions).
- `operation` représente l'opération à réaliser (*create* pour la création, ...)
- `np` indique que la fonction n'est pas portable (ajout à la norme)

4.1.4.2 Noms de type

Les noms de types utilisés respectent la norme suivante :

`pthread[_objet]_t`

dans laquelle `objet` a la même signification que précédemment.

4.1.4.3 Identification d'un thread

Pour obtenir son **TID** (**T**hread **Id**entification), un thread peut appeler la fonction `pthread_self` décrite ci-dessous.

```
#include <pthread.h>

pthred_t pthread_self(void);
```

Cette fonction renvoi l'identificateur du thread appelant.

4.2 Création et terminaison des threads

4.2.1 Création

La création d'un thread se fait par l'appel à la fonction `pthread_create` décrite ci-dessous. La valeur de retour est égale à 0 si il n'y a pas eu de problème, -1 sinon.

```
#include <pthread.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*start_routine)(void *),void *arg);
```

Les arguments utilisés sont les suivants :
- `thread` va contenir l'identité du thread créé ;
- `attr` définit les attributs du threads (voir section suivante 4.3) ;
- `start_routine` est la fonction exécutée par le thread ;
- `arg` correspond à l'argument de la fonction `start_routine`.

4.2.2 Terminaison

De même que pour un processus, un thread peut demander explicitement sa fin en faisant appel à la fonction `pthread_exit` décrite ci-dessous.

```
#include <pthread.h>

void pthread_exit(int *status);
```

L'argument `status` est la valeur de retour du thread.

4.2.3 Exemple

Voici un *Hello World !* multithreadé. Ce programme très simple nous permet de voir le fonctionnement des fonctions `pthread_create` et `pthread_exit`.

```
 1 #include <stdio.h>
 2 #include <pthread.h>
 3
 4 void print_message_function(void *ptr)
 5 {
 6     char *message;
 7     message = (char *) ptr;
 8     printf("%s", message);
 9     pthread_exit(NULL);
10 }
11
12 int main(void)
13 {
14     pthread_t thread1, thread2;
15     char *message1 = "Hello ";
16     char *message2 = "World\n";
17     pthread_attr_t attr1, attr2;
18     int tid1, tid2;
19
20     /* Création des attributs des threads */
21     /* Ici, initialisation par défaut     */
```

```
22      pthread_attr_init(&attr1);
23      pthread_attr_init(&attr2);
24
25      /* Création des threads */
26      tid1 = pthread_create(&thread1, &attr1,
27              (void *) &print_message_function, (void *) message1);
28
29      tid2 = pthread_create(&thread2, &attr2,
30              (void *) &print_message_function, (void *) message2);
31
32      /* Synchronisation du processus */
33      pthread_join(thread1,NULL);
34      pthread_join(thread2,NULL);
35
36      return 0;
37  }
```

La fonction `pthread_join` sera vue dans la section consacrée aux synchronisations.

4.3 Attributs d'un thread

Les attributs d'un threads sont gérés via une variable de type `pthread_attr_t`. Les attributs sont les suivants :

- **detachstate** : Contrôle si le thread est dans un état joignable (par défaut) ou non joignable. Dans l'état joignable il est possible de synchroniser des threads via `pthread_join` ;
- **schedpolicy** : Change la politique et les paramètres d'ordonnancement du thread ;
- **schedparam** : Contient la politique d'ordonnancement ;

4.3.1 Initialisation

La structure d'attribut est initialisée par la fonction `pthread_attr_init` qui la remplit avec les valeurs par défauts.

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
```

Cette fonction renvoie 0 en cas de succès.

4.3.2 Lecture

La lecture des attributs d'un thread se fait via une des fonctions suivantes selon l'attribut qui nous intéresse :

```
#include <pthread.h>

int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
```

```
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);
int pthread_attr_getschedparam(const pthread_attr_t *attr, struct sched_param *param);
```

Toutes ces fonctions renvoient 0 en cas de succès.

4.3.3 Modification

La modification des attributs d'un thread se fait via une des fonctions suivantes selon l'attribut qui nous intéresse :

```
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr, int *detachstate);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int *policy);
int pthread_attr_setschedparam(pthread_attr_t *attr, struct sched_param *param);
```

Toutes ces fonctions renvoient 0 en cas de succès.

4.4 Synchronisation des threads

Il existe trois méthodes pour synchroniser des threads :
- Les rendez-vous ;
- Les sémaphores d'exclusion mutuelle ;
- Les conditions.

Nous allons ici nous intéresser uniquement aux deux premières méthodes. Pour la troisième se reporter à [**?**].

4.4.1 Les Rendez-vous

La fonction `pthread_join` décrite ci-dessous, suspend l'exécution du thread appelant jusqu'à la terminaison du thread dont nous avons passé le TID en paramètre.

```
#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return);
```

4.4.2 Les sémaphores d'exclusion mutuelle

Un sémaphore d'exclusion mutuelle (ou `mutex` MUTual Exclusion device) permet de protéger des données en posant des verrous.

4.4.2.1 Initialisation

La première étape pour utiliser un mutex est de l'initialiser. Pour cela, il existe la commande `pthread_mutex_init` décrite ci-dessous.

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *mutexattr);
```

Il existe trois types de mutex :

- **PTHREAD_MUTEX_INÏTIALIZER** : pour les mutex de types " rapide " ;
- **PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP** : pour les mutex récursifs ;
- **PTHREAD_ERREURCHECK_MUTEX_INITIALIZER_NP** : pour les mutex à vérification d'erreur.

4.4.2.2 Acquisition d'un mutex (opération *P*)

Une fois notre mutex initialisé, nous pouvons poser un verrou (acquérir le mutex) en utilisant la fonction `pthread_mutex_lock` ou `pthread_mutex_unlock`.
La fonction `pthread_mutex_lock` verrouille un mutex. Si il est déjà en la possesion d'un autre thread, la fonction suspend le thread jusqu'à la libération.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
```

la fonction `pthread_mutex_trylock` se comporte de manière identique à la fonction `pthread_mutex_lock` sauf si le mutex est déjà verrouillé. Dans ce cas, la fonction ne se bloque pas.

```
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

4.4.2.3 Libération d'un mutex (opération *V*)

Lorsque nous n'avons plus besoin du mutex, nous pouvons le déverrouiller en utilisant la fonction `pthread_mutex_unlock` décrite ci-dessous.

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

4.4.2.4 Destruction d'un mutex

Et finalement, lorsque plus personne n'a besoin du mutex, nous pouvons le détruire en utilisant la fonction `pthread_mutex_destroy` décrite ci-dessous.

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

4.4.2.5 Exemple

L'exemple suivant montre l'utilisation d'un mutex pour permettre à deux threads de partager une variable. Cet exemple n'est pas parfait car les valeurs de retour des différentes fonctions n'est pas testé.

```
1  /*************************************************************/
2  /* Deux threads réalisent des calculs et additionnent leurs */
3  /* résultats dans une variable commune                       */
4  /*************************************************************/
5
6  #include <pthread.h>
7  #include <stdio.h>
8
9  /* Variables Partagées */
10 double result=0;
11 pthread_mutex_t mutex;
12
13 /* Fonction réalisée par chaque thread */
14 void thread()
15 {
16     register int i,n;
17
18     for(i=0; i<10; i++)
19     {
20         n = 0;
21         n = 2*i + 3;
22
23         /* Pose du verrou sur la variable partagée */
24         pthread_mutex_lock(&mutex);
25
26         /* Modification de la variable partagée */
27         result += n;
28
29 #ifdef DEBUG
30         printf("La valeur de result est %f\n",result);
31 #endif
32         /* Retire le verrou sur la variable partagée */
33         pthread_mutex_unlock(&mutex);
34     }
35
36     pthread_exit(0);
37 }
38
39
40 /* Fonction principale */
41 int main (void)
42 {
43     pthread_attr_t attr;
```

```
44      pthread_t thread1, thread2;
45
46      /* Initialisation du mutex */
47      pthread_mutex_init(&mutex,NULL);
48
49      /* Initialisation des attributs des threads */
50      pthread_attr_init(&attr);
51
52      /* Création des threads */
53      pthread_create(&thread1, &attr, (void *) thread, NULL);
54      pthread_create(&thread2, &attr, (void *) thread, NULL);
55
56      /* Synchronisation sur la fin des threads */
57      pthread_join(thread1,NULL);
58      pthread_join(thread2,NULL);
59
60      /* Destruction du mutex */
61      pthread_mutex_destroy(&mutex);
62
63      /* Affichage du résultat */
64      printf("Le résultat est %f\n",result);
65
66      exit(0);
67  }
```

Chapitre 5

Les sockets

5.1 Présentation

L'interface de programmation réseau, nommée *socket*, est apparue dans les distribution Unix de Berkeley (BSD) vers 1982. Elle fut rapidement intégrée dans le noyau, et est devenue une des interfaces les plus utilisées pour la programmation réseau.

5.2 Famille de sockets

Cette interface de programmation a été conçu indépendemment des protocoles réseau. Chaque socket appartient à une famille. Chaque famille fait référence à un protocole réseau et à un mode de fonctionnement particulier. Quelques familles sont représentées dans le tableau 5.1.

Famille	**Protocole**
AF_UNIX	Tube nommés
AF_INET	Protocole TCP et UDP
AF_INET6	Protocole IPv6
AF_NS	Protocole Xerox
AF_LAT	Protocole DEC
AF_APPLETALK	Protocole Apple Talk
AF_IPX	Protocole Novell
AF_CCITT	Protocole X25

TABLE 5.1 – Quelques familles de socket.

Dans la suite de document, nous allons nous intéresser uniquement à la famille `AF_INET` qui représente les différents protocoles Internet (IP, TCP, UDP, ...).

5.3 Structures utilisées

Dans cette section, nous allons voir les principaux types de données que nous allons manipuler par la suite. La première structure est `struct sockaddr`. Cette structure générique représente les informations d'adresses sans distinction de famille.

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

struct sockaddr{
      unsigned short   sa_family;    /* Famille d'adresse, ie AF_xxx          */
      char             sa_data[14];  /* 14 octets pour l'adresse de protocole */
};
```

le champ `sa_family` peut prendre une grande variété de valeurs, mais dans la suite de ce document, nous allons travailler exclusivement avec la famille `AF_INET`.
La structure `struct sockaddr_in` est une " spécialisation " de la structure `struct sockaddr`.

```
struct sockaddr_in {
      short int              sin_family;    /* Famille d'adresse */
      unsigned short int     sin_port;      /* Numéro de port    */
      struct in_addr         sin_addr;      /* Adresse Internet  */
      unsigned char          sin_zero[8];   /* 8 octets vides    */
};
```

Les 8 octets vides permettent à la structure `struct sockaddr_in` d'avoir la même taille que la `struct sockaddr`, cela permet de réaliser facilement la conversion de type. **Attention**, le numéro de port et le numéro d'adresse sont codés sous forme réseau (voir section suivante).
La `struct in_addr`, présentée ci-dessous, représente l'adresse Internet d'une machine (IPv4).

```
struct in_addr{
      unsigned long   s_addr;
};
```

5.4 Représentation des valeurs

Les informations transportées sur le réseau sont codées sous une forme réseau dite *Big Endian* dans laquelle les octets de poids fort sont placés avant les octets de poids faible. Il existe quatre fonctions qui permettent la conversion des `short` et des `long`.

```
#include <netinet/in.h>

/* Host TO Network Long */
unsigned long int htonl(unsigned long int hostlong);
/* Host TO Network Short */
unsigned short int htons(unsigned short int hostshort);

/* Network TO Host Long */
unsigned long int ntohl(unsigned long int netlong);
/* Network TO Host Short */
unsigned short int ntohl(unsigned short int netshort);
```

5.5 Manipulation des adresses IP

Il existe un ensemble de fonctions permettant de manipuler les adresses IP. Nous allons ici en voir quelques unes.

5.5.1 Affichage d'une adresse

Nous savons que l'adresse IP d'une machine se trouve dans le champ `sin_addr` d'une structure `struct sockaddr_in`. Les trois fonctions suivantes permettent de réaliser le formatage de cette adresse pour un affichage plus lisible.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* Conversion ASCII vers Network */
int inet_aton(const char *cp, struct in_addr *inp);

/* Conversion Network vers ASCII */
char *inet_ntoa(struct in_addr in);
```

L'affichage d'une adresse IP devient :

```
printf((%s(, inet_ntoa(ina.sin_addr));
```

5.5.2 Conversion noms de machine vers adresse IP

Il est plus facile de se souvenir du nom d'une machine que d'une adresse IP, nous allons récupérer l'adresse IP en fonction du nom grâce à la fonction `gethostbyname()` décrite ci-dessous.

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

Cette fonction renvoie un pointeur vers une structure de type `struct hostent`, décrite ci-dessous.

```
#include <netdb.h>

struct hostent {
      char   *h_name;       /* Nom officiel de la machine      */
      char   **h_aliases;   /* Liste d'alias                   */
      int    h_addrtype;    /* Type d'adresse (eg AF_INET)     */
      int    h_length;      /* Longueur de l'adresse en octets */
      char   **h_addr_list; /* Liste d'adresses                */
};
```

Le programme suivant est un exemple d'utilisation de cette fonction.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <errno.h>
 4 #include <netdb.h>
 5 #include <sys/types.h>
 6 #include <netinet/in.h>
 7
 8 int main(int argc, char *argv[])
 9 {
10     struct hostent *h;
11
12     /* On vérifie que l'utilisateur passe */
13     /* un nom de machine en paramètre     */
14     if(argc != 2)
15     {
16         fprintf(stderr,"usage: getip address\n");
17         exit(-1);
18     }
19
20     /* On récupére une structure hostent contenant */
21     /* les différentes informations relatives au   */
22     /* nom de machine donné en paramètre           */
23     if((h=gethostbyname(argv[1])) == NULL)
24     {
25         perror("gethostbyname");
26         exit(-2);
27     }
28
29     /* On affiche le résultat                */
30     /* Le Nom de machine est le nom officiel */
31     printf("Nom de machine : %s\n", h->h_name);
32     printf("Addresse IP    : %s\n", inet_ntoa(*((struct in_addr
*)h->h_addr)
33             ));
34
35     return 0;
36 }
```

5.6 Gestion des services et protocoles

```
#include <netdb.h>

struct servent {
      char    *s_name;       /* Nom officiel du service        */
      char    **s_aliases;   /* Liste d'alias                  */
```

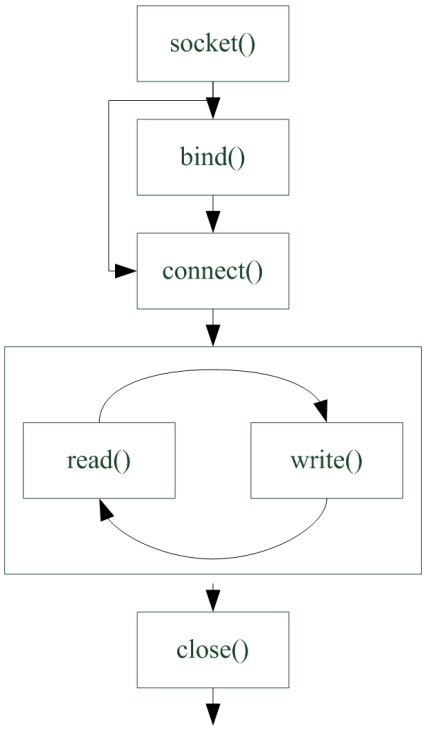


FIGURE 5.1 – Fonctionnement d'un client

```
      int    s_port;        /* Numéro de port                 */
      char   *s_proto;      /* protocole de transport utilisé */
};


#include <netdb.h>

struct protoent {
      char   *p_name;       /* Nom officiel du protocole */
      char   **p_aliases;   /* Liste d'alias             */
      int    p_proto;       /* Numéro de protocole       */
};
```

5.7 Création d'une socket

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

int socket(int domain, int type, int protocol);
```

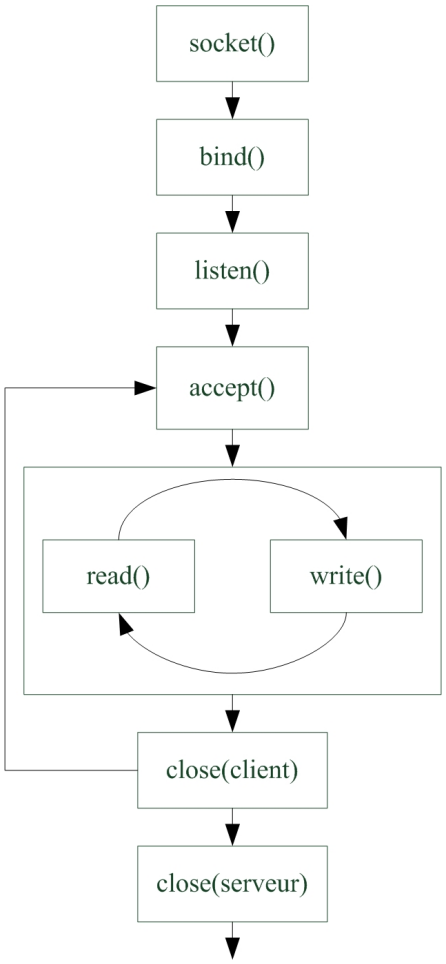


FIGURE 5.2 – Fonctionnement d'un serveur

5.8 Liaison sur un port

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

int bind(int sockfd, struct sockaddr *my_add, int addrlen);
```

5.9 Connexion

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

int connect(int sockfd, struct sockaddr *serv_add, int addrlen);
```

5.10 Ecoute des communications

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

int listen(int sockfd, int backlog);
```

5.11 Acceptation d'une connexion

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

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

5.12 Information sur la machine cliente

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

int getpeername(int sockfd, struct sockaddr *name, socklen_t *namelen);
```

5.13 Information sur la machine

```
#include <unistd.h>

int gethostname(char *name, size_t len);
int sethostname(const char *name, size_t len);
```

5.14 Communication en mode connecté

Voici un exemple de serveur TCP.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <errno.h>
 4 #include <string.h>
 5 #include <sys/types.h>
 6 #include <netinet/in.h>
 7 #include <sys/socket.h>
 8 #include <sys/wait.h>
 9 #include <unistd.h>
10 #include <arpa/inet.h>
11
12
13 #define PORT 4000   /* Port utilisé pour la connexion des clients */
```

```
14 #define BACKLOG 10  /* Nombre de connexions simultanées          */
15
16 int main(void)
17 {
18     int sockfd, new_fd;
19     struct sockaddr_in server_addr;
20     struct sockaddr_in client_addr;
21     int sin_size;
22
23     /* Création de la socket */
24     if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
25     {
26         perror("Erreur création socket\n");
27         exit(1);
28     }
29
30     server_addr.sin_family = AF_INET;
31     server_addr.sin_port = htons(PORT);
32     server_addr.sin_addr.s_addr = INADDR_ANY;
33     bzero(&(server_addr.sin_zero), 8);
34
35     if(bind(sockfd, (struct sockaddr *) &server_addr, sizeof(struct
36        sockaddr_in)) == -1)
37     {
38         perror("Erreur bind\n");
39         exit(2);
40     }
41
42     if(listen(sockfd, BACKLOG) == -1)
43     {
44         perror("Erreur listen\n");
45         exit(3);
46     }
47
48     while(1)
49     {
50         sin_size = sizeof(struct sockaddr_in);
51         if((new_fd = accept(sockfd, (struct sockaddr *) &client_addr,
52            &sin_size)) == -1)
53         {
54             perror("Erreur accept\n");
55             exit(4);
56         }
57
58         printf("Serveur : connexion de %s\n", inet_ntoa(client_addr.
59                sin_addr));
60
61         if(!fork())
```

```
62          {
63              if(send(new_fd, "Hello World !\n", 14, 0) == -1)
64                  perror("Erreur send\n");
65              close(new_fd);
66              exit(0);
67          }
68          close(new_fd);
69
70          while(waitpid(-1, NULL, WNOHANG) > 0);
71      }
72  }
```

Voici un exemple de client TCP.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <string.h>
5  #include <netdb.h>
6  #include <sys/types.h>
7  #include <netinet/in.h>
8  #include <sys/socket.h>
9  #include <unistd.h>
10
11 #define PORT 4000
12 #define MAXDATASIZE 100
13
14 int main(int argc, char *argv[])
15 {
16     int sockfd, numbytes;
17     char buf[MAXDATASIZE];
18     struct hostent *he;
19     struct sockaddr_in their_addr;
20
21     if(argc != 2)
22     {
23         fprintf(stderr, "Usage : client hostname\n");
24         exit(1);
25     }
26
27     if((he=gethostbyname(argv[1])) == NULL)
28     {
29         perror("Erreur gethostbyname\n");
30         exit(2);
31     }
32
33     if((sockfd=socket(AF_INET,SOCK_STREAM, 0)) == -1)
34     {
35         perror("Erreur création socket\n");
```

```
36          exit(3);
37      }
38
39      their_addr.sin_family = AF_INET;
40      their_addr.sin_port = htons(PORT);
41      their_addr.sin_addr = *((struct in_addr *)he->h_addr);
42      bzero(&(their_addr.sin_zero), 8);
43
44      if(connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct
45         sockaddr_in)) == -1)
46      {
47          perror("Erreur connect\n");
48          exit(4);
49      }
50
51      if((numbytes=recv(sockfd, buf, MAXDATASIZE, 0)) == -1)
52      {
53          perror("Erreur recv\n");
54          exit(5);
55      }
56
57      buf[numbytes] = '\0';
58
59      printf("Message reçu : %s\n",buf);
60
61      close(sockfd);
62      return 0;
63  }
```

5.15 Les options d'une socket

Afin de contrôler plus finement le fonctionnement d'une socket, il est possible de lui appliquer des options. Pour cela, il existe la fonction `setsockopt()`.

```
int setsockopt(int sd, int level, int optname, const void *optval, socklen_t optlen);
```

À l'inverse, il est possible de récupérer les options appliquées à une socket, en utilisant la fonction `getsockopt()`.

```
int getsockopt(int sd, int level, int optname, void *optval, socklen_t *optlent);
```

Les options d'une socket sont défnies par quatre niveau sous Linux : `SOL_SOCKET`, `SOL_IP`, `SOL_IPV6` et `SOL_TCP`

5.15.1 Les options générales

Les options globales s'appliquent à toutes les sockets. Le niveau pour ces options est `SOL_SOCKET`

— SO_BROADCAST : cette option permet d'émettre et de recevoir des paquets en diffusion.
— SO_DEBUG : cette option permet de sauvegarder des informations sur tous les messages envoyés ou reçus. TCP est le seul protocole qui supporte cette fonctionalité.
— SO_DONTROUTE : cette option active ou désactive le routage des paquets. Par défaut cette fonctionalité est désactivée.
— SO_ERROR : récupère et efface toutes les erreurs sockets en attentes.
— SO_KEEPALIVE : permet de garder la connexion ouverte en testant la communication
— SO_LINGER :
— SO_OOBINLINE :
— SO_PEERCRED :
— SO_RCVBUF :
— SO_RCVLOWAT :
— SO_REUSEADDR : Cette option permet de créer deux sockets qui partagent la même adresse et le même numéro de port. La plupart du temps, cette option est utilisée pour redémarrer rapidement un serveur suite à un arrêt brutal.
— SO_SNDBUF :
— SO_SNDLOWAT :
— SO_SNDTIMEO :
— SO_TYPE :

5.15.2 Les options spécifiques à IP

Pour utiliser les options spécifiques à IP, le paramètre `level` doit être égal à SOL_IP.

— IP_ADD_MEMBERSHIP : permet de joindre un groupe multicast
— IP_DROP_MEMBERSHIP : permet de quitter un groupe multicast
— IP_HDRINCL : cette option permet de construire l'entête du paquet IP raw. Le seul élément à ne pas remplir est la somme de contrôle.
— IP_MTU_DISCOVER :
— IP_MULTICAST_IF :
— IP_MULTICAST_LOOP :
— IP_MULTICAST_TTL : Positionne le nombre maximum de saut (*Time To Live*) autorisé pour le message multicast.
— IP_OPTIONS :
— IP_TOS : cette option permet de déterminer le type de service (ToS). Il est possible de choisir parmi IPTOS_LOWDELAY, IPTOS_THROUGHPUT, IPTOS_RELIABILITY, IPTOS_LOWCOST.
— IP_TTL : cette option permet de spécifier la durée de vie (*Time To Live*) de chaque paquet.

5.15.3 Les options spécifiques à IPv6

Pour utiliser les options spécifiques à IPv6, le paramètre `level` doit être égal à SOL_IPV6.

— IPV6_ADD_MEMBERSHIP :
— IPV6_ADDRFORM :
— IPV6_CHECKSUM :
— IPV6_DROP_MEMBERSHIP :
— IPV6_DSTOPTS :
— IPV6_HOPLIMIT :

- IPV6_HOPOPTS :
- IPV6_MULTICAST_HOPS :
- IPV6_MULTICAST_IF :
- IPV6_MULTICAST_LOOP :
- IPV6_NEXTHOP :
- IPV6_PKTINFO :
- IPV6_PKTOPTIONS :
- IPV6_UNICAST_HOPS :

5.15.4 Les options spécifiques à TCP

Pour utiliser les options spécifiques à TCP, le paramètre `level` doit être égal à SOL_TCP.

- TCP_KEEPALIVE : La valeur par défaut est 7200.
- TCP_MAXRT : cette option permet de spécifier le temps de retransmission en secondes. Un 0 correspond à la valeur par défaut du noyau. La valeur par défaut est 0.
- TCP_MAXSEG : La valeur par défaut est 540 octets.
- TCP_NODELAY :
- TCP_STDURG :

5.16 les socket en Java

Chapitre 6

Développement USB

6.1 Introduction

L'*Universal Serial Bus*, apparu en 1994, est devenu le bus de communication le plus utilisé pour la connexion entre un PC et des périphériques. Son mode de fonctionnement permettant notamment une connexion et une déconnexion "à chaud" (sans avoir à redémarrer le PC) et sa simplicité d'utilisation, l'ont rendu omniprésent. Il permet la communication entre une machine hôte (*host*) et un ensemble de périphériques (*device*) qui partagent la bande passante. USB fonctionne en mode maître-esclave (l'hôte est à l'origine de toute les requêtes).

6.2 Terminologie

- **Endpoint** (point d'accés) : un *endpoint* est une source ou un puit de données. Un périphérique USB peut avoir plusieurs *endpoint*, la limite est 16 IN et 16 OUT.
- **Transaction** : une transaction est un transfert de données sur le bus.
- **Pipe** : un tube est une connexion logique entre l'hôte et un *endpoint*.

6.2.0.1 Descripteur d'un périphérique

La hiérarchie des descripteurs d'un périphérique USB est représentée à la figure 6.1.
La commande `lsusb -vvv` vous permet d'obtenir toutes ces informations pour tous les périphériques USB attachés à votre machine.
Le tableau 6.1 présente les différentes informations composant la description d'un périphérique USB.
bcdUSB indique la plus haute version d'USB supportée. La valeur est au format 0xJJMN (JJ: major release, M: minor release, N: sub-minor release). Par exemple, pour USB 1.1, la valeur est `0x0110` et pour USB 2.0 la valeur est `0x0200`.
La notion de classe d'interface permet de simplifier l'écriture de driver. Quelques une des classes de périphériques sont visibles au tableau 6.2.

6.2.0.2 Descripteur de configuration

Le descripteur de configuration

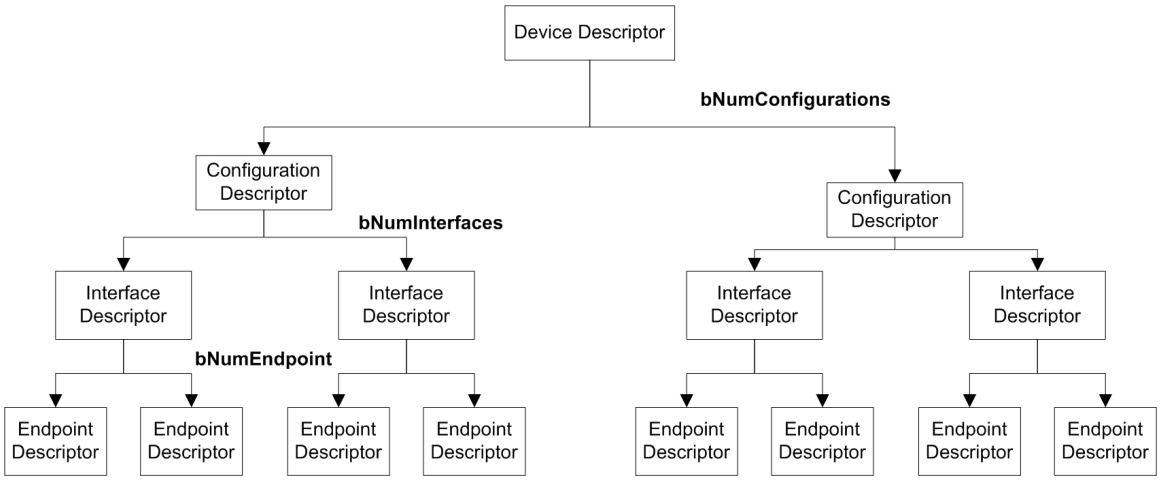


FIGURE 6.1 – Hiérarchie des descripteurs USB

6.2.0.3 Descripteur d'une interface

La description d'une interface est présentée au tableau 6.4.

6.2.0.4 Descripteur de point d'accès

ACHTUNG: vérifier le tableau endpoint descriptor dans la spec sur la taille de l'adresse du endpoint.
Un point d'accès est une portion unique et identifiable d'un périphérique USB qui est la terminaison d'un flux de communication entre un hôte et un device. Chaque device logique USB est composé d'une collection de points d'accès indépendants. Chaque point d'accès d'un device à un identifiant unique au sein du device déterminé lors de la conception du device (appelé endpoint number).
Les caractéristiques d'un endpoint déterminent le type de transfert. Il se décrit lui même par les caractéristiques du tableau 6.5.

6.3 Communication USB

6.3.1 Types de communication

La communication se déroule entre un logiciel côté hôte et un *endpoint* particulier d'un périphérique. Cette association est appelé un tube (*pipe*). Il existe quatre types de transfert de données en USB :

- *bulk* : Les transferts *bulk* consistent en de larges quantités de données comme celles utilisées par une imprimante ou un scanner. La bande passante peut varier en fonction de l'utilisation du bus.
- *interrupt* : Les transferts par interruption sont utilisés
- *isochronous* : Les transferts isochrones occupe une certaine quantité de bande passante prénégociée avec une latence prénégociée.

Décalage	Champ	Taille	Description
0	bLength	1	Taille du descripteur en octets
1	bDescriptorType	1	Type de descripteur
2	bcdUSB	2	Version USB supportée
4	bDeviceClass	1	Classe du périphérique
5	bDeviceSubClass	1	Sous-classe
6	bDeviceProtocol	1	Protocole
7	bMaxPacketSize0	1	Taille maximum du paquet
8	idVendor	2	Identifiant du vendeur
10	idProduct	2	Identifiant du produit
12	bcdDevice	2	Numéro de version du périphérique
14	iManufacturer	1	Manufacturer string descriptor
15	iProduct	1	Index of product string descriptor
16	iSerialNumber	1	index of serial number descriptor
17	bNumConfigurations	1	Nombre de configurations possible

TABLE 6.1 – Device descriptor

Classe	Description	Exemple
0x00	Réservé	-
0x01	Audio	Carte son
0x02	Communication	Modem, fax
0x03	Interface humaine (HID)	Clavier, souris
0x07	Imprimante	Imprimante
0x08	Stockage de masse	Carte mémoire
0x09	Hub	Hubs
0x0B	Lecteur de carte	
0x0E	Vidéo	Webcam, scanner
0xE0	Sans-fil	Bluetooth

TABLE 6.2 – Classes de périphériques USB

— *control* : Les transferts de type contrôle sont utilisés pour la configuration du périphérique et peuvent être utilisés pour d'autres utilisations spécifiques liées au périphérique.

6.4 Programmation noyau

to be continued ...

6.5 Programmation avec libusb-1.0

`libusb` est une bibliothèque de programmation en mode utilisateur permettant d'accéder aux périphériques USB.

Décalage	Champ	Taille	Description
0	bLength	1	Taille du descripteur en octets
1	bDescriptorType	1	Device descriptor
2	wTotalLength	2	Nombre total d'octets retournés
4	bNumInterfaces	1	Nombre d'interfaces
5	bConfigurationValue	1	Valeur utilisée pour sélectionner la configuration
6	iConfiguration	1	Index décrivant la chaîne de configuration
7	bmAttributes	1	Attributs pour la puissance électrique (Power supply)
8	bMaxPower	2	Consommation maximale

Table 6.3 – Configuration descriptor

Décalage	Champ	Taille	Description
0	bLength	1	Taille du descripteur en octets
1	bDescriptorType	1	Device descriptor
2	bInterfaceNumber	1	Numéro de l'interface
3	bAlternateSetting	1	Valeur pour sélectionner une configuration alternative
4	bNumEndpoints	1	Nombre de endpoint
5	bInterfaceClass	1	Code de la classe
6	bInterfaceSubClass	1	Code de la sous-classe
7	bInterfaceProtocol	1	Code du protocole
8	iInterface	1	Index of string descriptor to interface

Table 6.4 – Interface descriptor

6.5.1 Initialisation

```
libusb_init(NULL)
```

6.5.2 Découverte des périphériques

6.5.3 Appels synchrones

```
libusb_control_transfer()
```

Décalage	Champ	Taille	Description
0	bLength	1	Taille du descripteur en octets
1	bDescriptorType	1	Endpoint
2	bcdEndpointAddress	1	Adresse du endpoint
3	bmAttributes	1	Type de endpoint
4	wMaxPacketSize	2	Taille maximale du paquet
6	bInterval	1	Polling interval

Table 6.5 – Endpoint descriptor

Décalage	**Champ**	**Taille**	**Description**
0	bLength	1	Taille du descripteur en octets
1	bDescriptorType	1	HID (0x21)
2	bcdHID	2	Classe HID
4	bCountryCode	1	code spécial dépendant du pays
5	bNumDescriptors	1	Nombre de descripteurs additionnels
6	bDescriptorType	1	Type de descripteur additionnel
7	wDescriptorLength	2	Longueur du descripteur addtitionel