IUT A — Université NANCY II
Département Informatique
2000/2001

Poly Système

Bernard Mangeol, Isabelle Chrisment,
Jean François Mari et Denis Roegel

Table des matières

Première partie

Rappels

Chapitre 1

Les outils Unix pour développer en C

1.1 L'éditeur Emacs

Nous donnons ici les plus utilisées des commandes de l'éditeur Emacs. Dans `C-x`, le symbole `C` signifie *control*. Ainsi pour exécuter la séquence `C-x C-s` qui sauve un fichier, il faut simultanément appuyer sur `ctrl` et `x`, relâcher et enfin sans trop attendre, appuyer sur `ctrl` `s`.

Contrairement à la touche *control* qui s'utilise comme la touche *shift* ou ↑, la touche *escape* s'utilise comme une touche de caractère ordinaire. Pour reculer d'un mot — séquence `ESC b` — il faut appuyer d'abord sur la touche *escape* puis la lettre *b*. La table 1.1 donne les principales commandes de l'éditeur `Emacs`. On peut l'obtenir sous `Emacs` par la commande `ESC x describe-bindings`. Ne tapez pas tout ! Utilisez les touches *espace* ou *Tab* pour complémenter les commandes, comme sous *tcsh*.

1.2 Le compilateur gcc

`gcc` est un des compilateurs `C` les plus performants. La version installée 2.8.1 compile aussi bien des programmes `C` que des programmes écrits en `C++`. L'extension naturelle d'un programme `C` est `.c`, celle d'un programme écrit en `C++` est `.C`. `gcc` effectue aussi l'édition de lien. Le programme exécutable qui est produit s'appelle `a.out` par défaut.

1.2.1 Exemples d'utilisation

`gcc prog.c` compilation du fichier `prog.c`, édition de lien, l'exécutable s'appelle `a.out` ;

`gcc prog.c -o prog` compilation du fichier `prog.c`, édition de lien ; l'exécutable s'appelle `prog` ;

`gcc prog.c -o prog -lm` idem mais l'édition de lien utilise la bibliothèque mathématique (sin, cos, etc.) ;

`gcc -c prog.c` uniquement compilation du fichier `prog.c`, pas d'édition de lien ;

`gcc prog1.c ssp.c -o prog` compilation de deux fichiers `C`, édition de lien des deux fichiers, l'exécutable s'appelle `prog` ;

`gcc prog1.c prog.o -o prog` compilation de `prog1.c` et édition de lien avec `prog.o` qui a déjà été compilé.

1.2.2 Les options de compilation

`-c` supprime l'édition de lien. Le code objet compilé se trouve dans le fichier d'extension `.o`.

`-g` ajoute les informations pour déboguer à l'aide de `gdb` ; on peut aussi utiliser l'option `-ggdb` (mais elle est légèrement différente, cf. man).

`-Wall` (*Warning all*) ajoute tous les messages d'avertissements. Cette option ralentit la compilation mais accélère la mise au point du programme.

`-O2` optimise le code en taille et vitesse. Cette option accélère la vitesse d'exécution.

1.3 Le débogueur `gdb`

Un débogueur permet un contrôle du programme pendant son exécution. On peut, en autre chose, arrêter le programme en posant des points d'arrêt, visualiser le contenu des variables et connaître l'instruction qui a provoqué une erreur fatale. Les étudiants n'aiment pas utiliser de débogueurs, cela ne représente aucun intérêt pour eux car ils écrivent des programmes justes. Seuls les enseignants utilisent les débogueurs pour retrouver les erreurs des programmes des étudiants. Signalons qu'il existe des variantes de `gdb` telles que `ddd` ou `xxgdb`. Mais celles-ci nécessitent le système `X`.

1.3.1 Les principales commandes de `gdb`

Considérons un programme dont l'exécution se termine par le message `segmentation fault, core dumped`. Voici dans l'ordre les commandes du débogueur qui permettent de localiser et comprendre l'erreur.

`run` lance l'exécution ;

`where` donne l'instruction qui a provoqué l'erreur fatale ;

`up` permet de remonter dans l'enchaînement des appels de sous-programmes jusqu'au programme utilisateur intéressant ;

`list` imprime les lignes `C` pour en connaître les numéros ;

`break` ⟨*numéro*⟩ pose un point d'arrêt sur une ligne dont on donne le numéro ;

`run` relance le programme qui s'arrêtera sur le point d'arrêt qui vient d'être posé ;

`print` *variable* imprime le contenu d'une variable ;

`next` exécute une instruction (fastidieux) ;

`step` exécute une instruction en entrant, le cas échéant, dans les sous-programmes (encore plus fastidieux !).

`continue` permet de continuer jusqu'au prochain point d'arrêt.

1.4 Les Makefiles

Ce paragraphe est rédigé à partir du cours de licence d'informatique de Nancy [7] et du livre de Matt Welsh [6].

1.4.1 Définition

`make` est une commande Unix qui permet à partir d'un fichier de description d'exécuter une séquence de commandes afin :

- de construire la version exécutable d'une application ;
- d'installer des logiciels ;
- de nettoyer des fichiers temporaires et de déplacer des fichiers ;
- de mettre à jour des bibliothèques ;
- ...

Afin de construire la version exécutable d'une application, le programmeur avisé a recours à la compilation séparée qui permet :

- de faire le moins de travail possible ;
- de ne re-compiler que les fichiers qui ont été modifiés ou qui incluent des fichiers modifiés ;
- d'éviter d'écrire des commandes de compilation et d'édition inutiles, car les fichiers produits seraient les mêmes qu'auparavant.

`make` est un programme qui (si le `Makefile` est bien fait) ne re-compile que ce qui a changé puis qui réalise l'exécutable à partir de ce nouvel élément et des anciens objets toujours valides ; il suffit de taper `make`, c'est tout.

1.4.2 Comment ça marche ?

Le but de `make` est de vous permettre de construire un fichier par petites étapes. Afin d'y parvenir, `make` enregistre quels fichiers sont nécessaires pour construire votre projet. Cette description est contenue dans un fichier qui s'appelle en général `Makefile` ou `makefile` et qui se situe dans le même répertoire que vos fichiers sources. Voici un `Makefile` très simple :

```
client : protocole.o ppclient.o
tabulation gcc -o client  protocole.o ppclient.o
```

```
ppclient.o : ppclient.c protocole.h
 tabulation gcc -c -Wall ppclient.c

protocole.o :  protocole.c  protocole.h
 tabulation gcc -c -Wall protocole.c
# c'est fini
```

Grâce à la directive `#include "protocole.h"`, les deux fichiers sources `protocole.c` et `ppclient.c` incluent un fichier `protocole.h` qui se situe dans le même répertoire

Nous voyons qu'il y a trois entrées ou trois cibles. Chaque entrée débute par une ligne qui spécifie les dépendances. La première ligne spécifie que l'exécutable `client` dépend d'un sous-programme `protocole.o` et du programme principal `ppclient.o`. La deuxième ligne donne l'action à accomplir pour construire la cible `client`. Il s'agit d'une édition de lien. Attention, comme toute ligne de commande, celle-ci doit débuter par une tabulation. C'est la cause de beaucoup d'erreurs de syntaxe avec `make`.

Les entrées suivantes spécifient les dépendances des fichiers `protocole.c` et `ppclient.c`. Ces deux fichiers doivent être re-compilés à chaque modification et si le fichier d'en-tête `protocole.h` venait à être modifié. Dans les deux cas, l'action à accomplir est l'appel au compilateur `gcc` avec l'option `-c`.

La dernière ligne débutant par un `#` est une ligne de commentaire.

À chaque modification des programmes `C` ou du fichier inclus, il suffira de faire `make client` ou plus simplement `make` pour que `make` ne fasse que le strict minimum.

Voici un autre `Makefile` un peu plus long. Il possède une cible appelée `clean` dont le rôle est de nettoyer le répertoire des fichiers produits par `gcc`. C'est le cas quand il faut tout recompiler pour passer d'une version compilée avec l'option `-ggdb` à une version finale optimisée et compilée avec l'option `-O2`. On voit que `make` ne sert pas qu'à invoquer le compilateur. Toutes les commandes Unix sont possibles ; mais attention à la tabulation en début de ligne !

Dans cet exemple, l'application est constituée de deux programmes appelés `client` et `serveur`. Pour tout constuire il suffit de faire `make all`.

```
# pour effectuer le ménage
clean :
 tabulation rm  protocole.o ppclient.o client serveur ppserveur.o

#pour tout reconstruire
all : client serveur

# pour construire le client

client : protocole.o ppclient.o
 tabulation gcc -o client  protocole.o ppclient.o

ppclient.o : ppclient.c protocole.h
 tabulation gcc -c -Wall ppclient.c
```

```
protocole.o :  protocole.c  protocole.h
 tabulation gcc -c -Wall protocole.c

# pour construire le serveur

serveur : protocole.o ppserveur.o
 tabulation gcc -o serveur protocole.o ppserveur.o

ppserveur.o :  protocole.h ppserveur.c
 tabulation gcc  -c -Wall ppserveur.c
# c'est fini
```

Lorsque `make` est appelé sans paramètre, c'est la première cible qui est exécutée.

1.4.3 Quelques règles de syntaxe

Une ligne de commande débute par une tabulation. Une ligne trop longue peut être repliée en tapant le caractère `\` en fin de ligne. Vous pouvez placer un dièse (`#`) à n'importe quel endroit de la ligne pour introduire un commentaire ; tout ce qui suit sera ignoré.

Il est possible d'introduire des macro-définitions. Il s'agit de chaînes de caractères que `make` expanse en d'autres. Dans l'exemple suivant, les macros `OBJETS_CLI` et `OBJETS_SER` permettent d'économiser de la frappe au clavier et de reposer les petits doigts fatigués.

```
OBJETS_CLI =  protocole.o ppclient.o
OBJETS_SER =  protocole.o ppserveur.o
CFLAGS = -c -Wall

# pour effectuer le ménage
clean :
 tabulation rm  client serveur $(OBJETS_CLI)  $(OBJETS_SER)

#pour tout reconstruire
all : client serveur

# pour construire le client

client : $(OBJETS_CLI)
 tabulation gcc -o client  $(OBJETS_CLI)

ppclient.o : ppclient.c protocole.h
 tabulation gcc $(CFLAGS) ppclient.c

protocole.o :  protocole.c  protocole.h
 tabulation gcc $(CFLAGS) protocole.c
```

```
# pour construire le serveur

serveur : $(OBJETS_SER)
 tabulation gcc -o serveur  $(OBJETS_SER)

ppserveur.o :  protocole.h ppserveur.c
 tabulation gcc  $(CFLAGS) ppserveur.c
# c'est fini
```

Une macro non définie est remplacée par la chaîne vide. On peut définir une macro à l'appel de `make`. Par exemple, si on désire un comportement versatile du compilateur pour compiler tantôt avec l'option `-O2`, tantôt avec l'option `-ggdb`, on écrira :

```
DEBUG=-ggdb
...
 tabulation gcc  $(CFLAGS) $(DEBUG)  protocole.c
...
```

Si on veut une compilation optimisée, on invoquera la commande :

```
make DEBUG=-O2
```

Cette affectation supplantera celle faite dans le `Makefile` par la définition `DEBUG=-ggdb` qui restera la définition par défaut.

1.4.4 Règles implicites

Pour des opération routinières comme la construction d'un fichier objet à partir d'un fichier source, spécifier à chaque fois les dépendances est long et pénible. Sous Unix, un fichier se terminant par `.c` sera toujours compilé en fichier se terminant par `.o`. La commande `make` permet de définir des règles implicites pour traiter facilement ce type de cas. Voici une de ces règles pour traiter le cas de la compilation des programmes `C`.

```
...
.c.o :
 tabulation gcc  $(CFLAGS) $(DEBUG)  $<
...
```

Le signe `.c.o :` signifie « prendre un fichier `.c` en entrée et le transformer en fichier `.o` ». La chaîne `$<` est un code représentant le fichier d'entrée. Notre `Makefile` peut se réduire encore en utilisant cette règle.

```
.c.o :
        gcc  $(CFLAGS) $(DEBUG)  $<
OBJETS_CLI =  protocole.o ppclient.o
OBJETS_SER =  protocole.o ppserveur.o
CFLAGS = -c -Wall
```

```
# pour effectuer le ménage
clean :
	rm  client serveur $(OBJETS_CLI)  $(OBJETS_SER)

#pour tout reconstruire
all : client serveur

# pour construire le client

client : $(OBJETS_CLI)
	gcc -o client  $(OBJETS_CLI)

ppclient.o : ppclient.c protocole.h

protocole.o :  protocole.c  protocole.h

# pour construire le serveur

serveur : $(OBJETS_SER)
	gcc -o serveur  $(OBJETS_SER)

ppserveur.o :  protocole.h ppserveur.c

# c'est fini
```

1.5 Envoi de programmes par courrier électronique

De nos jours, la plupart des mailers sont très sophistiqués et semblent avoir éliminé les problèmes que posait l'envoi de messages il y a encore quelques années. Il est facile avec `netscape` ou d'autres outils d'ajouter des fichiers attachés, que ce soient des images ou du code compilé. Cette apparente facilité cache ce qui se passe réellement, à savoir un codage des messages pour que ceux-ci puissent franchir sans difficulté les différentes passerelles de mail. En règle générale, les caractères 8 bits (de code supérieur à 127) présentent beaucoup de risques d'être corrompus durant un transfert. Que ce soit explicitement ou implicitement, les outils codent souvent en ASCII (donc avec un code inférieur à 128) les messages à envoyer. Les possibilités de codage dépendent des mailers et dans certains cas, il est utile de connaître une marche à suivre pour coder soi-même les fichiers à envoyer.

Nous présentons ici une méthode qui permet d'envoyer sans corruption un répertoire de fichiers. Ces fichiers seront d'abord rassemblés dans une archive (`.tar`), puis compressés (avec `gzip`), codés (avec `uuencode`) et envoyés avec `mail`. Une procédure analogue permet de les récupérer. Les différents outils employés sont standard sous Unix.

La procédure permettant d'envoyer un ensemble de fichiers ou/et répertoires est la suivante. Nous supposons qu'il s'agit d'envoyer les fichiers `README`, `projet.tex` et `programme.c` :

– Création d'un répertoire `projet` : `mkdir projet`

- Les trois fichiers sont mis dans le répertoire : `mv README projet/`, etc.
- Le répertoire est archivé : `tar cvf projet.tar projet`
- Le fichier d'archive est compressé : `gzip projet.tar`
- Le fichier compressé est uuencodé :
  `uuencode projet.tar.gz projet.tar.gz > projet.tar.gz.uue`
  (attention à bien écrire `>` et non `<`)
- Le fichier `projet.tar.gz.uue` m'est envoyé : `mail enseignant@adresse.fr < projet.tar.gz.uue`
  (attention à bien écrire `<` et non `>`)

Dans tous les cas, avant d'envoyer un fichier de cette manière, il faut tester l'envoi sur vous-même ou l'un de vos camarades. Il ne suffit pas de tester l'envoi, mais il faut aussi vérifier que l'étape inverse se passe bien. Celui qui reçoit votre message devra procéder comme suit :

- Sauvegarde du message dans un fichier.
- Appel de `uudecode` sur ce fichier. Cela produit `projet.tar.gz`.
- Décompression : `gunzip projet.tar.gz`.
- Désarchivage : `tar xvf projet.tar`. Ceci produit le répertoire `projet`.

Concernant les fichiers envoyés, une bonne règle est de ne pas choisir de noms comportant des espaces ou des caractères accentués. Les mêmes règles pourront s'appliquer aux noms des identificateurs (variables, fonctions, etc.)

Rappelons enfin qu'il faut éviter dans la mesure du possible d'envoyer des fichiers inutiles et en particulier des fichiers binaires comme des fichiers exécutables qui pourront être recréés par le destinataire.

touche	commande
C-a	début de ligne
C-e	fin de la ligne
C-b	reculer d'un caractère (quand la flèche ne marche plus)
C-f	avancer d'un caractère
C-n	ligne suivante
C-p	ligne précédente
C-d	détruire un caractère
ESC a	reculer d'une phrase (recherche le point précédent)
ESC f	mot suivant
ESC b	reculer d'un mot
ESC d	détruire un mot
ESC g	aller à la ligne de numéro donné
C-g	fin de commande
C-k	détruire le reste de la ligne (= couper) et le ranger dans le buffer de destruction
C-l	repeindre l'écran
C-q	quoted-insert.(pour insérer des caractères de contrôle)
C-r	recherche arrière
C-s	recherche avant
C-u	argument universel
C-v	scroll-up
C-w	couper une région dans le buffer de destruction
C-y	insère le buffer de destruction (= coller)
C-x C-b	liste les noms de buffers
C-x C-c	sauver et sortir d'Emacs
C-x C-f	trouve un fichier pour l'éditer
C-x C-r	trouve un fichier en consultation (Read Only)
C-x C-s	sauvegarder sans quitter
C-x C-w	écrire un buffer sur fichier
C-x C-x	échanger la position courante et la marque (cf. C-@)
C-x (	début de macro
C-x)	fin de macro
C-x E	exécuter la macro
C-x 0	détruire une fenêtre
C-x b	changer de buffer
C-x i	insérer un fichier
C-x k	détruire un buffer
C-x s	sauver certains buffers
C-x !	shell-command
ESC C-r	substitution à la demande
ESC	(*escape espace*)place une marque

TAB. 1.1 – *Aide mémoire* `de Micro-Emacs`. *On peut obtenir ce tableau à l'aide de la commande* `ESC x describe-bindings`

Chapitre 2

Compléments en C

2.1 La manipulation des chaînes de bits

2.1.1 Définitions de constantes dans différentes bases

On peut définir des constantes dans diverses bases en C. Ainsi la constante notée 128 en base 10 s'écrira :

- 128 en base 10 ;
- 0200 en base 8 (octal) ;
- 0x80 en base 16 (hexa) ;

Les notations octales et hexadécimales servent généralement en système pour coder des constantes de type `int` et `char` et leurs dérivés.

Exemple :

```
int eol1 = 0x0D0A ; /* CR et LF en ASCII */
char  cr = 0x0D ;   /* CR */
```

2.1.2 Les opérateurs logiques ET et OU

L'opérateur ET (noté &) agit entre deux bits et a pour résultat la multiplication ; l'opérateur OU (noté |) agit entre deux bits et a pour résultat le maximum des deux bits. Ces opérateurs peuvent agir sur deux chaînes de bits rangées dans des variables de type `char` (= 8 bits) ou `int` (= 16, 32 ou 64 bits suivant l'architecture) en agissant bit à bit sur les composants des chaînes.

Exemple :

```
int eol1 = 0x0D0A ;
int mask = 0x00FF ;
```

```
int resEt  = eol1 & mask ; /* le résultat sera 000A */
int resOu = eol1 | mask ;  /* le résultat sera 0DFF */
```

2.1.3 Les opérateurs de décalage de bits

Il est possible de décaler vers la gauche (opérateur `<<`) une chaîne de bits, il s'agit d'une multiplication par une puissance de deux ; les bits qui entrent à droite sont des zéros, les bits qui sortent à gauche sont perdus. Le décalage à droite se note `>>`, il s'agit d'une division par une puissance de deux ; des zéros entrent à gauche et les bits qui sortent à droite sont perdus.

Exemple :

```
int eol1 = 0x0D0A ;
int left8 = eol1  << 8 ; /* décal. à gauche de 8 bits, le résu. est  0x0A00 */
int right4 = eol1 >> 4 ; /* décal. à droite de 4 bits, ...        :  0x00D0 */
```

2.1.4 Positionnement d'un bit à 1

Si on connaît le rang du bit, il suffit d'utiliser OU avec une constante appelée masque.

Exemple : mettre à 1 le bit n° 3 de la variable `mot` : `mot | 0x08`. Le bit le plus à droite a le rang zéro par convention.

On veut placer le bit n° `i` à 1 dans un mot. On agit par décalage et masque.

```
int i, mot ;
int res ;
... /* affectation de mot et i */
res = mot | (1 << i) ;
```

2.1.5 Positionnement d'un bit à 0

Si on connaît le rang du bit, on effectue un masque avec un ET.

Exemple : mettre à 0 le bit n° 3 de la variable mot : `mot & 0xF7`.

On veut placer le bit n° `i` à zéro dans un mot. On agit par décalage, complémentation et masque.

```
int i, mot ;
int res ;
... /* affectation de mot et i */
res = mot &  ~(1 << i) ;
```

2.1.6 Tester un bit

Encore une fois, on a deux possibilités suivant que l'on connaît ou non le rang du bit à la compilation. Soit on peut calculer la valeur du masque à la compilation, soit on calcule ce masque à l'exécution.

Exemple : tester le bit no. 3 de la variable mot

```
int mot ;
...
if (mot & 0x08 == 0x08)
  /* le bit no. 3 vaut 1 */
else
 /* devine ! */
```

2.2 Définition d'un pointeur

Un pointeur est une variable qui contient l'adresse d'une autre variable. Les pointeurs ont une grande importance en `C` ; aussi consacrerons nous une section particulière sur leurs utilisations en système d'exploitation et sur les sources d'erreurs dont ils sont la cause.

Pour déclarer un pointeur, il faut préciser quel sera le type de la variable dont il contiendra l'adresse. Cela se réalise à l'aide du symbole `*`.

Exemple : `int *pti ;` déclare un pointeur appelé `pti` qui contiendra l'adresse d'une variable de type `int`.

2.3 Utilisation des pointeurs

2.3.1 L'opérateur &

La première opération qu'on est amené à effectuer sur un pointeur est de préciser vers quelle variable il pointe. Cela consiste à mettre dans le pointeur une adresse, le plus souvent celle d'une variable déjà déclarée. On utilise l'opérateur `&` placé devant une variable pour en avoir l'adresse. `&var` signifie l'adresse de la variable `var`.

```
void main()
{
  int unObjetEntier ;
  int *pe ;

  pe = &unObjetEntier ; /* pe est affecté avec l'adresse de la
                           variable entière unObjetEntier */
  ...
```

```
}
```

2.3.2 Propriétés de l'opérateur &

L'opérateur `&` ne s'applique qu'aux variables simples (même de type pointeur) et indicées ainsi qu'aux structures. Sans précautions particulières, on ne peut pas écrire `&tab` si `tab` est un tableau ; par définition `tab` est une constante qui représente l'adresse du premier élément du tableau.

2.3.3 L'opérateur d'accès indirect *

Ce type d'adressage souvent utilisé dans les langages d'assemblage, consiste à utiliser un pointeur pour accéder à une autre variable. Il y a deux accès en mémoire : le premier pour lire la valeur du pointeur et le deuxième pour accéder à cette adresse soit en lecture, soit en écriture. En `C`, ce type d'adressage est réalisé grâce à l'opérateur `*` placé devant un pointeur.

`*pe` signifie une des trois assertions suivantes :

- accès indirect au travers de `pe` ;
- l'objet dont l'adresse est dans `pe` ;
- l'objet pointé par `pe`.

Attention, cela ne signifie pas : *contenu de pe*.

Exemple :

```
void main()
{
  int unObjetEntier ;
  int *pe ;

  pe = &unObjetEntier ; /* pe est affecté avec l'adresse de la
                           variable entière unObjetEntier */
  *pe = 1515 ;          /* on aurait pu aussi bien écrire :
                           unObjetEntier = 1515 ; */
}
```

Attention, avant d'utiliser un pointeur, il faut lui donner une valeur. Ceci est réalisé, soit par affectation, soit dans un appel de fonction. Regardez ce programme qui n'a pas de sens :

```
void main()
{
  char *buffer ;        /* un pointeur vers une mémoire tampon */
  FILE *pf ;          /* un pointeur de fichier */
  ...
  fread(buffer, 1, 80, pf) ; /* lecture de 80 caractères */

}
```

Ce programme effectue une lecture de 80 caractères depuis un fichier `pf` et les range dans une zone dont l'adresse est dans `buffer`. Mais `buffer` ne contient rien de bon ; le programme écrira n'importe où ! Pour corriger cette erreur, on peut utiliser une déclaration de tableau `char buffer[80]` à la place de la déclaration de pointeur `char *buffer`, ou bien encore faire un `malloc`.

2.3.4 L'opérateur d'accès indexé []

Cet opérateur qui se place derrière un identificateur de tableau, s'applique aussi aux pointeurs. Il permet de se déplacer dans la zone mémoire qui débute à l'adresse contenue dans le pointeur.

Exemple :

```
void main()
{
  char *buffer ;        /* un pointeur vers une mémoire tampon      */
  buffer = (char *) malloc(80) ; /* qui pointe vers une chaîne de caractères */
  ...
  /* initialisation avec 80 retours chariot */
  for (i = 0 ; i < 80 ; i++) buffer[i] = '\n' ;
  ...
  free(buffer) ; /* ne pas oublier ! */

}
```

2.4 Les tableaux de pointeurs

On définit un tableau de pointeurs par la déclaration :

`Type *identificateur[nElem] ;`

L'opérateur `[]` est plus prioritaire sur l'opérateur `*`. Il s'agit d'un tableau de `nElem` pointeurs vers des `Type`.

2.4.1 Exemple 1

Considérons un tableau de sept chaînes contenant les jours de la semaine.

```
void main()
{
  char *jours[7] ;
  ...
  jours[0] = "Lundi" ;
  jours[1] = "Mardi" ;
  ...
  jours[6] = "Dimanche" ;
}
```

On peut aussi user de la déclaration et de l'initialisation suivantes :

```
char *jours[7] = {
      "Lundi",
      "Mardi",
      ...
      "Dimanche"} ;
```

2.4.2 Les arguments de la commande `main`

En `C`, il est possible de passer des paramètres à un programme principal pour en faire une commande Unix qui accepte des paramètres.

Exemple : `com arg1 arg2 arg3`

Ces paramètres sont facilement récupérables ! Il suffit dans un premier temps de déclarer le `main` comme une procédure qui accepte deux paramètres qui sont nommés conventionnellement `argc` et `argv`.

```
void main(int argc, char *argv[])
{
...
}
```

`argc` est une variable entière qui contient à l'appel du programme principal *le nombre de mots de la ligne de commande*. `argv` est un tableau de pointeurs vers des chaînes de caractères, tout comme le tableau `jours`. Ces chaînes sont les différents mots (ou champs) de la ligne de commande. Si la ligne de commande est `com arg1 arg2 arg3`, on aura au début du programme principal :

- `argc` qui contiendra 4 ;
- `argv[0]` qui pointera vers la chaîne `com` ;
- `argv[1]` qui pointera vers la chaîne `arg1` ;
- `argv[2]` qui pointera vers la chaîne `arg2` ;
- `argv[3]` qui pointera vers la chaîne `arg3`.

2.5 Exemple récapitulatif : la fonction Unix `echo`

La fonction Unix `echo` reproduit des arguments sur la sortie standard (l'écran en général). Une version simplifiée peut se programmer ainsi :

```
/* fonction monecho.c
** version simplifiée de la fonction echo Unix
*/
#include <stdio.h>
```

```
main(int argc, char *argv[])
{
  int i ;

  for (i = 1 ; i < argc ; i++)      /* impression des arguments    */
     printf("%s ", argv[i]) ;       /* un par un sur la même ligne */
  printf("\n") ;                    /* passage à la ligne          */
}
```

2.5.1 Exemple d'interaction avec le *shell*

Considérons la compilation et l'exécution de notre fonction `echo`. Au début, tout se passe bien.

```
Systeme> gcc -Wall monecho.c -o monecho
Systeme> monecho un programme qui marche
un programme qui marche
Systeme>
```

Essayons avec le paramètre `*` :

```
Systeme> monecho *
#monecho.c# monecho monecho.c monecho.c~
Systeme>
```

Mais pourquoi ce résultat étrange ? Je voulais simplement imprimer le caractère `*` et j'obtiens la liste des fichiers du répertoire ! La réponse à cette question est aisée si on a bien compris le fonctionnement de l'interpréteur de commandes : *le shell*. Quand celui-ci analyse la ligne de commande, il remplace le caractère `*` par la liste de tous les fichiers. Ainsi la ligne de commande est considérablement allongée avant la début du programme principal.

Deuxième partie

Fichiers et processus

Chapitre 3

Le système de gestion de fichiers

3.1 Entrées/sorties fichiers

3.1.1 Notion de table des *i-nœuds*

Cette table est placée au début de chaque région de disque contenant un système de fichiers UNIX. Chaque nœud d'index de cette table — ou *i-nœuds*, ou encore *inode* — correspond à un fichier et contient des informations essentielles sur les fichiers inscrits sur le disque :

- le type du fichier (détaillé plus bas) ;
- le nombre de liens (nombre de noms de fichier donnant accès au même fichier) ;
- le propriétaire et son groupe ;
- l'ensemble des droits d'accès associés au fichier pour le propriétaire du fichier, le groupe auquel appartient le propriétaire, et enfin tous les autres usagers ;
- la taille en nombre d'octets ;
- les dates du dernier accès, de la dernière modification, et du dernier changement d'état (quand le nœud d'index lui-même a été modifié) ;
- des pointeurs vers les blocs du disque contenant les données du fichier.

La structure `stat` correspondante est définie comme suit, dans le fichier `<sys/stat.h>` :

```
struct stat {
     dev_t     st_dev;   /* identificateur du périphérique              */
                         /* où se trouve le fichier                     */
     ino_t     st_ino;   /* numéro du nœud d'index                      */
     mode_t    st_mode;  /* droits d'accès du fichier                   */
     nlink_t   st_nlink; /* nombre de liens effectués sur le fichier    */
     uid_t     st_uid;   /* identificateur du propriétaire              */
     gid_t     st_gid;   /* identificateur du groupe du propriétaire    */
     dev_t     st_rdev;  /* type de périphérique                        */
```

```
    off_t     st_size;  /* taille en octets du fichier                */
    time_t    st_atime; /* date du dernier accès au fichier           */
    time_t    st_mtime; /* date de la dernière modification du fichier */
    time_t    st_ctime; /* date du dernier changement du nœud d'index  */
} ;
```

La structure `stat` peut être obtenue par l'intermédiaire de la fonction `stat` :

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

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

Le premier paramètre est un pointeur sur un nom de fichier et le second pointe vers une structure `stat` qui va être remplie par l'appel de la fonction.

Remarque

La structure `stat` ne contient ni le nom du fichier, ni les données. Les types peuvent différer suivant l'architecture. Par exemple, sous HP-UX, `st_mode` et `st_nlink` sont de type `ushort`.

3.1.2 Les types de fichiers

Il y a trois types de fichiers UNIX :

- les fichiers ordinaires : tableaux linéaires d'octets identifiés par leur numéro d'index ;
- les répertoires : ces fichiers permettent de repérer un fichier par un nom plutôt que par son numéro d'index dans la table de nœud d'index ; le répertoire est donc constitué d'une table à deux colonnes contenant d'un côté le nom que l'utilisateur donne au fichier, et de l'autre, le numéro d'index donné par le système qui permet d'accéder à ce fichier. Cette paire est appelée un lien ;
- les fichiers spéciaux, périphériques, tubes, sockets, ..., que nous aborderons plus loin.

3.1.3 Descripteurs de fichier

Nous avons vu que le nœud d'index d'un fichier est la structure d'identification du fichier vis-à-vis du système. Lorsqu'un processus veut manipuler un fichier, il va utiliser plus simplement un entier appelé descripteur de fichier. L'association de ce descripteur au nœud d'index du fichier se fait lors de l'appel à la primitive `open()` (voir en open cf. 3.1.5). Le descripteur devient alors le nom local du fichier dans le processus. Chaque processus UNIX dispose de 20 descripteurs de fichier, numérotés de 0 à 19. Par convention, les trois premiers sont toujours ouverts au début de la vie d'un processus :

- le descripteur de fichier 0 est l'entrée standard (généralement le clavier) ;
- le descripteur de fichier 1 est la sortie standard (généralement l'écran) ;

– le descripteur de fichier 2 est la sortie erreur standard (généralement l'écran) ;

Les 17 autres sont disponibles pour les fichiers et les fichiers spéciaux que le processus ouvre lui-même. Cette notion de descripteur de fichier est utilisée par l'interface d'entrée/sortie de bas niveau, principalement avec les primitives `open()`, `write()`, `read()`, `close()`.

3.1.4 Pointeurs vers un fichier

En revanche, lorsqu'on utilise les primitives de la bibliothèque standard d'entrées/sorties `fopen`, `fread`, `fscanf`, ..., les fichiers sont repérés par des pointeurs vers des objets de type `FILE` (type défini dans le fichier `<stdio.h>`). Il y a trois pointeurs prédéfinis :

– `stdin` qui pointe vers le tampon de l'entrée standard (généralement le clavier) ;
– `stdout` qui pointe vers le tampon de la sortie standard (généralement l'écran) ;
– `stderr` qui pointe vers le tampon de la sortie d'erreur standard (généralement l'écran).

3.1.5 Description de quelques primitives et fonctions

Primitive `creat()`

```
int creat(const char *path, mode_t perm)
/* crée un fichier */
/*  path = nom du fichier */
/*  perm = droits d'accès */
```

Valeur retournée : descripteur de fichier ou -1 en cas d'erreur.

Cette primitive réalise la création d'un fichier, dont le nom est donné dans le paramètre `path`. L'entier `perm` définit les droits d'accès. Si le fichier n'existait pas, il est ouvert en écriture. Ce n'est pas une erreur de créer un fichier qui existait déjà. Reportez vous au manuel en ligne pour interpréter correctement les droits d'accès. Pour créer un fichier de nom « `essai_creat` » avec les autorisations lecture et écriture pour le propriétaire et le groupe, on écrira:

```
if ((fd = creat("essai_creat",0660)) == -1)
  perror("Erreur creat()") ;
```

Primitive `open()`

La fonction `open()` a deux profils : avec 2 ou 3 paramètres.

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

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

Valeur retournée : descripteur de fichier ou -1 en cas d'erreur.

Cette primitive permet d'ouvrir (ou de créer) le fichier de nom `pathname`. L'entier `flags` détermine le mode d'ouverture du fichier. Le paramètre optionnel `mode` n'est utilisé que lorsque `open()` réalise la création du fichier. Dans ce cas, il indique pour celui-ci les droits d'accès donnés à l'utilisateur, à son groupe et au reste du monde. Le paramètre `flags` peut prendre une ou plusieurs des constantes symboliques (qui sont dans ce cas séparées par des OU), définies dans le fichier d'inclusion `fcntl.h`.

00000	`O_RDONLY`	ouverture en lecture seule
00001	`O_WRONLY`	ouverture en écriture seule
00002	`O_RDWR`	ouverture en lecture et écriture
00010	`O_APPEND`	ouverture en écriture à la fin du fichier
00400	`O_CREAT`	création du fichier s'il n'existe pas (seul cas d'utilisation de l'argument `perm`)
01000	`O_TRUNC`	troncature à la longueur zéro, si le fichier existe
02000	`O_EXCL`	provoque un échec si le fichier existe déjà et si `O_CREAT` est positionné,

Exemple :

Pour effectuer la création et l'ouverture du fichier «`essai_open`» en écriture avec les autorisations de lecture et écriture pour le propriétaire et le groupe, il faut écrire :

```
if ((fd = open("essai_open" , O_WRONLY | O_CREAT | O_TRUNC, 0660)) == -1)
 perror("Erreur open()") ;
```

Remarque : l'inclusion du fichier `<sys/types.h>` est nécessaire, car des types utilisés dans `<fcntl.h>` y sont définis.

Fonction `fdopen()`

Cette fonction permet de faire le pont entre les manipulations de fichiers de la librairie standard C, qui utilise des pointeurs vers des objets de type `FILE` (`fclose()`, `fflush()`, `fprintf()`, `fscanf()`), et les primitives de bas niveau `open()`, `write()`, `read()` qui utilisent des descripteurs de fichiers de type `int`.

```
#include <stdio.h>

FILE* fdopen(int fd,const char *mode)
          /* convertit un descripteur de fichier en        */
          /* un pointeur sur un fichier                    */
          /* fd :   descripteur concerné par la conversion */
          /* mode : mode d'ouverture désiré                */
```

Valeur retournée : un pointeur sur le fichier associé au descripteur `fd`, ou la constante `NULL` (prédéfinie dans `<stdio.h>`) en cas d'erreur.

Remarque :

Le fichier doit, au préalable, avoir été ouvert à l'aide de la primitive `open()`. D'autre part, le paramètre `mode` choisi doit être compatible avec le mode utilisé lors de l'ouverture par `open`. Ce paramètre peut prendre les valeurs suivantes :

- `"r"` : le fichier est ouvert en lecture.
- `"w"` : le fichier est créé et ouvert en écriture. S'il existait déjà, sa longueur est ramenée à zéro.
- `"a"` : ouverture en écriture à la fin du fichier (avec création préalable si le fichier n'existait pas).

Exemple :

```
/* ouverture préalable par open() par exemple en lecture */
if ((fd = open("mon_fichier", O_RDONLY, 0666)) == -1) {
        perror("Erreur open()") ;
        exit(-1) ;
}
/* association de fp (de type FILE*) à fd (de type int)   */
if ((fp = fdopen(fd, "r")) == NULL) {
        perror("Erreur fdopen()") ;
        exit(-1) ;
}
```

Primitive `close()`

```
#include <unistd.h>
int close(int fd)   /* fermeture de fichier */
                    /* fd est le descripteur de fichier */
```

Valeur retournée : 0 ou -1 en cas d'échec.

Cette primitive ne vide pas le tampon du noyau correspondant au processus, elle libère simplement le descripteur de fichier pour une éventuelle réutilisation.

Primitive `dup()`

```
#include <unistd.h>
int dup(int fd)  /* fd est le descripteur donné */
```

Cette primitive duplique un descripteur de fichier existant et fournit donc un descripteur ayant exactement les mêmes caractéristiques que celui passé en argument. Elle garantit que la valeur retournée est la plus petite possible parmi les valeurs de descripteurs disponibles.

Valeur retournée : nouveau descripteur de fichier ou -1 en cas d'erreur.

Primitives `dup2()`

```
#include <unistd.h>
int dup2(int fd1,int fd2)     /* force fd2 comme synonyme de fd1 */
                              /* fd1 : descripteur à dupliquer */
                              /* fd2 : nouveau descripteur */
```

Valeur retournée : -1 en cas d'erreur.

Cette primitive oblige le descripteur `fd2` à devenir synonyme du descripteur `fd1`. Notons que `dup2()` ferme le descripteur `fd2` si celui-ci était ouvert.

Primitive `write()`

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t nbytes)
       /* écriture dans un fichier */
       /* fd : descripteur de fichier */
       /* buf : adresse du tampon */
       /* nbytes : nombre d'octets à écrire */
```

Valeur retournée : nombre d'octets écrits ou -1 en cas d'erreur.

Cette primitive écrit dans le fichier ouvert représenté par `fd` les `nbytes` octets sur lesquels pointe `buf`. Il faut noter que l'écriture ne se fait pas directement dans le fichier, mais passe par un tampon du noyau. Attention, ce n'est pas une erreur que d'écrire moins d'octets que souhaités.

Primitive `read()`

```
#include <unistd.h>
ssize_t read(int fd, void  *buf, size_t nbytes)  /* lecture dans un fichier */
                       /* fd : descripteur de fichier */
                       /* buf : adresse du tampon */
                       /* nbytes : nombre d'octets à lire */
```

Valeur retournée : nombre d'octets lus, 0 si la fin de fichier a été atteinte, ou -1 en cas d'erreur.

La primitive lit les `nbytes` octets dans le fichier ouvert représenté par `fd`, et les place dans le tampon sur lequel pointe `buf`.

Exemple 1 : Redirection standard.

Ce programme exécute la commande shell `ps`, puis redirige le résultat vers un fichier `fic_sortie`. Ainsi l'exécution de ce programme ne devrait plus rien donner à l'écran. La primitive `execl()` exécute la commande passée en argument ; nous nous y attarderons dans le chapitre suivant, concernant les processus.

```
                /* fichier test_dup2.c */

#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>

#define STDOUT 1
```

```
main()
{
  int fd ;
  /* affecte au fichier fic_sortie le descripteur fd */
  if ((fd = open("fic_sortie",O_CREAT | O_WRONLY | O_TRUNC, 0666)) == -1) {
    perror("Erreur sur l'ouverture de fic_sortie") ;
    exit(1) ;
  }

  dup2(fd,STDOUT) ;            /* duplique la sortie standard */
  execl("/bin/ps","ps",NULL) ; /* exécute la commande */
}
```

Résultat de l'exécution :

```
Systeme> test_dup2
Systeme> more fic_sortie
   PID TTY      TIME COMMAND
  3954 ttyp3    0:03 csh
```

Notons que les autres redirections suivent le même principe, et qu'il est possible de coupler les redirections d'entrée et de sortie.

Exemple 2 (en exercice) : programme réalisant la copie d'un fichier vers un autre.

3.2 La manipulation des répertoires

Un répertoire est un fichier contenant un ensemble de couples `(nom_de_fichier, numéro_d'inode)`. Il est possible d'ouvrir un répertoire pour le parcourir. C'est ce que fait la commande `ls`. Unix fournit plusieurs fonctions pour parcourir un répertoire. Rappelons que Unix hiérarchise l'ensemble des répertoires comme un arbre dont la racine s'appelle « `/` ». Par convention, le répertoire courant s'appelle « `.` » et son père dans la hiérarchie s'appelle « `..` ».

3.2.1 Les fonctions `opendir()` et `closedir()`

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

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

La fonction `opendir()` ouvre le répertoire `name` et retourne un pointeur vers la structure `DIR` qui joue le même rôle que `FILE`. La valeur `NULL` est renvoyée si l'ouverture n'est pas possible.

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

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

La fonction `closedir()` ferme un répertoire ouvert par `opendir` !

3.2.2 La fonction `readdir()`

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

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

La fonction `readdir()` retourne un pointeur vers une structure de type `struct dirent` représentant le prochain couple (*nom_de_fichier*, *inode*) dans le répertoire. La structure `dirent` contient deux champs `d_name` et `d_ino` qui représentent respectivement le nom du fichier et son numéro d'inode. Les données retournées par `readdir()` sont écrasées à chaque appel. Quand la fin du répertoire est atteinte, `readdir()` renvoie `NULL`.

3.2.3 Exercice : la lecture d'un répertoire

Écrire un programme qui ouvre le répertoire courant et imprime tous les fichiers ainsi que leurs numéros d'inodes. Ce programme doit réaliser la même sortie que la commande `ls -ai`.

3.3 Les verrous POSIX : définitions et utilité

Les verrous servent à empêcher que plusieurs processus accèdent simultanément aux mêmes enregistrements.

Considérons le programme suivant qui écrit dans un fichier 10 fois son *pid* ainsi que l'heure d'écriture.

```
/*
** fichier writel.c
*/
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

int main(int argc, char **argv) /* argv[1] = fichier à écrire */
{
  int desc ;
```

```
  int i ;
  char buf[1024] ;
  int n ;
  time_t secnow ;

  if (argc < 2) {
    fprintf(stderr,"Usage : %s filename \n", argv[0]) ;
    exit(1) ;
  }
  if ((desc = open(argv[1], O_WRONLY | O_CREAT | O_APPEND, 0666)) == -1) {
    perror("Ouverture impossible ");
    exit(1);
  }

#ifdef VERROU  /* on verrouille tout le fichier */
  if (lockf(desc, F_LOCK, 0) == -1) {
    perror("lock") ;
    exit(1) ;
  }
  else
    printf("processus %ld : verrou posé  \n", (long int)getpid()) ;
#endif

  for (i =0 ; i< 10 ; i++) {
    time(&secnow) ;
    sprintf(buf,"%ld : écriture à  %s ", (long int)getpid(),
      ctime(&secnow))  ;
    n =  write (desc, buf, strlen(buf)) ;
    sleep(1) ;
  }

#ifdef VERROU  /* levée du verrou */
  lockf(desc, F_ULOCK, 0) ;
#endif

  return 0 ;
}
```

On compile le programme sans définir la constante `VERROU`, donc sans verrou implémenté par la fonction `lockf` que nous verrons plus loin.

```
Systeme> gcc -Wall writel.c
Systeme> ./a.out essai & ./a.out essai & ./a.out essai & ./a.out essai &
 [1] 1959
 [2] 1960
 [3] 1961
 [4] 1962
Systeme> cat essai
```

```
1959 : écriture à  Sat Nov 29 18:43:59 1997
1960 : écriture à  Sat Nov 29 18:43:59 1997
1961 : écriture à  Sat Nov 29 18:43:59 1997
1962 : écriture à  Sat Nov 29 18:43:59 1997
1959 : écriture à  Sat Nov 29 18:44:00 1997
1960 : écriture à  Sat Nov 29 18:44:00 1997
1961 : écriture à  Sat Nov 29 18:44:00 1997
...
1962 : écriture à  Sat Nov 29 18:44:08 1997
```

Toutes les écritures sont mélangées. Voyons comment sérialiser les accès.

3.4 La primitive `lockf`

```
#include <unistd.h>
int lockf(int desc, int op, off_t taille)
```

Le paramètre `desc` est un descripteur de fichier ouvert, soit en écriture soit en lecture/écriture.

Les différentes valeurs possibles pour le paramètre `op` sont définies dans les fichiers `unistd.h` ou `lockf.h`.

`F_ULOCK` levée de verrou

`F_LOCK` verrouillage et accès exclusif, mode bloquant

`F_TLOCK` verrouillage et accès exclusif, mode non bloquant

`F_TEST` test d'existence de verrous.

Le paramètre `taille` permet de spécifier la portée du verrouillage. Cette portée s'exprime par rapport à la position du pointeur de fichier ; on peut verrouiller une zone précédente en donnant une valeur négative. Une valeur nulle (comme dans l'exemple) permet de verrouiller jusqu'à la fin du fichier.

L'opération `F_LOCK` est bloquante ; le programme qui l'exécute attend derrière le verrou comme derrière un sémaphore.

L'opération `F_TLOCK` n'est pas bloquante ; si le verrou existe, `lockf` renvoie -1 et `errno` est mis à `EACCES` si un verrou existe déjà.

L'opération `F_ULOCK` déverrouille une zone ; il est possible de ne déverrouiller qu'une partie d'une zone verrouillée.

3.5 Exemple

On compile notre fichier `writel.c` en définissant la constante `VERROU` dans la ligne de commande.

```
Systeme> gcc -Wall -DVERROU writel.c
```

```
Systeme> ./a.out essai & ./a.out essai & ./a.out essai & ./a.out essai &
[1] 2534
[2] 2535
[3] 2536
[4] 2537
Systeme> processus 2534 : verrou posé
processus 2536 : verrou posé
processus 2535 : verrou posé
processus 2537 : verrou posé

[4]    Done                          ./a.out essai
[3]  + Done                          ./a.out essai
[2]  + Done                          ./a.out essai
[1]  + Done                          ./a.out essai
Systeme>
```

Et si on imprime `essai`, on voit que chaque processus a pu écrire sans être interrompu.

```
Systeme> cat essai
 2534 : écriture à  Sat Nov 29 19:23:39 1997
 ...
 2534 : écriture à  Sat Nov 29 19:23:48 1997
 2536 : écriture à  Sat Nov 29 19:23:49 1997
 ..
 2536 : écriture à  Sat Nov 29 19:23:58 1997
 2535 : écriture à  Sat Nov 29 19:23:59 1997
 ..
 2535 : écriture à  Sat Nov 29 19:24:08 1997
 2537 : écriture à  Sat Nov 29 19:24:09 1997
 ..
 2537 : écriture à  Sat Nov 29 19:24:19 1997
 Systeme>
```

3.5.1 Remarque

L'ordre d'appropriation du fichier par les processus est quelconque : le processus 2536 est passé avant 2535.

3.6 Exercices sur le SGF Unix

3.6.1 Ex. 1 : to_exe (simple)

Écrire en `C` le programme `to_exe` qui lorsqu'il est appliqué à *fich* écrit en clair si le fichier *fich* peut être exécuté par son propriétaire.

3.6.2 Ex. 2 : facile

À l'aide des fonctions `opendir`, `readdir` et `stat`, écrire un programme qui imprime un répertoire en précisant la taille de chaque fichier régulier (utilisez la macro `S_ISREG()`). La taille totale de ceux-ci sera donnée en fin d'exécution.

3.6.3 Ex. 3 : `lwd` (facile)

Écrire la commande `lwd` (comme *list working directory* qui donne la dernière composante du répertoire courant.

3.6.4 Ex. 4 : plus dur et plus long

Écrire la commande `mon_pwd` qui donne le répertoire courant à partir de sa racine. Que se passe t-il si le volume est monté ?

3.7 Exercices sur les caractéristiques des utilisateurs

3.7.1 Ex. 1

À l'aide des fonctions `getpwuid`, `getgrgid`, des structures `passwd` et `group`, écrire en `C` un programme qui donne le nom d'un utilisateur en clair quand on lui fournit son `uid`. Si l'`uid` n'est pas attribué, ce programme écrira :
il n'y a pas d'utilisateur associé à cet ***`uid`****.*
Utilisez aussi la fonction `atoi` pour convertir la chaîne de caractères `argv[1]` en un entier.

3.7.2 Ex. 2

Utilisez l'exercice 1 pour écrire en `C` le programme qui imprime les fichiers réguliers d'un répertoire passé en paramètre, en précisant pour chaque fichier quel est son propriétaire (nom en clair et `uid`) et son groupe.

3.7.3 Ex. 3

Écrire en `C` un programme `sete` qui crée un fichier dans le répertoire courant et y range le nom de login, les numéros d'`uid` réels et effectifs. Utilisez les primitives `getlogin`, `getuid` et `geteuid` pour obtenir ces renseignements. Pour convertir des variables numériques en une chaîne de caractères formatée, utilisez la fonction `C sprintf`. Utilisez `open`, `write` et `close` pour accéder au fichier. Lancez ce programme dans votre répertoire et dans le répertoire d'un autre utilisateur. Dans le deuxième cas, vous ne devez pas avoir le droit d'écrire dans le répertoire. Votre programme doit détecter cet empêchement (utiliser `perror` pour imprimer le bon message d'erreur).

Copiez `sete` dans `/tmp`. Positionnez le bit *s* par la commande `chmod u+s sete` et faites exécuter ce programme par votre voisin. Vous devez voir l'utilisateur effectif différer de l'utilisateur réel. Votre voisin pourra écrire dans votre répertoire malgré les protections que vous avez placées.

Chapitre 4

Les processus

4.1 Définition

Un processus est un programme qui s'exécute. Un processus possède un identificateur qui est un numéro : le *pid*. Il s'agit d'un entier du type `pid_t` déclaré comme synonyme du type `int` ou `unsigned long int`. Un processus incarne — exécute — un programme ; il appartient à un utilisateur ou au noyau. Un processus possède une priorité et un mode d'exécution. Nous distinguerons deux modes : le mode utilisateur (ou esclave) de basse priorité et le mode noyau (ou maître) de plus forte priorité. Généralement un processus appartient à la personne qui a écrit le programme qui s'exécute, mais ce n'est pas toujours le cas. Quand vous exécutez la commande `ls` vous exécutez le programme `/bin/ls` qui appartient à `root`. Regardez le résultat de la commande `ls -l` !

```
% ls -l /bin/ls
-rwxr-xr-x   1 root     root        29980 Apr 24  1998 /bin/ls
%
```

Un processus naît, crée d'autres processus (ses fils), attend la fin d'exécution de ceux-ci ou entre en compétition avec eux pour avoir des ressources du système et enfin meurt. Pendant sa période de vie, il accède à des variables en mémoire suivant son mode d'exécution. En mode noyau, tout l'espace adressable lui est ouvert ; en mode utilisateur, il ne peut accéder qu'à ses données privées.

4.2 Les identificateurs de processus

Chaque processus possède un identificateur unique nommé `pid`. Comme pour les utilisateurs, il peut être lié à un groupe ; on utilisera alors l'identificateur `pgrp`. Citons les différentes primitives permettant de connaître ces différents identificateurs :

```
pid_t getpid()      /* retourne l'identificateur du processus */

pid_t getpgrp()     /* retourne l'identificateur du groupe de processus */
```

```
pid_t getppid()     /* retourne l'identificateur du père du processus */

pid_t setpgrp()     /* positionne l'identificateur du groupe de   */
                    /* processus à la valeur de son pid : crée un */
                    /* nouveau groupe de processus                */
```

Valeur retournée : nouvel identificateur du groupe de processus.

Exemple :

```
                /* fichier test_idf.c */
#include <stdio.h>

main()
{
   printf("je suis le processus %d de père %d et de groupe %d\n",getpid(),
             getppid(),getpgrp()) ;
}
```

Résultat de l'exécution :

```
Systeme> ps
   PID TTY      TIME COMMAND
  6658 ttyp5    0:04 csh
Systeme> test_idf
je suis le processus 8262 de père 6658 et de groupe 8262
```

Notons que le père du processus exécutant `test_idf` est `csh`.

4.3 L'utilisateur réel et l'utilisateur effectif

Vous pouvez utiliser le programme `/usr/bin/passwd` qui appartient à `root` et bénéficier des mêmes droits que `root` afin de changer votre mot de passe en écrivant dans le fichier des mots de passe. Unix distingue deux utilisateurs de processus : l'utilisateur réel et l'utilisateur effectif. Quand vous modifiez votre mot de passe, vous restez l'utilisateur réel de la commande `passwd`, mais l'utilisateur effectif est devenu `root`. Un bit particulier du fichier exécutable permet d'effectuer ce changement temporaire d'identité : c'est le bit `s`.

4.3.1 Les primitives `setuid` et `seteuid`

`getuid` et `geteuid` donnent l'identité de l'utilisateur.

```
#include <unistd.h>
uid_t getuid(void);
uid_t geteuid(void);
```

`getuid` donne le numéro de l'usager réel à qui appartient le processus effectuant l'appel ; `geteuid` donne le numéro de l'usager effectif associé processus. Ce numéro peut être différent si le bit `s` du fichier est positionné. Le type `uid_t` est généralement un `int`.

Remarque : la commande `chmod u+s fichierExecutable` positionne ce bit sur le fichier exécutable `fichierExecutable`.

4.3.2 La primitive `getlogin`

```
#include <unistd.h>

char * getlogin ( void );
```

Cette commande donne le nom – sous forme d'une suite de caractères – de l'utilisateur connecté au système sur un terminal de contrôle. Chaque appel détruit l'ancien nom.

4.3.3 Les commandes `getpwnam` et `getpwuid`

Ces deux commandes explorent le fichier des mots de passe à la recherche de la ligne qui décrit l'utilisateur passé en paramètre.

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

struct passwd *getpwnam(const char * name);

struct passwd *getpwuid(uid_t uid);
```

La structure `passwd` comporte entre autres les champs suivants :

```
struct passwd {
  char    *pw_name;       /* nom de login de l'utilisateur          */
  char    *pw_passwd;     /* son mot de passe crypté                */
  uid_t   pw_uid;         /* numéro de l'utilisateur                */
  gid_t   pw_gid;         /* numéro du groupe                       */
  char    *pw_gecos;      /* nom réel                               */
  char    *pw_dir;        /* répertoire par défaut (home directory) */
  char    *pw_shell;      /* interpréteur de commandes              */
};
```

La commande `getpwuid(geteuid())` permet d'obtenir les caractéristiques de l'utilisateur effectif.

4.4 Les principales primitives de gestion de processus

4.4.1 Primitive `fork()`

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

pid_t fork()          /* création d'un fils */
```

Valeur retournée : 0 pour le processus fils, et l'identificateur du processus fils pour le père, -1 dans le cas d'épuisement de ressource.

Cette primitive est l'unique appel système permettant de créer un processus. Les processus père et fils partagent le même code. Le segment de données de l'utilisateur du nouveau processus est une copie exacte du segment correspondant de l'ancien processus, alors que la copie du segment de données système diffère par certains attributs (par exemple le *pid*, le temps d'exécution). Le fils hérite d'un double de tous les descripteurs de fichiers ouverts du père (si le fils en ferme un, la copie du père n'est pas changée) ; par contre les pointeurs de fichier associés sont partagés (si le fils se déplace dans le fichier, la prochaine manipulation du père se fera à la nouvelle adresse). Cette notion est importante pour implémenter des tubes.

Exemple introductif

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void main()
{
  pid_t p1 ;
  printf("Début de fork\n") ;
  p1 = fork() ;
  printf("Fin de fork %d\n", p1) ;
}
```

Résultat de l'exécution :

```
xures sources 174 % ./a.out
Début de fork
Fin de fork 16099
xures sources 175 % Fin de fork 0

xures sources 175 %
```

Dans cet exemple, on voit qu'un seul processus exécute l'écriture `Début de fork`, par contre, on voit deux écritures `Fin de fork` suivies de la valeur de retour de la primitive `fork()`. Il y a bien apparition d'un processus : le fils, qui ne débute pas son exécution au début du programme mais à partir de la primitive `fork`.

Exercice

On considère le programme suivant :

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void main()
{
  pid_t p1, p2 ;
  p1 = fork() ;
  p2 = fork() ;
  pause() ;
}
```

Combien de processus sont lancés par la commande `a.out` ? À partir du résultat de la commande `ps -l`, donnez la généalogie de cette famille. Donnez pour chaque processus, les valeurs des variables `p1` et `p2`.

Autre exercice

Voici un exercice donné au DS 1999 :

Écrire un bout de code qui crée 15 processus avec les contraintes suivantes :

- le processus initial est l'un des 15 ;
- un processus donné a ou bien 0, ou bien 2 ou bien 3 fils (mais jamais 1 seul) ;
- tous les fils du processus initial ont eux-même des fils ;
- le code est le plus simple possible et contient le plus petit nombre de `fork()` possibles.

Représentez votre arborescence de manière graphique et justifiez clairement que vous obtenez exactement 15 processus.

Exercice pour le DS 2000 !

L'une des réponses à l'exercice précédent était :

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

main() {
  int i,j,k;
  printf("je suis un processus\n");fflush(stdout);
  for (i=1;i<3;i++) {
    if (fork()==0) {
```

```
      printf("Je suis un processus\n");fflush(stdout);
      sleep(1);
      for (j=1;j<3;j++) {
        if (fork()==0) {
          printf("Je suis un processus\n");fflush(stdout);
          sleep(1);
          for (k=1;k<3;k++) {
            if (fork()==0) {
              printf("Je suis un processus\n");fflush(stdout);
            }
            sleep(1);
          }
        }
      }
    }
  }
}
```

La question se pose alors de savoir combien ce programme produit de processus. Une question similaire (avec un programme différent, évidemment !) pourra être posée au DS 2000.

4.4.2 Primitive `wait()`

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

int wait(int *status)
```

Valeur retournée : identificateur du processus mort ou -1 en cas d'erreur.

Le code retourné via `status` indique la raison de la mort du processus, qui est :

- soit l'appel de `exit()`, et dans ce cas, l'octet de droite de `status` vaut 0, et l'octet de gauche est le paramètre passé à `exit` par le fils ;
- soit la réception d'un signal fatal, et dans ce cas, l'octet de droite est non nul. Les 7 premiers bits de cet octet contiennent le numéro du signal qui a tué le fils. De plus, si le bit de gauche de cet octet est 1, un fichier image a été produit (qui correspond au fichier core du répertoire courant). Ce fichier image est une copie de l'espace mémoire du processus, et est utilisé lors du débogage.

```
/* fichier test_wait.c */
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <sys/wait.h>
int main()
{
```

```
int pid ;
printf("Bonjour, je me présente, je suis %d.\n",getpid()) ;
if (fork() == 0) {
  printf("\tSalut, je suis %d, le gamin de %d.\n",getpid(),getppid()) ;
  sleep(3) ;
  printf("\t exit(7)\n") ;
  exit(7) ;
}
else {
  int ret1, status1 ;
  printf("Attente de l'exit(7)\n") ;
  ret1 = wait(&status1) ;
  if ((status1&255) == 0) {
    printf("Valeur de retour de wait(): %d\n",ret1) ;
    printf("Paramètre de exit(): %d\n",(status1>>8)) ;
  }
  else
    printf("Le fils n'est pas mort par exit.\n") ;
  printf("\nC'est toujours moi, %d, et je sens que ça recommence.\n",
  getpid()) ;
  if ((pid=fork()) == 0) {
    printf("\tHello, je suis %d, le deuxième fils de %d\n",
    getpid(),getppid()) ;
    sleep(3) ;
    printf("\tje boucle à l'infini!\n") ;
    for(;;) ;
  }
  else {
    int ret2, status2, s ;
    printf("Attente\n") ;
    ret2 = wait(&status2) ;
    if ((status2&255) == 0) {
      printf("Le fils %d s'achève par un exit\n",ret2) ;
    }
    else {
      printf("Valeur de retour de wait() : %d\n",ret2) ;
      s = status2&255 ;
      if (s>127) {
        printf("Le signal fatal est : %d\n",s-128) ;
        printf("Il a créé un fichier 'core'\n") ;
      }
      else {
        printf("Le signal fatal est : %d.\n",s) ;
        printf("Il n'a pas créé de fichier 'core'\n") ;
      }
    }
  }
}
```

```
  return 0 ; /* pour exit(0) */
}
```

Résultat de l'exécution :

On lance le programme en arrière-plan, et lorsque le deuxième fils boucle à l'infini, on lui envoie un signal par la commande *shell* `kill -numero_signal pid_fils2`.

```
bolz> test_wait&
[1]     10299
Bonjour, je me présente, je suis 10299.
        Salut, je suis 10300, le gamin de 10299.
        exit(7)
Valeur de retour de wait(): 10300
Valeur de retour de exit(): 7

C'est toujours moi,10299, et je sens que ça recommence.
        Hello, je suis 10301, le deuxième fils de 10299
Attente
        Et pour cela, je boucle à l'infini!
kill -8 10301
bolz> Valeur de retour de wait(): 10301
Le signal fatal est : 8.
Il n'a pas créé de fichier 'core'

[1] +  Done (29)              test_wait&
```

Commentaires :

Après la création de chaque fils, le père se met en attente de la mort de celui-ci. Le premier fils meurt par un appel à `exit()`, le paramètre de `wait()` contient, dans son octet de gauche, le paramètre passé à `exit()` (ici 7). Le deuxième fils meurt à la réception d'un signal, le paramètre de `wait()` contient, dans ses sept premiers bits, le numéro du signal (ici 8), et le huitième bit indique si un fichier `core` est créé.

Remarque concernant les processus *zombis* Un processus peut se terminer quand son père n'est pas en train de l'attendre. Un tel processus devient un processus *zombi*. Il est alors identifié par le nom `<defunct>`. Ses segments d'instructions, de données de l'utilisateur et de données système sont supprimés, mais il continue d'occuper une place dans la table des processus du noyau. Lorsqu'il est attendu à son tour, il disparaît.

```
                /* fichier test_defunct.c */

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

```
#include <signal.h>
#include <sys/wait.h>
int main() {

  int pid ;
  printf("je suis %d et je vais créer un fils\n",getpid());
  printf("je bouclerai ensuite à l'infini\n") ;
  pid = fork() ;
  if(pid == -1) {
    /* erreur */
    perror("impossible de créer un fils") ;
    exit(-1) ;
  }
  else
    if (pid == 0) { /* fils */
      printf("je suis %d le fils\n",getpid());
      sleep(10) ;
      printf(" exit(0) ;\n") ;
      exit(0) ;
    }
    else { /* père */
      for(;;) ; /* le père boucle à l'infini */
    }
}
```

Résultat de l'exécution :

On lance le programme `test_defunct` en arrière-plan.

```
je suis 28339 et je vais créer un fils
je bouclerai ensuite à l'infini
je suis 28340 le fils
 exit(0) ;
```

Le processus 28339 crée un fils 28340, on effectue tout de suite dans le *shell* un `ps`. On obtient :

```
  PID TTY      TIME COMMAND
28339 ttyp9    0:08 test_defunct
28340 ttyp9    0:00 test_defunct
28341 ttyp9    0:00 ps
```

On refait un `ps` dès que le fils a annoncé sa mort, et on obtient :

```
  PID TTY      TIME COMMAND
28340 ttyp9    0:00 <defunct>
28339 ttyp9    0:16 test_defunct
28341 ttyp9    0:00 ps
```

4.4.3 Primitive `exit()`

```
void exit(int status)  /* terminaison du processus */
                       /* status : état de sortie */
```

Valeur retournée : c'est la seule primitive qui ne retourne jamais.

Tous les descripteurs de fichier ouverts sont fermés. Si le père meurt avant ses fils, le père du processus fils devient le processus `init` de *pid* 1.

Par convention, un code de retour égal à zéro signifie que le processus s'est terminé normalement, et un code non nul (généralement 1 ou -1) signifie qu'une erreur s'est produite.

4.5 Les primitives `exec()`

Il s'agit d'une famille de primitives permettant le lancement de l'exécution d'un programme externe. Il n'y a pas création d'un nouveau processus, mais simplement changement de programme. Il y a six primitives `exec()` que l'on peut répartir dans deux groupes : les `execl()`, pour lesquels le nombre des arguments du programme lancé est connu, puis les `execv()` où il ne l'est pas. En outre toutes ces primitives se distinguent par le type et le nombre de paramètres passés.

Premier groupe d'`exec()`. Les arguments sont passés sous forme de liste :

```
int execl(char *path, char *arg0, char *arg1,..., char *argn,NULL)
          /* exécute un programme */
          /* path : chemin du fichier programme */
          /* arg0 : premier argument */
          /* ... */
          /* argn : (n+1)ième argument */

int execle(char *path,char *arg0,char *arg1,...,char *argn,NULL,char *envp[])
          /* envp : pointeur sur l'environnement */

int execlp(char *file,char *arg0,char *arg1,...,char *argn,NULL)
```

Dans `execl` et `execle`, `path` est une chaîne indiquant le chemin exact d'un programme. Un exemple est `"/bin/ls"`. Dans `execlp`, le « p » correspond à `path` et signifie que les chemins de recherche de l'environnement sont utilisés. Par conséquent, il n'est plus nécessaire d'indiquer le chemin complet. Le premier paramètre de `execlp` pourra par exemple être `"ls"`.

Le second paramètre des trois fonctions `exec` est le nom de la commande lancée et reprend donc une partie du premier paramètre. Si le premier paramètre est `"/bin/ls"`, le second doit être `"ls"`. Pour la troisième commande, le second paramètre est en général identique au premier si aucun chemin explicite n'a été donné.

Second groupe d'`exec()`. Les arguments sont passés sous forme de tableau :

```
int execv(char  *path,char  *argv[])
```

```
            /* argv : pointeur vers le tableau contenant les arguments */

int execve(char  *path,char  *argv[],char *envp[])

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

`path` et `file` ont la même signification que dans le premier groupe de commandes. Les autres paramètres du premier groupe de commandes sont regroupés dans des tableaux dans le second groupe. La dernière case du tableau doit être un pointeur nul, car cela permet d'identifier le nombre d'éléments utiles dans le tableau.

Pour `execle` et `execve`, le dernier paramètre correspond à un tableau de chaînes de caractères, chacune correspondant à une variable d'environnement. On trouvera plus de détails dans le manuel en ligne.

Il est important de noter que les caractères que le *shell* expanse (comme ~) ne sont pas interprétés ici.

4.5.1 Recouvrement

Lors de l'appel d'une primitive `exec()`, il y a recouvrement du segment d'instructions du processus appelant, ce qui implique qu'il n'y a pas de retour d'un `exec()` réussi (l'adresse de retour a disparu). Le code du processus appelant est détruit.

```
                /* fichier test_exec.c */

#include <stdio.h>

main()
{
  execl("/bin/ls","ls",NULL) ;
  printf ("je ne suis pas mort\n") ;
}
```

Résultat de l'exécution :

```
Systeme> test_exec
fichier1
fichier2
test_exec
test_exec.c
Systeme>
```

On note que la commande `ls` est réalisée, contrairement à l'appel à `printf()`, ce qui montre que le processus ne retourne pas après `exec()`. D'où l'intérêt de l'utilisation de la primitive `fork()` :

```
                /* fichier test_exec_fork.c */
```

```
#include <stdio.h>

main() {

  if ( fork()==0 ) execl( "/bin/ls","ls",NULL) ;
  else {
    sleep(2) ;  /* attend la fin de ls pour exécuter printf() */
    printf ("je suis le père et je peux continuer") ;
  }
}
```

Résultat de l'exécution :

```
Systeme> test_exec_fork
fichier1
fichier2
test_exec
test_exec.c
test_exec_fork
test_exec_fork.c
je suis le père et je peux continuer
Systeme>
```

Dans ce cas, le fils meurt après l'exécution de `ls`, et le père continue à vivre et exécute `printf()`.

4.5.2 Comportement vis-à-vis des fichiers ouverts

Les descripteurs de fichiers ouverts avant l'appel d'un `exec()` le restent, sauf demande contraire (par la primitive `fcntl()`). L'un des effets du recouvrement est l'écrasement du tampon associé au fichier dans la zone utilisateur, et donc la perte des informations qu'elle contenait. Il est donc nécessaire de forcer avant l'appel à `exec()` le vidage de ce tampon au moyen de la fonction `fflush()`.

4.5.3 Remarques

`stdout` est défini dans `stdio.h` et correspond à la sortie écran. La commande `cd` si elle était exécutée par un processus fils, serait sans effet parce qu'un attribut changé par un processus fils (ici le répertoire courant) n'est pas remonté au père. Pour suivre un chemin relatif (commençant sur le répertoire courant) le noyau doit savoir où commencer. Pour cela, il conserve tout simplement pour chaque processus le numéro d'index du répertoire courant.

Ce programme utilise la primitive `chdir()`. Cette primitive est utilisée essentiellement dans l'implémentation de la commande *shell* `cd`. Elle change le répertoire courant du processus qui l'exécute.

```
int chdir(char *path)    /* change le répertoire courant */
              /* path : chaîne spécifiant le nouveau répertoire */
```

```
                /* fichier test_cd.c */

/* le changement de répertoire n'est valable */
/* que le temps de l'exécution du processus */

#include <stdio.h>

main()
{
  chdir("..") ; /* on va au répertoire précédent */

     /* on exécute un pwd qui va tuer le processus et  */
     /* renvoyer le répertoire dans lequel il se trouve */
  if (execl("/bin/pwd","pwd",NULL)==-1) {
    perror("impossible d'exécuter pwd") ;
    exit(-1) ;
  }
}
```

Résultat de l'exécution :

On utilise, une première fois, la commande *shell* `pwd`. On obtient :

```
/users/ens/bernard/Documentation/Exemples/Processus
```

On lance maintenant `test_cd` :

```
/users/ens/bernard/Documentation/Exemples
```

On relance `pwd` :

```
/users/ens/bernard/Documentation/Exemples/Processus
```

Comme le montre cet exemple, le processus créé par le *shell* pour l'exécution du programme `test_cd` a hérité du répertoire courant. L'exécution de `chdir("..")` a permis effectivement de remonter dans l'arborescence. Cependant, ce changement n'est valable que pour le processus effectuant `chdir()`. Aussi, à la mort du processus, on peut constater que le répertoire dans lequel on se trouve est le même qu'au départ.

4.5.4 Exercices : écriture d'une version simplifiée des commandes `if` et `for`

La commande `if`

1. **Syntaxe :**
   `if` condition `then` commande_alors `else` commande_sinon
2. **Exemple**
   `if test -f version.sty then rm version.sty else echo fichier absent`
   Si le fichier `version.sty` est trouvé, on le supprime, sinon on imprime un message d'erreur. On considère que la condition est vérifiée si la commande testée renvoie 0.

3. **Analyse syntaxique**
   On utilise le découpage en mots fournis par le *shell* ; on retrouve dans le tableau `argv[]` du programme principal la liste des mots de la commande. On doit obligatoirement avoir les chaînes `if`, `then` et `else`.

La commande `for`

Écrire une version simplifiée de la commande `for` qui répète une commande UNIX pour chaque argument.
Usage : `monfor` ⟨*nomfich*⟩ `[`⟨*nomfich*⟩`*]` *commandeUNIX*

Troisième partie

Communication

Chapitre 5

Communication inter-processus

5.1 Les signaux

5.1.1 Introduction

Un signal est une interruption logicielle qui est envoyée aux processus par le système pour les informer sur des événements anormaux se déroulant dans leur environnement (violation de mémoire, erreur dans les entrées/sorties). Il permet également aux processus de communiquer entre eux. À une exception près (`SIGKILL`), un signal peut être traité de trois manières différentes :

- Il peut être ignoré. Par exemple, le programme peut ignorer les interruptions clavier générées par l'utilisateur (c'est ce qui se passe lorsqu'un processus est lancé en arrière-plan).
- Il peut être pris en compte. Dans ce cas, à la réception d'un signal, l'exécution d'un processus est détournée vers une procédure spécifiée par l'utilisateur, puis reprend où elle a été interrompue.
- Son comportement par défaut peut être restitué par un processus après réception de ce signal.

5.1.2 Types de signaux

Les signaux sont identifiés par le système par un nombre entier. Le fichier `/usr/include/signal.h` contient la liste des signaux accessibles. Chaque signal est caractérisé par un mnémonique. La liste des signaux usuels est donnée ci-dessous :

- `SIGHUP (1)` Coupure : signal émis aux processus associés à un terminal lorsque celui-ci se déconnecte. Il est aussi émis à chaque processus d'un groupe dont le chef se termine.
- `SIGINT (2)` Interruption : signal émis aux processus du terminal lorsqu'on frappe la touche d'interruption (`INTR` ou `CTRL-C`) de son clavier.
- `SIGQUIT (3)`* Abandon : idem avec la touche d'abandon (`QUIT` ou `CTRL-D`).
- `SIGILL (4)`* Instruction illégale : signal émis à la détection d'une instruction illégale, au niveau matériel (exemple : lorsqu'un processus exécute une instruction flottante alors que l'ordinateur ne possède pas d'instructions flottantes câblées).

- `SIGTRAP (5)`* Piège de traçage : signal émis après chaque instruction en cas de traçage de processus (utilisation de la primitive `ptrace()`).
- `SIGIOT (6)`* Piège d'instruction d'E/S : signal émis en cas de problème matériel.
- `SIGEMT (7)` Piège d'instruction émulateur : signal émis en cas d'erreur matérielle dépendant de l'implémentation.
- `SIGFPE (8)`* signal émis en cas d'erreur de calcul flottant, comme un nombre en virgule flottante de format illégal. Indique presque toujours une erreur de programmation.
- `SIGKILL (9)` Destruction : arme absolue pour tuer les processus. Ne peut être ni ignoré, ni intercepté. (voir `SIGTERM` pour une mort plus douce).
- `SIGBUS (10)`* signal émis en cas d'erreur sur le bus.
- `SIGSEGV (11)`* signal émis en cas de violation de la segmentation : tentative d'accès à une donnée en dehors du domaine d'adressage du processus actif.
- `SIGSYS (12)`* Argument incorrect d'un appel système.
- `SIGPIPE (13)` Écriture sur un conduit non ouvert en lecture.
- `SIGALRM (14)` Horloge : signal émis quand l'horloge d'un processus s'arrête. L'horloge est mise en marche par la primitive `alarm()`.
- `SIGTERM (15)` Terminaison logicielle : signal émis lors de la terminaison normale d'un processus. Il est également utilisé lors d'un arrêt du système pour mettre fin à tous les processus actifs.
- `SIGUSR1 (16)` Premier signal à la disposition de l'utilisateur : utilisé pour la communication inter-processus.
- `SIGUSR2 (17)` Deuxième signal à la disposition de l'utilisateur : idem `SIGUSR1`.
- `SIGCLD (18)` Mort d'un fils : signal envoyé au père à la terminaison d'un processus fils.
- `SIGPWR (19)` Réactivation sur panne d'alimentation.

Remarque :

Les signaux repérés par « * » génèrent un fichier `core` sur le disque, lorsqu'ils ne sont pas traités correctement.

Pour plus de portabilité, on peut écrire des programmes utilisant des signaux en appliquant les règles suivantes : éviter les signaux `SIGIOT`, `SIGEMT`, `SIGBUS` et `SIGSEGV` qui dépendent de l'implémentation. Il est correct de les intercepter pour imprimer un message, mais il ne faut pas essayer de leur attribuer une quelconque signification.

5.1.3 Traitement des signaux

A : Réception d'un signal

Primitive `signal()`

```
#include <signal.h>

void (*signal (int sig, void (*fcn)(int)))(int) /* réception d'un signal */
```

```
/* sig : numéro du signal */
/* (*fcn) : action après réception */
```

Valeur retournée : adresse de la fonction spécifiant le comportement du processus vis-à-vis du signal considéré, -1 sinon.

Cette primitive intercepte le signal de numéro `sig`. Le second argument est un pointeur sur une fonction qui peut prendre une des trois valeurs suivantes :

1. `SIGDFL` : ceci choisit l'action par défaut pour le signal. La réception d'un signal par un processus entraîne alors la terminaison de ce processus, sauf pour `SIGCLD` et `SIGPWR`, qui sont ignorés par défaut. Dans le cas de certains signaux, il y a création d'un fichier image `core` sur le disque.
2. `SIGIGN` : ceci indique que le signal doit être ignoré : le processus est immunisé. On rappelle que le signal `SIGKILL` ne peut être ignoré.
3. Un pointeur sur une fonction (nom de la fonction) : ceci implique la capture du signal. La fonction est appelée quand le signal arrive, et après son exécution, le traitement du processus reprend où il a été interrompu. On ne peut procéder à un déroutement sur la réception d'un signal `SIGKILL` puisque ce signal n'est pas interceptable.

Nous voyons donc qu'il est possible de modifier le comportement d'un processus à l'arrivée d'un signal donné. C'est ce qui se passe pour un certain nombre de processus standard : le *shell*, par exemple, à la réception d'un signal `SIGINT` affiche le prompt (et n'est pas interrompu).

Primitive `pause()`

```
#include <unistd.h>
void pause(void)              /* attente d'un signal quelconque */
```

Cette primitive correspond à de l'attente pure. Elle ne fait rien, et n'attend rien de particulier. Cependant, puisque l'arrivée d'un signal interrompt toute primitive bloquée, on peut tout aussi bien dire que `pause()` attend un signal. On observe alors le comportement de retour classique d'une primitive bloquée, c'est à dire le positionnement de `errno` à `EINTR`. Notons que le plus souvent, le signal que `pause()` attend est l'horloge d'`alarm()`.

Exercice 1 : test de la fonction `pause()`.

Exercice 2 : héritage des signaux par `fork()`. Les processus fils recevant l'image mémoire du père, héritent de son comportement vis-à-vis des signaux.

B : Émission d'un signal

Primitive `kill()`

```
#include <signal.h>

int kill(pid_t pid,int sig)  /* émission d'un signal */
        /* pid : identificateur du processus ou du groupe destinataire */
        /* sig : numéro du signal */
```

Valeur retournée : 0 si le signal a été envoyé, -1 sinon.

Cette primitive émet à destination du processus de numéro `pid` le signal de numéro `sig`. De plus, si l'entier `sig` est nul, aucun signal n'est envoyé, et la valeur de retour permet de savoir si le nombre `pid` est un numéro de processus ou non.

Utilisation du paramètre `pid` :

- Si `pid` > 0 : `pid` désigne alors le processus d'identificateur `pid`.
- Si `pid` $= 0$: le signal est envoyé à tous les processus du même groupe que l'émetteur (cette possibilité est souvent utilisée avec la commande *shell*

  ```
  kill -9 0
  ```

  pour tuer tous les processus en arrière-plan sans avoir à indiquer leurs identificateurs de processus).
- Si `pid` $= -1$:
  si le processus appartient au super-utilisateur, le signal est envoyé à tous les processus, sauf aux processus système et au processus qui envoie le signal. Sinon, le signal est envoyé à tous les processus dont l'identificateur d'utilisateur réel est égal à l'identificateur d'utilisateur effectif du processus qui envoie le signal (c'est un moyen de tuer tous les processus dont on est propriétaire, indépendamment du groupe de processus).
- Si `pid` < -1 : le signal est envoyé à tous les processus dont l'identificateur de groupe de processus (`pgid`) est égal à la valeur absolue de `pid`.

Notons que la primitive `kill()` est le plus souvent exécutée via la commande *shell* `kill`.

Primitive `alarm()`

```
#include <unistd.h>

unsigned alarm(unsigned secs) /* envoi d'un signal SIGALRM */
                              /* secs : nombre de secondes */
```

Valeur retournée : temps restant dans l'horloge.

Cette primitive envoie un signal `SIGALRM` au processus appelant après un laps de temps `secs` (en secondes) passé en argument, puis réinitialise l'horloge d'alarme. À l'appel de la primitive, l'horloge est initialisée à `secs` secondes, et est décrémentée jusqu'à 0. Si le paramètre `secs` est nul, toute requête est annulée. Cette primitive peut être utilisée, par exemple, pour forcer la lecture au clavier dans un délai donné. Le traitement du signal doit être prévu, sinon le processus est tué.

Exemple 1 : fonctionnement de `alarm()`

```
                /* fichier test_alarm.c */

/*
 * test des valeurs de retour de alarm()
```

```
 * ainsi que de son fonctionnement
 */

#include <errno.h>
#include <signal.h>

it_horloge(int sig) /* routine exécutée sur réception de SIGALRM */
{
     printf("réception du signal %d : SIGALRM\n",sig) ;
}

main()
{
  unsigned sec ;
  signal(SIGALRM,it_horloge) ; /* interception du signal */
  printf("on fait alarm(5)\n") ;
  sec=alarm(5) ;
  printf("valeur retournée par alarm : %d\n",sec) ;
  printf("le principal boucle à l'infini (CTRL-c pour arrêter)\n") ;
  for(;;) ;
}
```

Résultat de l'exécution :

```
on fait alarm(5)
valeur retournée par alarm : 0
le principal boucle à l'infini (CTRL-C pour arrêter)
réception du signal 14 : SIGALRM
```

Exemple 2 (exercice) : test avec deux interruptions (`alarm()` et `CTRL-C`)

5.1.4 Communication entre processus

Il s'agit d'un exemple simple utilisant les primitives d'émission et de réception de signaux, afin de faire communiquer deux processus entre eux. En outre, l'exécution de ce programme permet de s'assurer que le processus exécutant la routine de déroutement est bien celui qui reçoit le signal.

```
                /* fichier test_kill_signal.c */
/*
 * communication simple entre deux processus
 *  au moyen des signaux
 */
#include <stdio.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>
```

```
void it_fils()
{
  printf("\tInterruption : kill (getpid(),SIGINT)\n") ;
  kill (getpid(),SIGINT) ;
}

void fils()
{
  signal(SIGUSR1,it_fils) ;
  printf("Looping !\n") ;
  while(1) ;
}

int main()
{
  int pid ;

  if ((pid=fork())==0) fils() ;
  else {
    sleep(3) ;
    printf(" kill (pid,SIGUSR1) ;\n") ;
    kill (pid,SIGUSR1) ;
  }
  return 0 ;
}
```

Résultat de l'exécution :

```
lixieres sources 164 % gcc -Wall test_kill_signal.c
lixieres sources 165 % ./a.out
Looping !
 kill (pid,SIGUSR1) ;
lixieres sources 166 %  Interruption : kill (getpid(),SIGINT)

lixieres sources 166 %
```

Contrôle de l'avancement d'une application

Tous ceux qui ont lancé des programmes de simulation ou de calcul numérique particulièrement longs ont sûrement désiré connaître l'état d'avancement de l'application pendant son exécution. Ceci est parfaitement réalisable grâce à la commande *shell* `kill`, en envoyant au processus concerné un signal ; le processus peut alors à la réception d'un tel signal, afficher les données désirées. Voici un exemple qui permet de résoudre le problème :

```
                /* fichier surveillance.c */

#include <errno.h>
```

```
#include <signal.h>
#include <time.h>

/* les variables à éditer doivent être globales */
long somme ;

void it_surveillance()
{
  time_t t_date ;
  signal(SIGUSR1, it_surveillance) ;/* réactive SIGUSR1 */
  time(&t_date) ;
  printf("date du test : %s\n",ctime(&t_date));
  printf("valeur de la somme : %ld\n",somme);
}

main()
{
  signal(SIGUSR1,it_surveillance) ;
  printf ("Envoyez le signal\n");
  while(1) somme++ ;
}
```

Exécution :

Si on lance le programme en tâche de fond, il suffira de taper sous le *shell* la commande :

```
kill -16 pid
```

pour obtenir l'affichage des variables de contrôle.

5.1.5 Deux signaux particuliers : SIGCLD et SIGHUP

Signal SIGCLD : gestion des processus *zombis*

Le signal SIGCLD se comporte différemment des autres. S'il est ignoré, la terminaison d'un processus fils, alors que le processus père n'est pas en attente, n'entraînera pas la création de processus *zombi*.

Exemple :

Le programme suivant génère un *zombi*, lorsque le père reçoit à la mort de son fils un signal SIGCLD.

```
                /* fichier test_sigcld.c */
#include <stdio.h>

main()
{
```

```
  if (fork() != 0) {
    while(1) ; /* boucle exécutée par le père */
  }
}
```

Résultat de l'exécution :

```
Systeme> test_sigcld &
Systeme> ps -a
   PID TTY      TIME COMMAND
  8507          0:00 <defunct>
```

Dans le programme qui suit, le père ignore le signal `SIGCLD`, et son fils ne devient plus un *zombi*.

```
                /* fichier test_sigcld2.c */
#include <stdio.h>
#include <signal.h>

main()
{
  signal(SIGCLD,SIG_IGN) ;/* ignore le signal SIGCLD */
  if (fork() != 0) {
    while(1) ;
  }
}
```

Résultat de l'exécution :

```
Systeme> test_sigcld2 &
Systeme> ps -a
   PID TTY      TIME COMMAND
```

Remarque : la primitive `signal()` sera détaillée plus loin.

Signal `SIGHUP` : gestion des applications longues

Ce signal peut être gênant lorsqu'on désire qu'un processus se poursuive après la fin de la session de travail (application longue). En effet, si le processus ne traite pas ce signal, il sera interrompu par le système au moment du délogage. Différentes solutions se présentent :

1. Utiliser la commande *shell* `at`, qui permet de lancer l'application à une certaine date, via un processus du système, appelé démon. Dans ce cas, le processus n'étant attaché à aucun terminal, le signal `SIGHUP` sera sans effet.
2. Inclure dans le code de l'application la réception du signal `SIGHUP`.

3. Lancer le processus en arrière-plan (en effet un processus lancé en arrière-plan traite automatiquement le signal `SIGHUP`).
4. Lancer l'application sous le contrôle de la commande `nohup`, qui entraîne un appel à `trap`, et redirige la sortie standard sur `nohup.out`.

5.1.6 Conclusion

À l'exception de `SIGCLD`, les signaux qui arrivent ne sont pas mémorisés. Ils sont ignorés, ils mettent fin aux processus, ou bien ils sont interceptés. C'est la raison principale qui rend les signaux inadaptés pour la communication inter-processus : un message sous forme de signal peut être perdu s'il est reçu à un moment où ce type de signal est temporairement ignoré. Lorsqu'un signal a été capté par un processus, le processus réadopte son comportement par défaut vis-à-vis de ce signal. Donc, si l'on veut pouvoir capter un même signal plusieurs fois, il convient de redéfinir le comportement du processus par la primitive `signal()`. Généralement, on réarme l'interception du signal le plus tôt possible (première instruction effectuée dans la procédure de traitement du déroutement).

Un autre problème est que les signaux sont plutôt brutaux : à leur arrivée, ils interrompent le travail en cours. Par exemple, la réception d'un signal pendant que le processus est en attente d'un événement (ce qui peut arriver lors de l'utilisation des primitives `open()`, `read()`, `write()`, `msgrcv()`, `pause()`, `wait()`), lance l'exécution de la fonction de déroutement ; à son retour, la primitive interrompue renvoie un message d'erreur sans s'être exécutée totalement (`errno` est positionné à `EINTR`). Par exemple, lorsqu'un processus père qui intercepte les signaux d'interruption et d'abandon, est en cours d'attente de la terminaison d'un fils, il est possible qu'un signal d'interruption ou d'abandon éjecte le père hors du `wait()` avant que le fils n'ait terminé (d'où la création d'un `<defunct>`). Une méthode pour remédier à ce type de problème, est d'ignorer certains signaux avant l'appel de telles primitives (ce qui pose alors d'autres problèmes, puisque ces signaux ne seront pas traités).

5.2 Les *pipes* ou tubes de communication

5.2.1 Introduction

Les *pipes* (ou conduits, tubes) constituent un mécanisme fondamental de communication unidirectionnelle entre processus. Ce sont des files de caractères (FIFO : *First In First Out*). Les informations y sont introduites à une extrémité et en sont extraites à l'autre. Les conduits sont implémentés comme les fichiers (ils possèdent un *i-node*), même s'ils n'ont pas de nom dans le système. La technique du conduit est fréquemment mise en œuvre dans le *shell* pour rediriger la sortie standard d'une commande sur l'entrée d'une autre (symbole « | »).

5.2.2 Particularités des tubes

Comme ils n'ont pas de nom, les tubes de communication sont temporaires, ils n'existent que le temps d'exécution du processus qui les crée. De plus, leur création doit se faire à partir d'une

primitive spéciale : `pipe()`. Plusieurs processus peuvent écrire et lire sur un même tube, mais aucun mécanisme ne permet de différencier les informations à la sortie. La capacité est limitée (en général à 4096 octets). Si on continue à écrire dans le conduit alors qu'il est plein, on se place en situation de blocage (*deadlock*). Les processus communiquant au travers de tubes doivent avoir un lien de parenté, et les tubes les reliant doivent avoir été ouverts avant la création des fils (voir le passage des descripteurs de fichiers ouverts à l'exécution de `fork()`, sous fork, cf. 4.4.1). Il est impossible de se déplacer à l'intérieur d'un tube. Afin de faire dialoguer deux processus par tube, il est nécessaire d'ouvrir deux conduits, et de les utiliser l'un dans le sens contraire de l'autre.

5.2.3 Création d'un conduit

Primitive `pipe()`

```
int pipe(int p_desc[2]) /* crée un tube */
                        /* p_desc[2] : descripteurs d'écriture et de lecture */
```

Valeur retournée : 0 si la création s'est bien passée, et -1 sinon.

- `p_desc[0]` contient le numéro du descripteur par lequel on peut lire dans le tube.
- `p_desc[1]` contient le numéro du descripteur par lequel on peut écrire dans le tube.

Ainsi, l'écriture dans `p_desc[1]` introduit les données dans le conduit, la lecture dans `p_desc[0]` les en extrait.

Sécurités apportées par le système

Dans le cas où tous les descripteurs associés aux processus susceptibles de lire dans un conduit sont fermés, un processus qui tente d'y écrire reçoit le signal `SIGPIPE`, et donc est interrompu s'il ne traite pas ce signal.

Si un tube est vide, ou si tous les descripteurs susceptibles d'y écrire sont fermés, la primitive `read()` renvoie la valeur 0 (fin de fichier atteinte).

Exemple 1 : émission du signal `SIGPIPE`

```
                /* fichier test_pipe_sig.c */

/*
 * teste l'écriture dans un conduit fermé en lecture
 */

#include <errno.h>
#include <signal.h>

void it_sigpipe()
{
```

```
  printf("Réception du signal SIGPIPE\n") ;
}

main()
{
  int p_desc[2] ;
  signal(SIGPIPE,it_sigpipe) ;
  pipe(p_desc) ;      /* création du tube */
  close(p_desc[0]) ;  /* fermeture du conduit en lecture */
  if (write(p_desc[1],"0",1) == -1)
    perror("Erreur write") ;
}
```

Résultat de l'exécution :

```
Systeme> test_pipe_sig
 Réception du signal SIGPIPE
 Erreur write: Broken pipe
```

Dans cet exemple on essaie d'écrire dans le tube alors qu'on vient de le fermer en lecture ; le signal `SIGPIPE` est émis, et le programme est dérouté. Au retour, la primitive `write()` renvoie -1 et `perror` affiche le message d'erreur.

```
                /* fichier test_pipe_read.c */
/*
 * teste la lecture dans un tube fermé en écriture
 */

#include <errno.h>
#include <signal.h>

main()
{
  int i, ret, p_desc[2] ;
  char c ;
  pipe(p_desc) ; /* création du tube */
  write(p_desc[1],"AB",2) ; /* écriture de deux lettres dans le tube */
  close(p_desc[1]) ;        /* fermeture du tube en écriture */

  /* tentative de lecture dans le tube */
  for (i=1; i<=3,i++) {
    ret = read(p_desc[0],&c,1) ;
    if (ret == 1)
      printf("valeur lue: %c\n",c) ;
    else
      perror("impossible de lire dans le tube\n") ;
  }
```

```
}
```

Résultat de l'exécution :

```
Systeme> test_pipe_read
valeur lue:A
valeur lue:B
impossible de lire dans le tube: Not a typewriter
```

Cet exemple montre que la lecture dans le tube est possible même si celui-ci est fermé en écriture. Bien sûr, lorsque le tube est vide, `read()` renvoie la valeur 0.

Application des primitives d'entrée/sortie aux tubes

Il est possible d'utiliser les fonctions de la bibliothèque standard sur un tube ayant été ouvert, en lui associant, au moyen de la fonction `fdopen()`, un pointeur sur un objet de structure `FILE`.

`write()` : les données sont écrites dans le conduit dans leur ordre d'arrivée. Lorsque le conduit est plein, `write()` se bloque en attendant qu'une place se libère. On peut éviter ce blocage en positionnant le drapeau `O_NDELAY`.

`read()` : les données sont lues dans le conduit dans leur ordre d'arrivée. Une fois extraites, les données ne peuvent pas être relues ou restituées au conduit.

`close()` : cette fonction a plus d'importance sur un conduit que sur un fichier. Non seulement elle libère le descripteur de fichier, mais lorsque le descripteur de fichier d'écriture est fermé, elle agit comme une fin de fichier pour le lecteur.

`dup()` : cette primitive combinée avec la primitive `pipe()` permet d'implémenter des commandes reliées par des tubes, en redirigeant la sortie standard d'une commande vers l'entrée standard d'une autre.

Exercice

Réaliser `ls | wc | wc`.

Cet exemple permet d'observer comment combiner les primitives `pipe()` et `dup()` afin de réaliser des commandes *shell* du type `ls|wc|wc`. Notons qu'il est nécessaire de fermer les descripteurs non utilisés par les processus exécutant la routine.

Communication entre père et fils grâce à un tube

Exercice 1 : envoi d'un message à l'utilisateur. Il s'agit d'écrire un programme qui crée un fils qui exécute la commande `mail` sur un utilisateur donné. Le père communique avec le fils via un tube et les données envoyées sur le tube sont donc envoyées par *mail* à un autre utilisateur.

Exercice 2 : mise en évidence de l'héritage des descripteurs lors d'un `fork()`

5.2.4 Conclusion

Il est possible pour un processus d'utiliser lui-même un tube à la fois en lecture et en écriture ; ce tube n'a plus alors son rôle de communication mais devient une implémentation de la structure de file. Cela permet, sur certaines machines, de dépasser la limite de taille de zone de données. Le mécanisme de communication par tubes présente un certain nombre d'inconvénients comme la non-rémanence de l'information dans le système et la limitation de la classe de processus pouvant s'échanger des informations.

5.3 Les messages

5.3.1 Introduction

La communication inter-processus (en anglais *inter-process communication*, IPC) par messages s'effectue par échange de données, stockées dans le noyau, sous forme de files. Chaque processus peut émettre des messages et en recevoir.

5.3.2 Principe

De même que pour les sémaphores (qui seront vues plus loin) et pour la mémoire partagée, une file de messages est associée à une clé qui représente son nom numérique. Cette clé est utilisée pour définir et obtenir l'identificateur de la file de messages, noté `msqid`. Celui-ci est fourni par le système au processus qui donne la clé associée.

Un processus qui désire envoyer un message doit obtenir l'identificateur de la file `msqid`, grâce à la primitive `msgget()`. Il utilise alors la primitive `msgsnd()` pour obtenir le stockage de son message (auquel est associé un type), dans une file.

De la même manière, un processus qui désire lire un message doit se procurer l'identificateur de la file par l'intermédiaire de la primitive `msgget()`, avant de lire le message en utilisant la primitive `msgrcv()`.

Structure associée aux messages : `msqid_ds`

Comme nous l'avons vu, chaque file de messages est associée à un identificateur `msqid`. On lui associe également la structure `msqid_ds`, définie dans le fichier `sys/msg.h` :

```
struct msqid_ds {
 /* une pour chaque file dans le système */
        struct ipc_perm msg_perm;     /* opérations permises */
        struct msg      *msg_first;   /* pointeur sur le premier message
                                       * d'une file */
        struct msg      *msg_last;    /* pointeur sur le dernier message
                                       * d'une file */
        ushort          msg_cbytes;   /* nombre courant d'octets de la file */
```

```
        ushort          msg_qnum;     /* nombre de messages dans la file */
        ushort          msg_qbytes;   /* nombre maximal d'octets de la file */
        ushort          msg_lspid;    /* pid du dernier processus écrivain */
        ushort          msg_lrpid;    /* pid du dernier processus lecteur */
        time_t          msg_stime;    /* date de la dernière écriture */
        time_t          msg_rtime;    /* date de la dernière lecture */
        time_t          msg_ctime;    /* date du dernier changement */
};
```

Primitive `msgget()`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget ( key_t key, int msgflg )
```

Valeur retournée : l'identificateur `msqid` de la file, ou -1 en cas d'erreur.

Cette primitive est utilisée pour créer une nouvelle file de messages, ou pour obtenir l'identificateur de file `msqid` d'une file de messages existante. Elle prend deux arguments. Le premier (`key`) est une clé indiquant le nom numérique de la file de messages. Le second (`msgflg`) est un drapeau spécifiant les droits d'accès sur la file.

Les valeurs possibles pour `key`

- `IPC_PRIVATE` ($= 0$) : la file de messages n'a alors pas de clé d'accès, et seul le processus propriétaire, ou le créateur ont accès à cette file.
- la valeur désirée de la clé de la file de messages.

Les valeurs possibles pour `msgflg` Le drapeau `msgflg` est semblable à `semflg`, utilisé pour les sémaphores, et à `shmflg`, utilisé pour la mémoire partagée. Ce drapeau est la combinaison de différentes constantes prédéfinies, permettant d'établir les droits d'accès, et les commandes de contrôle (la combinaison est effectuée de manière classique à l'aide de « OU »).

Les constantes prédéfinies dans `sys/msg.h`, et `sys/ipc.h` sont :

```
#define   MSG_R   400   /* permission en lecture pour l'utilisateur */
#define   MSG_W   200   /* permission en écriture pour l'utilisateur */

#define   IPC_CREAT   0001000   /* création d'une file de messages */
#define   IPC_EXCL    0002000   /* associé au bit IPC_CREAT provoque */
                                /* un échec si le fichier existe déjà */
```

Comment créer une file de messages

La création d'une file de messages est semblable à la création d'un ensemble de sémaphores ou d'un segment de mémoire partagée. Il faut pour cela respecter les points suivants :

- `key` doit contenir la valeur identifiant la file ; elle est obtenue par un appel à `ftok`.
- `msgflg` doit contenir les droits d'accès désirés et la constante `IPC_CREAT`.
- si l'on désire tester l'existence d'une file correspondant à la clé désirée, il faut rajouter à `msgflg` la constante `IPC_EXCL`. L'appel `msgget()` échouera dans le cas d'existence d'une telle file.

Notons que lors de la création d'une file de messages, un certain nombre de champs de l'objet de structure `msqid_ds` sont initialisés (propriétaire, modes d'accès).

Le profil de la fonction `ftok` (sous HP) est le suivant :

```
#include <sys/ipc.h>
key_t ftok(const char *path, int id);
```

- `path` doit être le chemin d'un fichier existant et accessible par le processus ;
- `id` est un caractère qui identifie de manière unique un « projet ».

Exemple d'utilisation de `msgget()`

Ce programme crée une file de messages associée à la clé 123, et vérifie le contenu des structures du système propres à cette file.

```
                /* fichier test_msgget.c */

/*
 *  exemple d'utilisation de msgget()
 */

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

#define CLE 123

main()
{
  int msqid ;  /* identificateur de la file de messages */
  char *path = "nom_de_fichier_existant" ;

    /*
     * création d'une file de messages en lecture et écriture
     * si elle n'existe pas
```

```
     */
     if (( msqid = msgget(ftok(path,(key_t)CLE),
                          IPC_CREAT|IPC_EXCL|MSG_R|MSG_W)) == -1) {
          perror("Échec de msgget") ;
          exit(1) ;
     }
   printf("identificateur de la file: %d\n",msqid);
   printf("cette file est identifiée par la clé unique : %ld\n",
                            ftok(path,(key_t)CLE)) ;
}
```

Résultat de l'exécution :

```
identificateur de la file: 700
cette file est identifiée par la clé unique : 2064941749
```

Primitive `msgctl()`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl ( int msqid, int  cmd, struct msqid_ds *buf )
```

Valeur retournée : 0 en cas de réussite, -1 sinon.

L'appel système `msgctl()` est utilisé pour examiner et modifier des attributs d'une file de messages existante. Il prend trois arguments : un identificateur de file de messages (`msqid`), un paramètre de commande (`cmd`), et un pointeur vers une structure de type `msqid_ds` (`buf`).

Les différentes commandes possibles sont définies dans le fichier `sys/ipc.h` :

`IPC_RMID (0)` : la file de messages identifiée par `msqid` est détruite. Seul le super-utilisateur ou un processus ayant pour numéro d'utilisateur `msg_perm.uid` (le propriétaire) peut détruire une file. Toutes les opérations en cours sur cette file échoueront et les processus en attente de lecture ou d'écriture sont réveillés.

`IPC_SET (1)` : donne à l'identificateur de groupe, à l'identificateur d'utilisateur, aux droits d'accès de la file de messages, et au nombre total de caractères des textes, les valeurs contenues dans le `champ msg_perm` de la structure pointée par `buf`. On met également à jour l'heure de modification.

`IPC_STAT (2)` : la structure associée à `msqid` est rangée à l'adresse pointée par `buf`.

Primitive `msgsnd()`

Le profil sous HP-UX est :

```
#include <sys/msg.h>
int msgsnd( int msqid, const void *msgp, size_t msgsz, int msgflg );
```

Sous linux, c'est :

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

int  msgsnd  ( int msqid, struct msgbuf *msgp, int msgsz, int msgflg );
```

Valeur retournée : 0 si le message est placé dans la file, -1 en cas d'erreur.

Cette primitive permet de placer un message dans une file.

Elle prend quatre arguments : l'identificateur de la file (`msqid`), un pointeur (`msgp`) vers la structure de type `msgbuf` qui contient le message, un entier (`msgsz`) indiquant la taille (en octets) de la partie texte du message, et un drapeau (`msgflg`) agissant sur le mode d'exécution de l'envoi du message.

La primitive `msgsnd()` met à jour la structure `msqid_ds` :

- incrémentation du nombre de messages de la file (`msg_qnum`)
- modification du numéro du dernier écrivain (`msg_lspid`)
- modification de la date de dernière écriture (`msg_stime`)

La structure `msgbuf`

La structure `msgbuf` décrit la structure du message proprement dit. Elle est définie dans le fichier `/usr/include/sys/msg.h` de la façon suivante :

```
struct msgbuf {
     long mtype ;       /* type du message */
     char mtext[1] ;    /* texte du message (de longueur msgsz) */
}
```

`mtype` est un entier positif. On peut s'en servir pour effectuer en lecture une sélection parmi les éléments présents dans la file. Il est impératif que le champ `mtype` soit au début de la structure. `mtext` est le message envoyé (tableau d'octets).

Il faut faire très attention, car la structure `msgbuf` est en fait une structure à taille variable. Une structure de ce type là ne doit pas être passée en paramètre autrement qu'en passant un pointeur sur la structure. Bien que `mtext` semble être un tableau d'un caractère (ce qui n'a en fait pas vraiment d'utilité, car on aurait pu mettre un unique caractère à la place), il s'agit en réalité d'un pointeur vers un tableau de `msgsz` caractères.

Cette déclaration d'un tableau d'un caractère est un *hack* du C, qui fera d'ailleurs partie, sous une forme légèrement différente (`char mtext[]`), du standard C9X.

On définira en général une autre structure (que l'on utilisera à la place de `msgbuf`), de la façon suivante :

```
#define MSG_SIZE_TEXT 256
```

```
struct msgtext {
     long mtype ;     /* type du message */
     char mtexte[MSG_SIZE_TEXT] ;    /* texte du message */
} ;
```

On pourra régler, suivant ses besoins, la taille maximum des messages échangés avec `MSG_SIZE_TEXT`. Il faut noter que le champ `mtype` est bien placé au début de la structure.

Drapeau `msgflg`

Pour ce qui concerne le drapeau `msgflg`, il est utilisé comme suit : ce paramètre est mis à 0 pour provoquer le blocage de `msgsnd()` lorsque la file de messages est pleine. Il est positionné à `IPC_NOWAIT` pour retourner immédiatement de `msgsnd()` avec une erreur lorsque la file est pleine.

Cet indicateur agit comme `O_NDELAY` sur les conduits nommés.

Primitive `msgrcv()`

Sous HP-UX, le profil est le suivant :

```
#include <sys/msg.h>
int msgrcv(int msqid, void *msgp, size_t  msgsz, long msgtyp, int msgflg);
```

Sous linux, le profil est :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv ( int msqid, struct msgbuf  *msgp, int  msgsz,
             long msgtyp, int msgflg )
```

Valeur retournée : nombre d'octets du message extrait, ou -1 en cas d'erreur.

Cette primitive permet de lire un message dans une file.

Elle prend cinq arguments : l'identificateur de la file (`msqid`), un pointeur (`msgp`) vers la structure de type `msgbuf` qui contiendra le message, un entier (`msgsz`) indiquant la taille maximum du message à recevoir, un entier (`msgtyp`) indiquant quel message on désire recevoir, et un drapeau (`msgflg`) agissant sur le mode d'exécution de l'envoi du message.

La primitive range le message lu dans une structure pointée par `msgp` qui contient les éléments suivants :

```
long mtype ;              /* type du message */
char mtext[] ;            /* texte du message */
```

La taille du champ `mtext` est fixée selon les besoins (voir `msgsnd()` pour plus de détail).

Pour ce qui concerne le paramètre `msgflg` :

- si `IPC_NOWAIT` est positionné, l'appel à `msgrcv()` retourne immédiatement avec une erreur, lorsque la file ne contient pas de message de type désiré.
- si `IPC_NOWAIT` n'est pas positionné, il y a blocage du processus appelant `msgrcv()` jusqu'à ce qu'il y ait un message du type `msgtyp` dans la file.
- si `MSG_NOERROR` est positionné, le message est tronqué à `msgsz` octets, la partie tronquée est perdue, et aucune indication de la troncature n'est donnée au processus.
- si `MSG_NOERROR` n'est pas positionné, le message n'est pas lu, et `msgrcv()` renvoie un code d'erreur.

Le paramètre `msgtyp` indique quel message on désire recevoir :

- Si `msgtyp` $= 0$, on reçoit le premier message de la file, c'est-à-dire le plus ancien.
- Si `msgtyp` > 0, on reçoit le premier message ayant pour type une valeur égale à `msgtyp`.
- Si `msgtyp` < 0, on reçoit le message dont le type a une valeur t qui vérifie :
  - t est minimum
  - $t \leqslant |\texttt{msgtyp}|$

Par exemple, si l'on considère trois messages ayant pour types 100, 200, et 300, le tableau suivant indique le type du message retourné pour différentes valeurs de `msgtyp` :

`msgtyp`	type du message retourné
0	100
100	100
200	200
300	300
−100	100
−200	100
−300	100

Exercice : écriture d'une messagerie simplifiée

Un ensemble d'utilisateurs décide de communiquer entre-eux à l'aide d'une file de messages. Un programme autonome est chargé de créer cette file. Quand un utilisateur décide d'envoyer un message à quelqu'un, il se contente de déposer un message dans cette file connue de tous. Tout message possède un champ qui contient le nom de destinataire.

Quand un utilisateur désire savoir s'il a reçu du courrier, il regarde dans la file des messages si celle-ci contient des messages à son nom.

Dans cet exemple, les utilisateurs sont représentés par leur `uid`. On utilisera le champ `type` d'un message pour y placer l'`uid` du destinataire.

On demande d'écrire:

- la définition de la structure d'un message ;
- la commande de création de la file ;
- la commande qui dépose un message = qui l'envoie à un utilisateur ;
- la commande qui lit vos message.

5.4 Les sémaphores

5.4.1 Introduction

Les sémaphores sont des objets d'IPC utilisés pour synchroniser des processus entre eux. Ils constituent aussi une solution pour résoudre le problème d'exclusion mutuelle, et permettent en particulier de régler les conflits d'accès concurrents de processus. L'implémentation qui en est faite ressemble à une synchronisation « *sleep/wakeup* ».

5.4.2 Principe

Tout comme pour les files de messages, pour créer un sémaphore, un utilisateur doit lui associer une clé. Le système lui renvoie un identificateur de sémaphore auquel sont attachés n sémaphores (ensemble de sémaphores), numérotés de 0 à $n-1$. Pour spécifier un sémaphore, l'utilisateur devra alors indiquer l'identificateur de sémaphore et le numéro de sémaphore. À chaque sémaphore est associée une valeur, toujours positive, que l'utilisateur va pouvoir incrémenter ou décrémenter du nombre qu'il souhaite. Soit N la valeur initiale, et n la valeur de l'incrément de l'utilisateur :

- si $n > 0$ alors l'utilisateur augmente la valeur du sémaphore de n et continue en séquence.
- si $n < 0$
  - si $N + n \geqslant 0$ alors l'utilisateur diminue la valeur du sémaphore de $|n|$ et continue en séquence.
  - si $N + n < 0$ alors le processus de l'utilisateur se bloque, en attendant que $N + n \geqslant 0$.
- si $n = 0$
  - si $N = 0$ alors l'utilisateur continue en séquence.
  - si $N \neq 0$ alors le processus de l'utilisateur se bloque en attendant que $N = 0$.

Nous verrons que le blocage des processus est paramétrable, c'est-à-dire que l'on peut spécifier au système de ne pas bloquer les processus mais de simplement renvoyer un code d'erreur et continuer en séquence. D'autre part, à chaque identificateur de sémaphore sont associés des droits d'accès. Ces droits d'accès sont nécessaires pour effectuer des opérations sur les valeurs des sémaphores. Ces droits sont inopérants pour les deux manipulations suivantes :

- la destruction d'un identificateur de sémaphore ;

– la modification des droits d'accès.

Pour cela, il faudra être soit super-utilisateur, soit créateur, soit propriétaire du sémaphore. Notons que le bon fonctionnement des mécanismes suppose que les opérations effectuées sur les sémaphores sont indivisibles (non interruptibles).

Structures utiles : `semid_ds`, `__sem`, `sembuf`

Chaque ensemble de sémaphores du système est associé à plusieurs structures. La donnée de ces structures n'est pas superflue, car elle permet de comprendre ce que provoquent les primitives `semget()`, `semctl()`, `semop()`, au niveau du système. Ces structures sont contenues dans `sys/sem.h`. Elles varient beaucoup d'une architecture à l'autre. Nous donnons ci-dessous celles sous HP-UX :

```
struct semid_ds {    /* une par ensemble de sémaphores dans le système */
     struct ipc_perm  sem_perm; /* opérations permises */
     struct __sem  *sem_base;   /* pointeur sur le premier */
                                /* sémaphore de l'ensemble */
     time_t  sem_otime;         /* date de la dernière opération semop()*/
     time_t  sem_ctime;         /* date de la dernière modification */
     unsigned short sem_nsems;  /* nombre de sémaphores dans l'ensemble */
     ...
};

struct __sem { /* une pour chaque sémaphore dans le système */
     unsigned short int semval;  /* valeur du sémaphore */
     unsigned short int sempid;  /* pid du dernier processus ayant effectué */
                                 /* une opération sur le sémaphore */
     unsigned short int semncnt; /* nombre de processus en attente que semval */
                                 /* soit supérieur à la valeur courante */
     unsigned short int semzcnt; /* nombre de processus en attente que */
                                 /* semval devienne nulle */
   };

struct sembuf {
  unsigned short int  sem_num; /* numéro du sémaphore */
  short               sem_op;  /* opération à réaliser */
  short               sem_flg; /* indicateur d'opération (drapeaux) */
};
```

Primitive `semget()`

Sous HP-UX :

```
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

Sous linux :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

Valeur retournée : l'identificateur de sémaphore `semid`, ou -1 en cas d'erreur.

L'appel système `semget()` est utilisé pour créer un nouvel ensemble de sémaphores, ou pour obtenir l'identificateur de sémaphore d'un ensemble existant. Le premier argument, `key`, est une clé indiquant le nom numérique de l'ensemble de sémaphores. Le second, `nsems`, indique le nombre de sémaphores de l'ensemble. Le dernier, `semflg`, est un drapeau spécifiant les droits d'accès sur l'ensemble de sémaphores.

Les valeurs possibles pour `key`

- `IPC_PRIVATE` ($= 0$) : l'ensemble n'a alors pas de clé d'accès, et seul le processus propriétaire, ou le créateur a accès à l'ensemble de sémaphores.
- la valeur désirée de la clé de l'ensemble de sémaphores.

Les valeurs possibles pour l'option `semflg` Ce drapeau est en fait la combinaison de différentes constantes prédéfinies, permettant d'établir les droits d'accès, et les commandes de contrôle (la combinaison est effectuée de manière classique à l'aide de « OU »). Notons la similitude existant entre les droits d'accès utilisés pour les sémaphores, et les droits d'accès aux fichiers UNIX : on retrouve la notion d'autorisation en lecture ou écriture aux attributs utilisateur/groupe/autres. Le nombre octal défini en octal pourra être utilisé (en forçant à 0 les bits de droite d'exécution 3, 6 et 9). Les constantes prédéfinies dans `sys/sem.h`, et `sys/ipc.h` sont :

```
#define   SEM_A        0200     /* permission de modification (a pour alter)*/
#define   SEM_R        0400     /* permission en lecture */
#define   IPC_CREAT 0001000     /* création d'un ensemble de sémaphores */
#define   IPC_EXCL  0002000     /* associé au bit IPC_CREAT provoque un */
                                /* échec si le fichier existe déjà */
```

Pour créer un ensemble de sémaphores, les points suivants doivent être respectés :

- `key` doit contenir la valeur identifiant l'ensemble (différent de `IPC_PRIVATE`$= 0$).
- `semflg` doit contenir les droits d'accès désirés, et la constante `IPC_CREAT`.
- si l'on désire tester l'existence d'un ensemble correspondant à la clé `key` désirée, il faut rajouter à `semflg` la constante `IPC_EXCL`. L'appel à `semget()` échouera dans le cas d'existence d'un tel ensemble.

Notons que lors de la création d'un ensemble de sémaphores, un certain nombre de champs de l'objet de structure `semid_ds` sont initialisés (propriétaire, modes d'accès).

Exemple : ce programme crée un ensemble de quatre sémaphores associé à la clé 123.

```
                /* fichier test_semget.c */

/* exemple d'utilisation de semget() */

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define CLE 123

main()
{
  int semid ;  /* identificateur des sémaphores */
  char *path = "nom_de_fichier_existant" ;

    /*
     * allocation de quatre sémaphores
     */
  if (( semid = semget(ftok(path,(key_t)CLE), 4,
                       IPC_CREAT|IPC_EXCL|SEM_R|SEM_A)) == -1) {
    perror("Échec de semget") ;
    exit(1) ;
  }
  printf(" le semid de l'ensemble de sémaphore est : %d\n ",semid) ;
  printf(" cet ensemble est identifié par la clé unique : %d\n ",
                            ftok(path,(key_t)CLE)) ;
}
```

Résultat de l'exécution :

```
Systeme> test_semget
 le semid de l'ensemble de sémaphore est : 2
 cet ensemble est identifié par la clé unique : 2073103658
Systeme> ipcs
IPC status from /dev/kmem as of Fri Nov 20 11:08:06 1998
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
m    100 0x0000004f --rw-rw-rw-     root     root
Sémaphores:
s      2 0x7b910d2a --ra-------     bernard ens
Systeme> test_semget
Échec de semget: File exists
```

Primitive `semctl()`

Sous HP-UX :

```
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

La construction « `, ...` » représente un paramètre de taille variable. En l'occurrence, il peut ici s'agir soit d'un entier, soit d'un pointeur sur une structure `struct semid_ds`, soit d'un tableau d'entiers. On peut voir le quatrième paramètre comme une union :

```
union semun {
  int val ;
  struct semid_ds *buf ;
  ushort array[] ;  /* tableau de taille égale au nombre */
                    /* de sémaphores de l'ensemble */
} arg ;
```

La valeur retournée dépend de la valeur de `cmd` :

- si `cmd` = `GETVAL` : valeur de `semval`
- si `cmd` = `GETPID` : valeur de `sem_pid`
- si `cmd` = `GETNCNT` : valeur de `sem_ncnt`
- si `cmd` = `GETZCNT` : valeur de `sem_zcnt`

Pour les autres valeurs de `cmd`, la valeur retournée est 0 en cas de réussite, et -1 en cas d'erreur.

L'appel système `semctl()` est utilisé pour examiner et changer les valeurs de sémaphores d'un ensemble de sémaphores. Elle a besoin de quatre arguments : un identificateur de l'ensemble de sémaphores (`semid`), le numéro du sémaphore à examiner ou à changer (`semnum`), un paramètre de commande (`cmd`), et une variable de taille variable analogue au type union (`arg`).

Les diverses commandes possibles pour `semctl()` sont décrites dans les fichiers `sys/ipc.h` et `sys/sem.h`, les premières données ci-dessous sont communes à tous les IPC, les secondes sont spécifiques.

`IPC_RMID (0)` : l'ensemble de sémaphores identifié par `semid` est détruit. Seul le super-utilisateur ou un processus ayant pour numéro d'utilisateur `sem_perm.uid` peut détruire un ensemble. Tous les processus en attente sur les sémaphores détruits sont débloqués et renvoient un code d'erreur.

`IPC_SET (1)` : donne à l'identificateur de groupe, à l'identificateur d'utilisateur, et aux droits d'accès de l'ensemble de sémaphores, les valeurs contenues dans le champ `sem_perm` de la structure pointée par le quatrième paramètre de `semctl` qui doit être un pointeur sur `struct semid_ds`. On met également à jour l'heure de modification.

`IPC_STAT (2)` : la structure associée à `semid` est rangée à l'adresse pointée par le quatrième paramètre de `semctl` qui doit être un pointeur sur `struct semid_ds`.

Commandes spécifiques aux sémaphores

`GETNCNT (3)` : la fonction retourne la valeur de `semncnt` qui est le nombre de processus en attente d'un incrément de la valeur d'un sémaphore particulier.

`GETPID (4)` : la fonction retourne le *pid* du processus qui a effectué la dernière opération sur un sémaphore particulier.

`GETVAL (5)` : la fonction retourne la valeur `semval` du sémaphore de numéro `semnum`.

`GETALL (6)` : les valeurs `semval` de tous les sémaphores sont rangées dans le tableau dont l'adresse est le tableau d'entier passé en quatrième paramètre.

`GETZCNT (7)` : la fonction retourne la valeur `semzcnt` qui est le nombre de processus en attente d'un passage à zéro de la valeur d'un sémaphore particulier.

`SETVAL (8)` : cette action est l'initialisation de la valeur du sémaphore. La valeur `semval` du sémaphore de numéro `semnum` est mise à la valeur donnée en quatrième paramètre de `semctl`.

`SETALL (9)` : les valeurs `semval` des `semnum` premiers sémaphores sont modifiées en concordance avec les valeurs correspondantes du tableau dont l'adresse est le quatrième paramètre de `semctl` vu comme un pointeur sur un tableau d'entiers.

Primitive `semop()`

Sous HP-UX :

```
#include <sys/sem.h>
int semop( int semid, struct sembuf *sops, unsigned int nsops );
```

Valeur retournée : la valeur `semval` du dernier sémaphore manipulé, ou -1 en cas d'erreur.

L'appel système `semop()` permet d'effectuer des opérations sur les sémaphores. Il utilise trois arguments : un identificateur d'ensemble de sémaphores (`semid`), un pointeur vers un tableau de structures de type `struct sembuf` (`sops`), et un entier donnant le nombre d'éléments de ce tableau (`nsops`). Il sera donc possible de spécifier en une fois plusieurs opérations sur un sémaphore. La structure `sembuf` spécifie le numéro du sémaphore qui sera traité, l'opération qui sera réalisée sur ce sémaphore, et les drapeaux de contrôle de l'opération.

Le type d'opération dépend de la valeur de `sem_op` (champ de la structure `sembuf`) :

- Si `sem_op` < 0 (demande de ressource)
  - si `semval` $\geqslant |$`sem_op`$|$ alors `semval` $=$ `semval` $-\ |$`sem_op`$|$: décrément du sémaphore
  - si `semval` $< |$`sem_op`$|$ alors le processus se bloque jusqu'à ce que `semval` $\geqslant |$`sem_op`$|$
- Si `sem_op` $= 0$ (lecture)
  - si `semval` $= 0$ alors l'appel retourne immédiatement
  - si `semval` $\neq 0$ alors le processus se bloque jusqu'à ce que `semval` $= 0$
- Si `sem_op` > 0 (restitution de ressource) : alors `semval` $=$ `semval` $+$ `sem_op`

En effectuant des opérations `semop()` qui n'utilisent que des valeurs de `sem_op` égales à 1 ou -1, on trouve le fonctionnement des sémaphores de Dijsktra.

System V fournit une gamme d'opérations plus étendue en jouant sur la valeur de `semop`. L'implémentation est alors beaucoup plus complexe et la démonstration des garanties d'exclusion mutuelle est extrêmement délicate.

Le positionnement de certains drapeaux (dans le champ `sem_flg` de la structure `sembuf`) entraîne une modification du résultat des opérations de type `semop()` :

- `IPC_NOWAIT` : évite le blocage du processus (dans le cas où l'on aurait dû avoir ce comportement) et renvoie un code d'erreur.
- `SEM_UNDO` : les demandes et restitutions de ressource sont automatiquement équilibrées à la fin du processus. Toutes les modifications faites sur les sémaphores par un processus sont défaites à la mort de celui-ci. Pendant toute la vie d'un processus, les opérations effectuées avec le drapeau `SEM_UNDO` sur tous les sémaphores de tous les identificateurs sont cumulées, et lors de la mort de ce processus, le système refait ces opérations à l'envers. Cela permet de ne pas bloquer indéfiniment des processus sur des sémaphores, après la mort accidentelle d'un processus. Notons que ce procédé coûte cher, tant au point de vue temps CPU qu'au point de vue réservation de place mémoire.

Exercice 1 : un premier processus (exécutant le programme `processus1`) crée un ensemble de sémaphores, fixe la valeur de l'un des sémaphores à 1, puis demande une ressource. Il se met en attente pendant 10 secondes. Un second processus (exécutant le programme `processus2`) récupère l'identificateur `semid` de l'ensemble de sémaphores, puis demande également une ressource. Il reste bloqué jusqu'à ce que le premier processus ait fini son attente et libère alors la ressource.

Exercice 2 : les cuisinières et les quatre-quarts.

5.4.3 Sémaphores de Dijsktra

Principe : les sémaphores de Dijkstra (prononcé [DAÏKSTRA]) sont une solution simple au problème de l'exclusion mutuelle. On accède à ces sémaphores par les deux opérations P (acquisition) et V (libération)[1]. Lorsqu'on réalise l'opération P sur un sémaphore, sa valeur s est décrémentée de 1 si s est différent de 0 ; sinon, le processus appelant est bloqué et est placé dans une file d'attente liée au sémaphore. Lorsqu'on réalise l'opération V sur un sémaphore, on incrémente sa valeur s de 1 s'il n'y a pas de processus dans la file d'attente ; sinon, s reste inchangée, et on libère le premier processus de la file.

5.4.4 Implémentation des sémaphores de Dijkstra

Le code ci-dessous réalise l'implémentation des sémaphores de Dijkstra à partir des mécanismes de sémaphores de System V. La fonction `sem_create()` permet de créer un sémaphore. Les opérations P et V sont réalisées par les fonctions `P()` et `V()`. La fonction `sem_delete()` permet de détruire un sémaphore.

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

1. « P » et « V » proviennent des mots hollandais « passeren » (passer) et « vrijgeven » (libérer). (D'après Dijkstra, en réponse à un étudiant le 20/11/1999.)

```
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>

int dij_create(key_t cle, int val_init) {
  int sid;

  /* obtention du sémaphore */
  if ((sid = semget(cle, 1, IPC_CREAT | 0644)) == -1) {
    perror("dij_create:semget");
    exit(1);
  }

  /*initialisation du sémaphore */
  if (semctl(sid, 0, SETVAL, val_init) == -1) {
    perror("dij_create:semctl");
    exit(1);
  }

  return sid;
}

void dij_delete(int semid) {
  /* destruction du sémaphore */
  if (semctl(semid, 0, IPC_RMID) == -1) {
    perror("dij_delete: semctl");
    exit(1);
  }
}

void P(int semid) {
  struct sembuf sops[1];

  sops[0].sem_num = 0;
  sops[0].sem_op = -1;
  sops[0].sem_flg = SEM_UNDO;

  if (semop(semid, sops, 1) == - 1) {
    perror("P:semop");
    exit(1);
  }
}

void V(int semid) {
```

```
  struct sembuf sops[1];

  sops[0].sem_num = 0;
  sops[0].sem_op = +1;
  sops[0].sem_flg = SEM_UNDO;

  if(semop(semid, sops, 1) == -1) {
    perror("V:semop");
    exit(1);
  }
}
```

5.4.5 Exemple d'utilisation des sémaphores de Dijkstra

L'exemple qui suit montre une utilisation simple de l'implémentation des sémaphores de Dijkstra réalisée ci-dessus. Un processus est bloqué sur un sémaphore, et se débloque une fois que son fils libère la ressource.

```
                /*fichier test_sem_dijkstra.c */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "dijkstra.h"

#define CLE 1

main()
{
  int sem ;

  sem = sem_create(CLE,0) ;
  printf("Création du sémaphore d'identificateur %d\n",sem);
  if (fork() == 0) {
    printf("Je suis le fils et j'attends 15 secondes...") ;
    sleep(15) ;
    printf("Je suis le fils et je fais V sur le sémaphore") ;
    V(sem) ;
    exit(0) ;
  }
  else {
    printf("Je suis le père et je me bloque en faisant P \
                  sur le sémaphore") ;
    P(sem) ;
    printf("Je suis le père et je suis libre") ;
    sem_delete(sem) ;
  }
```

```
}
```

Résultat de l'exécution : le programme `test_sem_dijkstra` est lancé en arrière plan, ce qui permet de vérifier l'existence dans le système du sémaphore créé, grâce à la commande *shell* `ipcs` :

```
Systeme> test_sem_dijkstra &
Creation du sémaphore d'identificateur 275
[1]     1990
Systeme> Je suis le fils et j'attends 15 secondes...
Je suis le père et je me bloque en faisant P sur le sémaphore

ipcs
IPC status from /dev/kmem as of Fri Nov 20 10:56:41 1998
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
m    100 0x0000004f --rw-rw-rw-     root     root
Semaphores:
s    275 0x01910d23 --ra-ra-ra-     bernard   ens
Systeme> Je suis le fils et je fais V sur le sémaphore
Je suis le père et je suis libre

ipcs
IPC status from /dev/kmem as of Fri Nov 20 10:56:57 1998
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
m    100 0x0000004f --rw-rw-rw-     root     root
Semaphores:
[1] +  Done                    test_sem_dijkstra &
```

5.4.6 Conclusion

Le mécanisme des sémaphores sous System V est complexe à mettre en œuvre. D'autre part, dans un programme utilisant des sémaphores, il faut être capable de démontrer que l'accès aux ressources partagées est exclusif, et qu'il n'y a ni interblocage, ni situation de famine (dans laquelle on n'obtient jamais d'accès). Si cette analyse est assez difficile avec les sémaphores de Dijkstra, elle devient extrêmement délicate avec les primitives de System V.

Chapitre 6

Communication réseaux : les sockets

6.1 Définition

Une socket est un point de communication par lequel un processus peut émettre et recevoir des informations (communication dans les deux sens). Deux domaines de communication existent :

- Le domaine Unix : communication locale à une machine hôte. Les sockets sont alors assimilées à des fichiers spéciaux sans données associées.
- Le domaine Internet : communication via les réseaux qui permettent à deux processus situés sur des machines hôtes différentes de communiquer entre eux.

Une socket est identifiée par un descripteur de même nature que ceux identifiant les fichiers. Tout processus fils hérite donc des descripteurs de sockets de son père.

Dans cette section, nous nous intéresserons uniquement aux sockets du domaine Internet qui permettent la communication entre un processus serveur et un ou plusieurs processus client(s). Le serveur propose un service (messagerie, web, ...) et attend des requêtes émanant d'un client.

6.2 Les types de sockets

Le type d'une socket est déterminé à partir d'un ensemble de propriétés de communication :

- fiabilité de la transmission : aucune donnée perdue ;
- préservation de l'ordre des données ;
- non duplication des données ;
- communication en mode connecté : une connection ou liaison virtuelle est établie entre deux points avant le transfert des messages ;

– préservation des limites de messages $\Longrightarrow$ un envoi de n octets implique une seule réception de n octets ;
– envoi de messages urgents.

Les principaux types de socket sont les suivants :

– `SOCK_STREAM` ou « flot d'octets » fournit une transmission fiable, ordonnée, sans duplication, en mode connecté et autorise des messages urgents. Les limites des messages ne sont pas préservées.
– `SOCK_DATAGRAM` ou « messages » fournit une transmission sans fiabilité (possibilié de pertes, de duplication et désordre pour les messages) et en mode non connecté. Les limites des messages émis sont préservées.
– `SOCK_RAW` permet l'accès directement aux protocoles de bas niveaux et est reservé au super-utilisateur.

6.3 Création d'une socket

Elle permet l'acquisition et l'initialisation d'entrées dans les tables du système de gestion de fichiers

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

int socket(int domain, int type, int protocol);
/*
   domain    = PF_INET ou PF_UNIX, ....
   type      = SOCK_STREAM ou SOCK_DGRAM ou SOCK_RAW,...
   protocole = 0 par défaut
*/
```

Les valeurs de domaine, type et protocole sont définies dans le fichier `/usr/include/sys/socket.h`.

La valeur de retour est un descripteur à travers lequel un processus pourra émettre et recevoir des messages. En cas d'erreur, le code de retour est de -1.

6.4 Nommage d'une socket

6.4.1 Primitive `bind`

Une socket est créée sans nom. La relation entre un nom et une socket se fait de manière explicite par la fonction `bind`.

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

```
int bind(int s, const struct sockaddr *name, int namelen);
/*
   s       = descripteur de socket retourné par la fonction socket()
   name    = nom associé à la socket qui pointe sur une structure
             contenant une adresse Internet
   namelen = longueur de l'adresse
*/
```

6.4.2 Domaine Internet : préparation de l'adresse

Pour le domaine Internet (Famille `AF_INET`) les adresses des sockets ont la structure `sockaddr_in` définie dans `/usr/include/netinet/in.h`.

```
/*
 * Adressse Internet (une structure pour des raisons historiques)
 */
struct in_addr {
        u_long s_addr;
};

/*
 * Adresse Socket,  style Internet.
 */
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
/*
   sin_familiy = AF_INET
   sin_port    = numéro de port
   sin_addr    = l'adresse internet de la machine locale
   sin_zero    = chaîne devant être initialisée à zéro
*/
```

- Le *numéro de port* référence de manière unique un processus exécutant un service particulier (par exemple, un serveur de mail, de news, ...) sur une machine donnée identifiée par une adresse appelée adresse IP. Le choix du numéro de port :
  - Il peut s'agir d'un port correspondant à un service existant. Il faut alors faire appel à la fonction `getservbyname ()` qui est décrite dans la section 6.7.
  - Il peut s'agir d'un port quelconque défini par le programmeur du serveur d'application. Ce port doit alors être $\geqslant$ `IPPORT_RESERVED` i.e. $\geqslant 1024$
  - Il peut s'agir d'un port attribué par le système, ce qui est en général le cas pour le client. Le champ `sin_port` de la structure doit alors être positionné à zéro.

– L'*adresse Internet* de la machine locale est obtenue :
  – Soit par les fonctions `gethostname(), gethostbyname()` détaillées dans la section 6.7
  – Soit, dans le cas d'un serveur, la valeur `INADDR_ANY` est choisie. (Le processus serveur peut alors recevoir des messages sur n'importe quelle interface réseau par exemple plusieurs cartes Ethernet).

6.4.3 Remarque

Les adresses de type Internet sont définies dans une structure nommée `sockaddr_in` alors que tous les appels systèmes utilisent une structure nommée `sockaddr`. Il peut en effet exister d'autres types d'adresses comme celles propres au domaine UNIX ou X25, Novell IPX, ... Il a donc été spécifié, pour avoir une interface commune quelque soit le type d'adresses, d'avoir une structure générique `struct sockaddr` pour les appels systèmes. Afin d'éviter les erreurs de *warning* lors de la compilation, il est donc recommandé de faire un *cast* adéquat. Exemple :

```
struct sockaddr_in  serv_addr; /* adresse Internet du serveur    */

if ( bind (sockfd, (struct sockaddr *)&serv_addr, namelen) < 0) {
   perror("bind ");
   exit(1);
 }
```

6.5 Fermeture d'une socket

La fermeture implique la libération des ressources système associées. L'appel à `close()` n'interrompt pas immédiatement la communication : le système continue de transmettre les données stockées dans son buffer, s'il en existe.

```
#include <unistd.h>
int close(int s);
/*
   s       = descripteur de socket retourné par la fonction socket()
 */
```

Il existe une autre primitive UNIX qui permet de contrôler plus finement la fermeture des sockets.

```
  int shutdown(int s, int how);
/*
   s       = descripteur de socket retourné par la fonction socket()
   how     = définit la méthode de fermeture
                0 : fermeture en réception seulement
                1 : fermeture en émission
                2 : fermeture en émission et réception, <=> au close()
 */
```

Une valeur de retour de -1 signale une erreur.

6.6 La communication entre client-serveur

Dans cette section, nous étudierons seulement les communications en mode connecté (type `SOCK_STREAM`) qui sont les plus utilisées et qui correspondent au protocole TCP/IP (Transmission Control Protocol/Internet Protocol). Ce protocole et le mode de communication non connecté seront étudiés plus en détail dans le cours Réseaux.

Dans le modèle client-serveur décrit dans la figure 6.1 sont énumérés en gras l'ensemble des primitives système à utiliser pour réaliser la communication. Le serveur ne traite ici qu'un client à la fois.

6.6.1 La primitive `listen`

Elle permet à un serveur de signaler au système qu'il accepte les demandes de connexion.

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

int listen(int s, int backlog);
/*
   s           = descripteur de socket d'écoute du serveur retourné
                 par socket()
   backlog     = SOMAXCONN nombre maximal de demandes de connexion
                 en attente
 */
```

Le second paramètre définit la taille de la file d'attente des demandes de connexion émises par les clients et non encore traitées par le serveur. Il est différent du nombre maximum de clients possible.

La constante `SOMAXCONN` est définie dans `/usr/include/sys/socket.h`.

6.6.2 La primitive `accept`

Elle permet d'extraire une connexion pendante dans la file associée à une socket pour laquelle un appel `listen` a été réalisé. Si une connexion client est en attente, la primitive retourne une nouvelle socket appelée socket de dialogue (ou socket active) à travers laquelle se fera le dialogue client/serveur. La socket d'écoute (ou encore socket passive) est uniquement là pour recevoir des demandes de connexions. Si aucune connexion n'est en attente, le processus par défaut est bloqué.

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

int accept(int s, struct sockaddr *addr, int *addrlen);
/*
   s           = descripteur de socket d'écoute du serveur
   addr        = pointeur sur une structure sauvegardant l'adresse du
```

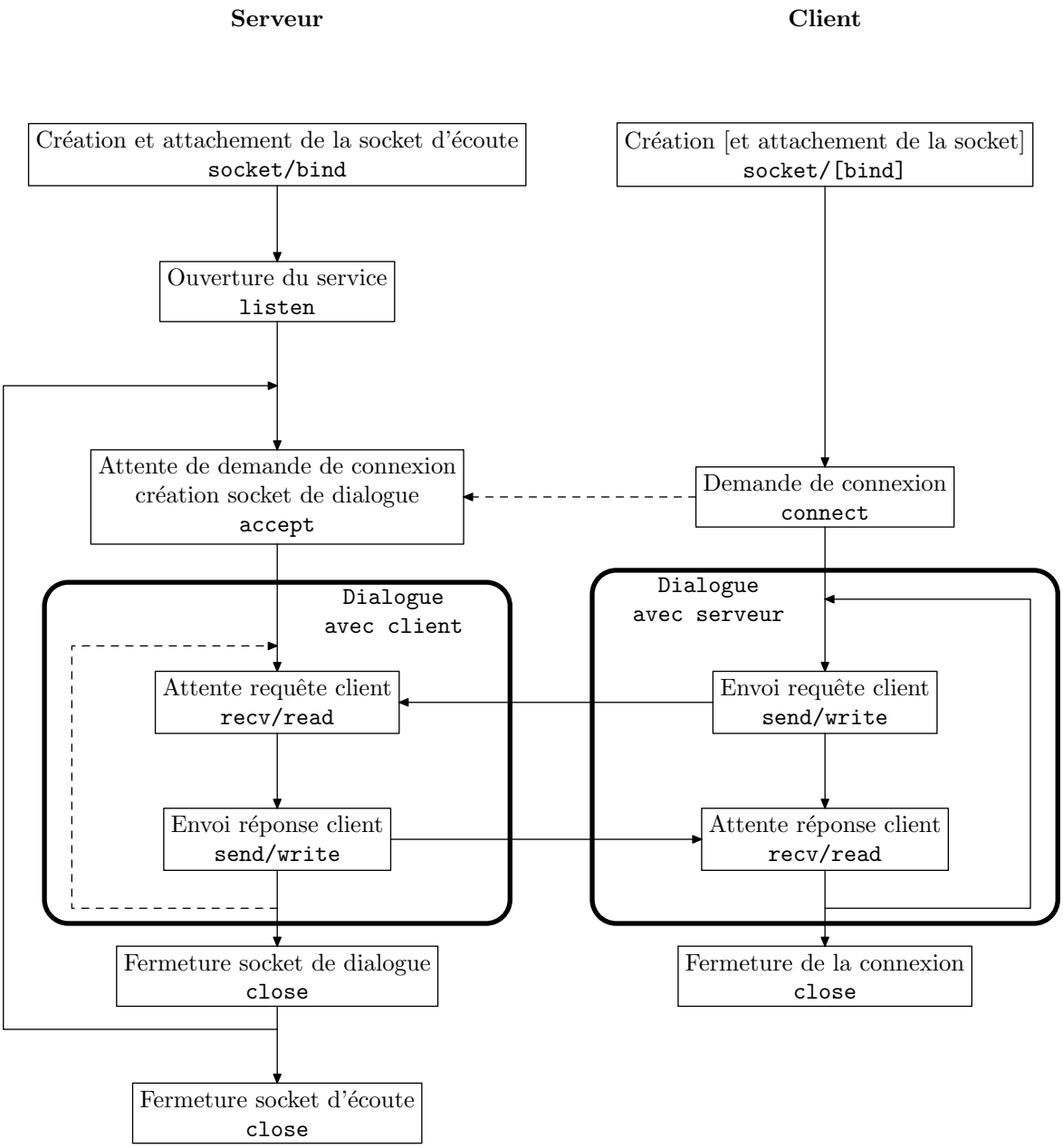

FIG. 6.1 – *Modèle un serveur-un client*

```
                client connecté
  addrlen     = pointeur sur la taille de la zone allouée à addr
 */
```

Le paramètre `addrlen` doit pointer sur un entier dont le contenu est de taille `sizeof(struct sockaddr_in)`.

6.6.3 La primitive `connect`

Elle permet de créer une sorte de liaison virtuelle entre les deux processus dont les deux extrémités sont les sockets. Cette liaison est bidirectionnelle.

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int s, struct sockaddr *name, int namelen);
/*
   s       = descripteur de socket retourné par la fonction socket()
   name    = nom associée à la socket qui pointe sur une structure
             contenant l'adresse Internet du serveur
   namelen = longueur de l'adresse
*/
```

L'exécution réussie du connect a pour effet de débloquer le serveur en attente sur l'`accept`. Le champ name est une structure de type `sockaddr_in` (cf section 6.4.2).

- Le *numéro de port* référence le port se service sur lequel est lancé le serveur. Le choix du numéro de port :
  - Il peut s'agir d'un port correspondant à un service existant. Il faut alors faire appel à la fonction `getservbyname ()` qui est décrite dans la section 6.7.
  - Il peut s'agir d'un port quelconque défini par le programmeur du serveur d'application. Ce port doit alors être $\geqslant$ `IPPORT_RESERVED` i.e. $\geqslant 1024$
- L'*adresse Internet* du serveur est obtenue par la fonction `gethostbyname` détaillée dans la section 6.7.

6.6.4 Les primitives d'émission et de réception

Une fois la connexion établie entre le serveur et le client, les deux processus peuvent échanger des informations ou plus exactement des flots d'octets dans le cas des `SOCK_STREAM`. Le découpage en différents messages n'est pas préservé sur la socket, i.e. le résultat d'une écriture chez le client peut provoquer plusieurs opérations de lecture chez le serveur.

1. **L'émission**
   L'écriture sur une socket connectée peut être réalisée soit la primitive `write` qui est utilisée aussi pour les fichiers ou par la primitive `send` utilisée uniquement pour les sockets.

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

```
ssize_t write(int s, const void *buf, size_t nbyte);
/*
   s           = descripteur de socket (de dialogue pour le serveur)
   buf         = adresse en mémoire du message à envoyer
   nbytes      = longueur du message
 */

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

int send(int s, const char *msg, int len, int flags);
/*
   s           = descripteur de socket (de dialogue pour le serveur)
   msg         = adresse en mémoire du message à envoyer
   len         = longueur du message
   flags       = 0 ou MSG_OOB pour les données urgentes
 */
```

2. **La réception**

Pour la réception, il est aussi possible d'utiliser la primitive standard `read` ou celle, plus spécifique aux sockets, `recv`.

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

size_t read(int s, void *buf, size_t nbyte);
/*
   s           = descripteur de socket (de dialogue pour le serveur)
   buf         = adresse en mémoire de sauvegarde du message
   nbytes      = longueur de la zone allouée à l'adresse buf
 */

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

int recv(int s, char *buf, int len, int flags);
/*
   s           = descripteur de socket (de dialogue pour le serveur)
   msg         = adresse en mémoire du message à envoyer
   len         = longueur du message
   flags       = 0 ou MSG_OOB ou MSG_PEEK
*/
```

`MSG_OOB` permet la lecture des données urgentes (out-of-band) `MSG_PEEK` permet de lire la

donnée mais elle n'est pas sortie de la file de réception, i.e. une autre lecture verra la même donnée.
Quand le serveur (resp. client) ferme la connexion, la primitive `read` retourne 0 au niveau du client (resp. serveur).

6.6.5 Serveur concurrent

Dans le modèle à accès concurrent, quand le serveur reçoit une demande de connexion, il lance un processus fils pour chaque client. Il peut ainsi se remettre à l'écoute d'autres clients éventuels.

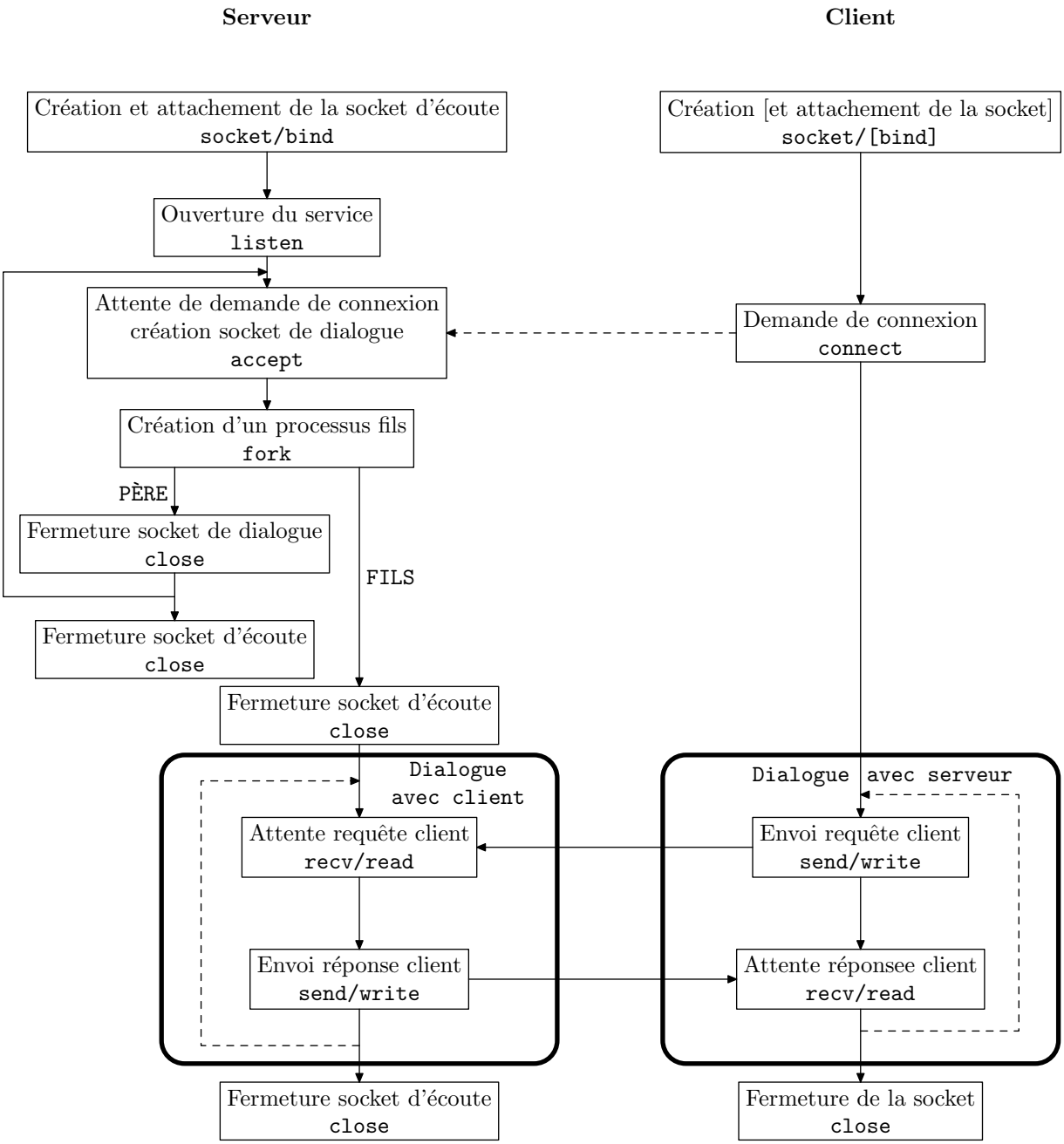

FIG. 6.2 – *Serveur Multi-clients*

6.7 Fonctions et structure de bases pour les adresses IP

6.7.1 Fonction `gethostname`

Cette fonction permet d'accéder au nom de la machine courante. En cas de succès, 0 est retourné. En cas d'erreur -1 est retourné.

```
#include <unistd.h>

int gethostname(char *name, size_t len);
/*  name  = le nom du système local est placé à l'adresse name
 *  len   = indique l'espace qui a été réservé pour le champ name
*/
```

6.7.2 Fonction `gethostbyname`

Les ordinateurs sont généralement connus par des noms et non par leur adresse IP. La fonction de base qui permet d'établir la correspondance est la fonction `gethostbyname`. En cas de succès, elle retourne un pointeur sur une structure `hostent` qui est décrite ci-dessous.

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);


struct hostent {
                char    *h_name;        /* nom officiel de la machine */
                char    **h_aliases;    /* liste d'alias */
                int     h_addrtype;     /* type d'adresse = AF-INET*/
                int     h_length;       /* longueur de l'adresse  = 4 octets = 32 bits*/
                char    **h_addr_list;  /* liste des adresses */
#define h_addr  h_addr_list[0]          /* la première adresse de la liste */
        }
```

L'appel à la fonction `gethostbyname(`*localhost*`)` permet d'obtenir les informations relatives à la machine hôte courante.

6.7.3 Fonction `getservbyname`

Les services comme les machines hôtes sont aussi connues par leurs noms. La correspondance entre le nom du service et le numéro de port est obtenue par l'appel à la fonction `getservbyname`. En cas de succès, elle retourne un pointeur sur une structure `servent` qui est décrite ci-dessous. Le numéro du port est retourné dans l'ordre du réseau défini par le réseau.

```
#include <netdb.h>
```

```
struct servent *getservbyname(const char *name, const char *proto);

struct servent {
                char    *s_name;        /* nom officiel du service */
                char    **s_aliases;    /* liste d'alias           */
                int     s_port;         /* numéro du port          */
                char    *s_proto;       /* protocole à utiliser    */
        }
```

6.7.4 Fonctions `bcopy/memcpy` et `bzero/memset`

Ces fonctions permettent de manipuler des octets sans interpréter les données et sans supposer que les données se terminent par `'\0'`. Nous avons besoin de ces fonctions quand nous utilisons les sockets car nous manipulons les champs comme des adresses IP pouvant contenir un ou plusieurs octets à 0 sans que ce soient des chaînes de caractères. Le premier groupe dont le nom commence par `cccccb` (pour *bytes*) a comme origine `UNIX BSD` et le second groupe dont le nom commence par `mem` (pour *memory*) fait partie de libraires `C ANSI` standard.

```
#include <string.h>

void bcopy (const void *src, void *dest, int n);

void *memcpy(void *dest, const void *src, size_t n);

void bzero(void *s, int n);

void *memset(void *s, int c, size_t n);
```

La fonction `bcopy()` copie les n premiers octets de la source `src` vers la destination `dest`.

La fonction `memcpy()` copie les n premiers octet de la zone mémoire pointée par `src` vers celle pointée par `dest`.

La fonction `bzero()` positionne les n premiers octets de la chaîne pointée par `s` à zéro.

La fonction `memset()` remplit les n premiers octets de la zone mémoire pointée par `s` avec le caractère `c`.

6.8 Fonctions de conversion des entiers

Si on considère un entier de 16 bits, il y a deux manières de stocker ces 2 octets en mémoire. Soit l'octet de poids faible est rangé à l'adresse de poids faible (*little endian*), soit l'octet de poids fort est rangé à l'adresse de poids faible (*big endian*). Il n'y a pas de standard entre ces deux rangements et nous pouvons rencontrer ces deux formats sur les machines hôtes (*host byte order*).

Pour pouvoir faire communiquer deux machines hôtes avec deux formats différents à travers le réseau, les protocoles de communication doivent imposer un ordre de transfert (*network byte order*). Des fonctions système permettent de passer du format de la machine vers le format du réseau. Dans les fonctions ci-dessous `h` signifie *host* et `n` *network*, `s` *short* et `l` *long*.

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

unsigned long int htonl(unsigned long int hostlong);

unsigned short int htons(unsigned short int hostshort);

unsigned long int ntohl(unsigned long int netlong);

unsigned short int ntohs(unsigned short int netshort);
```

6.8.1 Exercices

Exercice 1

Soit un serveur et un client permettant de lancer une commande (comme `rsh`) sur la machine hébergeant le serveur et de récupérer le résultat à distance (côté client). Les fichiers sources contiennent des erreurs et doivent être débuggés.

Exercice 2

- Écrire un serveur echo qui reçoit un message d'un client et lui renvoie. Le client peut envoyer n messages différents.
- Modifier ce code pour permettre au serveur de traiter plusieurs clients.

Quatrième partie

Environnement PC

Chapitre 7

Architecture de l'ordinateur

La carte-mère (*Mainboard* ou *Motherboard*) est l'un des principaux éléments du PC. Elle se présente sous la forme d'un circuit imprimé sur lequel sont présents divers composants. La carte-mère détermine le type de tous les autres composants comme le format des cartes d'extension (ISA, EISA, PCI, ...) ou le type de barrettes à utiliser (SIMM, DIMM, ...). Actuellement, de plus en plus de cartes-mères intègrent des composants habituellement séparés : contrôleur disque, floppy, carte réseau ainsi que prises séries et parallèles sont directement sur la carte-mère.

Un *chipset* est un composant utilisé par la carte mère pour gérer la coopération entre plusieurs éléments de la carte mère. Le *chipset* est composé de plusieurs *chips*, chacune pilotant un composant précis : le processeur (CPU), l'horloge système, le contrôleur d'interruption, le contrôleur de DMA, les bus, les contrôleurs des interfaces.

Dans ce chapitre nous allons détailler certains de ces composants.

7.1 La mémoire

7.1.1 Définitions

La mémoire permet d'enregistrer et de sauvegarder des informations. Outre la mémoire de masse (disques), l'ordinateur dispose d'une mémoire de travail, mémoire vive ou RAM (*Random Access Memory*) qui peut-être lue ou écrite par opposition à la mémoire morte (ROM) qui ne peut que se lire. On distingue deux catégories de mémoires vives : mémoires vives statiques et mémoires vives dynamiques.

7.1.2 Mémoires vives statiques SRAM

Les SRAMs (*Static Random Access Memory*) sont utilisées pour des mémoires de faibles capacités (mémoire cache) car très chères. Par contre, l'information peut être conservée longtemps sans rafraîchissement et les accès sont très rapides (autour de 15 ns).

7.1.3 Mémoires vives dynamiques

- La mémoire `DRAM` (*Dynamic Random Access Memory*)
  C'est la `RAM` classique qui a été utilisée pendant de longues années. La vitesse varie de 100 ns à 60 ns selon l'ordinateur dans lequelle elle se trouve : des rafraîchissements réguliers devant être effectués (lecture/écriture des informations). Actuellement la `DRAM` est de plus en plus dépassée.
- La mémoire `RAM EDO` (*Extended Data Out*)
  Elle est aussi appelée `HPM DRAM` (*Hyper Page Mode DRAM*). Les composants de cette mémoire permettent de conserver plus longtemps l'information, on peut donc ainsi espacer les cycles de rafraîchissement La `RAM EDO` est une version améliorée de la `DRAM`, ce qui permet d'atteindre des vitesses allant de 60 à 40 ns.
- La mémoire `RAM BEDO` (*Burst Extended Data Out*)
  C'est une évolution de la `RAM EDO` où les lectures et les écritures sont effectuées en mode rafale. On n'adresse plus chaque unité de mémoire. On se contente de transmettre l'adresse de départ du processus de lecture/écriture et la longueur du bloc de données (*Burst*). Ce procédé permet de gagner beaucoup de temps, notamment avec les grands paquets de données. La `RAM BEDO` n'est supportée que par peu de *chipsets*.
- La mémoire `SDRAM` (*Synchronous DRAM*)
  La `SDRAM` échange ses données directement avec le processeur en se synchronisant avec lui; ce qui permet d'éviter des états d'attente. C'est la `RAM` en vogue actuellement : les temps d'accès sont de l'ordre de 10 ns et les coûts ont baissé.
- La mémoire `RDDAM` (*Rambus DRAM*)
  la `RDRAM` est une `DRAM` développée par la société Rambus avec des contrôleurs spécifiques lui permettant d'être 10 fois plus rapide qu'une `DRAM` classique

La `RAM`, vendue en barettes, se trouve actuellement en deux formats : `SIMM` (en perte de vitesse) et `DIMM` (le standard).

- `SIMM` (*Single In-line Memory Module*)
  Type de barettes 8 et 32 bits. Module mémoire à simple rangée de broches de connexion.
- `DIMM` (*Dual Inline Memory Module*)
  Type de barettes 64 bits normalisées.

Il existe aussi des barettes `RDRAM`. Ce sont des barrettes mémoires séries constituées de composants 16 bits. Ce type de mémoire devrait bientôt se généraliser.

7.1.4 Rappels sur les mesures de capacités mémoire

Les capacités mémoire s'expriment en octets. Pendant très longtemps, les unités utilisées en informatique violaient le système international standard. Par exemple, un méga-octet n'était pas égal à 1000000 octets, mais un peu plus.

Plusieurs propositions récentes vont dans le sens d'une plus grande uniformisation des significations des préfixes. Le NIST (*National Institute of Standards and Technology*) signale que l'IEC

(*International Electrotechnical Commission*) propose le standard suivant pour les unités binaires :

Facteur	Nom	Symbole	Origine	Dérivé de
2^{10}	kibi	Ki	kilobinaire : $(2^{10})^1$	kilo : $(10^3)^1$
2^{20}	mebi	Mi	megabinaire : $(2^{10})^2$	mega : $(10^3)^2$
2^{30}	gibi	Gi	gigabinaire : $(2^{10})^3$	giga : $(10^3)^3$
2^{40}	tebi	Ti	terabinaire : $(2^{10})^4$	tera : $(10^3)^4$
2^{50}	pebi	Pi	petabinaire : $(2^{10})^5$	peta : $(10^3)^5$
2^{60}	exbi	Ei	exabinaire : $(2^{10})^6$	exa : $(10^3)^6$

(cf. `http://physics.nist.gov/cuu/Units/binary.html`)

En anglais, Ki doit être prononcé « kibi », comme « kilo binaire ». Un Kio est donc prononcé un « kibioctet »...

Une autre proposition émane de Donald Knuth. Il propose les notations suivantes :

Facteur	Nom	Symbole	Origine	Dérivé de
2^{10}	?	KK	kilobinaire : $(2^{10})^1$	kilo : $(10^3)^1$
2^{20}	?	MM	megabinaire : $(2^{10})^2$	mega : $(10^3)^2$
2^{30}	?	GG	gigabinaire : $(2^{10})^3$	giga : $(10^3)^3$
2^{40}	?	TT	terabinaire : $(2^{10})^4$	tera : $(10^3)^4$
2^{50}	?	PP	petabinaire : $(2^{10})^5$	peta : $(10^3)^5$
2^{60}	?	EE	exabinaire : $(2^{10})^6$	exa : $(10^3)^6$

(cf. `http://www-cs-faculty.Stanford.EDU/~knuth/news.html`)

On pourrait ici parler de 1024 octets comme d'un KKo, ou d'un kilo binaire. Dans « KKo », il y a à la fois l'idée qu'un « KKo » est plus grand qu'un « Ko » et l'idée de quelque chose de binaire, puisque « K » apparait deux fois.

Dans tous les cas, quelle que soit la convention adoptée dans le futur, que ce soit l'une de celles-ci ou une autre, il n'en reste pas moins que l'on souhaite redonner aux préfixes un sens non ambigu.

Ainsi, lorsqu'on parlera de kilo-octets, il s'agira réellement de 1000 octets. Lorsqu'on parlera de méga-octets, il s'agira réellement de 1000000 octets, etc.

7.2 Les bus

7.2.1 Définitions

Un bus peut-être vu comme un ensemble de « fils » chargés de relier divers composants de l'ordinateur afin de transférer des informations. À l'intérieur du microprocesseur, on trouve les classiques bus de données, d'adresses et de commandes. Les bus d'extension permettent, à l'extérieur du microprocesseur, de faire communiquer les périphériques avec la mémoire et l'unité centrale. Les

cartes graphiques ou cartes réseaux sont également considérées comme des périphériques même si elles sont localisées à l'intérieur du boîtier. Il existe différents bus d'extension.

7.2.2 Les bus d'extension

1. Le bus `ISA` (*Industry Standard Architecture*) ou `PC-AT`
   - Apparu en 1984 avec `IBM PC-AT`.
   - Bus d'une largeur de 16 bits avec taux de transfert max de 8 Mo/s.
   - Configuration matérielle par positionnement de *switchs*.
2. Le bus `EISA` (*Extended ISA*)
   - Apparu en 1988.
   - Bus de 32 bits destiné à fonctionner avec les processeurs 32 bits.
   - Compatibilité avec bus `ISA` et taux de transfert jusqu'à 33 Mo/s.
3. Le bus `PCI` (*Peripheral Component Interconnect*)
   - Bus 32 bits développé par `Intel` en 1993.
   - Taux de transfert de 132 Mo/s.
   - Indépendant du processeur et disposant de sa propre mémoire.
   - Autoconfigurable.
4. Le bus `USB` (*Universal Serial Bus*)
   - Bus série récent et évolué.
   - Permet d'utiliser 127 périphériques : souris, clavier, imprimantes, scanner, ... chaînés sur un même canal via un mini-hub.
   - Reconnait automatiquement le périphérique branché.
   - Fonctionnement similaire à celui des réseaux locaux en bus.
   - Débit de 1,5 Mo/s, destiné aux périphériques lents.

   Pour fonctionner, les périphériques `USB` doivent être reconnus par le `BIOS`.
5. Le bus graphique `AGP` (*Accelerated Graphics Port*)
   - Bus 32 bits récent 1997, spécialisé dans l'affichage.
   - Relie directement le processeur de la carte graphique avec le processeur de l'Unité Centrale et avec la mémoire vive.
   - Débits atteignant de 264 Mo/s pour `AGP` de base et de 528 Mo/s pour `AGP 2x` et supérieur à 1 Go/s pour `AGP 4x`.

7.3 Les contrôleurs/interfaces de périphériques

7.3.1 Définitions

Les périphériques sont reliés au processeur au travers du bus. Certains périphériques se présentent sous la forme d'une carte qui s'enfiche dans un connecteur de bus. Certaines cartes gèrent à leur tour un bus et s'appellent alors contrôleur.

Le rôle, par exemple, d'un contrôleur (ou de l'interface) de disque dur est de transmettre et de recevoir des données en provenance du disque dur. La vitesse de transfert des données entre le disque dur et l'ordinateur dépend du type d'interface utilisé. Chaque contrôleur d'interface offre des performances différentes.

Pour les performances d'un disque dur, le paramètre le plus important est le taux de transfert de données. Le taux de transfert correspond à la vitesse à laquelle les informations passent du périphérique à l'ordinateur.

7.3.2 L'interface IDE

L'acronyme IDE, *Integrated Drive Electronics*, est le nom donné aux disques durs qui intègrent leur contrôleur sur le disque. L'interface d'un disque IDE s'appelle officiellement ATA (Advanced Technology Attachment). Ce contrôleur fait partie des standards adoptés par l'ANSI. Il est apparu en 1986 et est encore très répandu. Il permet de gérer des disques de 20 à 528 Mo avec des taux de transfert de l'ordre de 8.3 Mo/s.

Les caractéristiques de l'interface ATA sont devenues une norme ANSI en mars 1989. Les standards ATA 1 et ATA 2 (également appelé *Enhanced IDE*), ont été approuvés respectivement en 1994 et 1995. Ces standards ont permis d'éliminer les incompatibilités et les problèmes que pose l'interfaçage de disques durs IDE pour des ordinateurs utilisant un bus ISA ou EISA.

Actuellement, le contrôleur IDE est délaissé au profit du mode EIDE (ATA-2 ou Fast-ATA-2) qui permet de gérer deux périphériques : un rapide (disque) et un lent (CD-ROM).

ATA-4 ou Ultra ATA (EIDE DMA-33 ou plus souvent Ultra DMA) est intégré sur les dernières cartes mère à base de Pentium. Il permet de piloter quatre périphériques (deux rapides et deux lents). Il existe des cartes Ultra ATA PCI pour PC.

7.3.3 L'interface SCSI

L'interface SCSI (*Small Computer System Interface*) ou contrôleur SCSI adopté par les constructeurs comme moyen de piloter les disques durs ou des lecteurs de CD-ROM sur des machines haut de gamme. La vitesse de transfert varie de 5 Mo/s à 80 Mo/s selon la largeur du bus et le standard SCSI (1/2/3):

Protocole	Standard	Largeur du bus (en bits)	Débit (Mo/s)	Nombre de périphériques
Regular	SCSI 1	8	5	7
Fast SCSI	SCSI 2	8	10	7
Wide SCSI	SCSI 2	16	10	15
Fast/Wide SCSI	SCSI 2	16	20	15
Ultra SCSI	SCSI 3	8	20	7
Ultra Wide SCSI	SCSI 3	16	40	15
Ultra 2 SCSI	SCSI 3	16	80	15

L'interface *FireWire* est un bus série de haute vitesse qui a été créé à l'origine par Apple Computers en 1986. En 1995 la norme a reçu la certification `IEEE 1394` :

- Utilisé pour des périphériques rapides (imagerie, PAO, ...). Il est dédié aux disques externes et aux applications de montage vidéo numérique. Il permet par exemple de transmettre des séquences vidéos numériques (depuis une caméra vidéo numérique, un magnétoscope ou une télévison numérique) directement à un ordinateur.
- Se présente sous la forme d'une carte d'extension
- Similitude avec `USB` mais débit 10 à 20 fois supérieur. Il permet d'atteindre de taux de transfert de 12,5 Mo/s, 25 à 50 Mo/s (de 100, 200 à 400 Mbit/s).
- L'interface offre la possibilité de relier jusqu'à 63 périphériques, sans hub et de manière autoconfigurable.

7.4 Le traitement des interruptions

Les interruptions permettent la communication entre les logiciels et le matériel. Le périphérique signale qu'il a fini une tâche en émettant une interruption. Le système d'exploitation sait quand le travail est terminé et peut, en attendant, se consacrer éventuellement à d'autres tâches.

Une interruption arrête un programme en cours d'exécution pour traiter une tâche plus urgente. Quand la tâche est terminée, le processus interrompu est repris à l'endroit où il avait été arrété. Comme exemple d'interruptions, on peut citer : les alarmes, les périphériques prêts à lire/écrire, ...

Le déroulement d'une interruption est le suivant :

- Réception par l'unité centrale d'une demande d'interruption ;
- Acceptation ou rejet de cette demande ;
- Fin de traitement de l'instruction en cours ;
- Sauvegarde de l'état du système ;
- Exécution du programme relatif à l'interruption ;
- Restauration de l'état du système ;
- Reprise du programme interrompu.

On distingue trois types d'interruptions :

- Interruption externe et matérielle par un périphérique
- Interruption externe et matérielle par un programme
- Interruption interne (exceptions, traps) par le processeur : division par zéro, dépassement de capacité

Il y a également une hiérarchisation parmi les interruptions :

- Interruptions non masquables : ne peuvent être interdites. Elles sont prioritaires.
- Interruptions masquables : elle peuvent être ignorées ou retardées ; afin d'éviter des interruptions d'interruptions en cascade.

7.5 Accès aux périphériques : les DMAs

Pour piloter du matériel (carte réseau, carte son, carte SCSI, ...), on utilise des programmes écrits spécifiquement. Ils sont appelés pilotes ou *drivers*. On les trouve souvent suffixés par `.drv`, `.vxd` ou `.dll`.

L'échange de données entre le matériel et les pilotes de périphériques (ou `drivers`) peut poser des problèmes de performance quand la quantité de données devient importante. Il serait suceptible de générer beaucoup d'interruptions. Les canaux `DMA` (*Direct Memory Access*), au nombre en général de 8, sont gérés par un contrôleur intégré à la carte mère. Ils permettent de communiquer directement avec la mémoire sans monopoliser le processeur. Une fois les informations transmises, le circuit de `DMA` charge en mémoire les données du périphérique relié au canal. Lorsque le transfert est terminé, le périphérique émet une interruption. Pendant le transfert, le processeur est libre.

Chapitre 8

Systèmes de gestion de fichiers

Le contenu de ce chapitre a été rédigé à partir notamment du livre de David Solomon [5] et de celui de P.-A. Goupille [3].

8.1 Introduction

Pour rappel, les systèmes de gestion de fichiers (SGFs) permettent d'organiser les données sur les disques. On peut rencontrer plusieurs types de SGFs :

- celui d'UNIX que vous avez vu dans les cours précédents ;
- celui des PCs dans le monde DOS : FAT (*File Allocation Table*) ;
- celui utilisé sur OS/2 (IBM) : HPFS (*High Performance File System*) ;
- celui utilisé sur Windows NT : NTFS (*Windows NT File System*).

Il est possible de faire coexister différents SGFs sur un même disque physique.

On crée pour cela des partitions. Ensuite chaque partition peut être formatée pour y installer le SGF voulu. On distingue en général deux types de partitions : les partitions primaires et étendues.

La partition primaire ou principale Elle est reconnue par le BIOS (*Basic Input Output System*) comme pouvant être amorçable ou *bootable*, i.e. le système peut démarrer à partir de cette partition. On peut créer jusqu'à 4 partitions principales (il n'y a pas alors de partition étendue) ; cependant sous MS-DOS, il ne peut y avoir qu'une seule partition primaire (appelée `PRI-DOS`).

La partition étendue, non *bootable* On peut la diviser en plusieurs disques logiques auxquels sont attachés des lecteurs (`F:`, `G:`, ...). Sous MS-DOS elle est appelée `EXT-DOS`.

La table des partitions se trouve dans un enregistrement spécial appelé MBR (*Master Boot Record*) rangé sur le premier secteur du disque. Elle contient des informations relatives aux partitions primaires. Les informations sont sous la forme :

```
stuct partable_entry {
```

```
    unsigned char boot_flag;      /* 0 inactive , 0x80 active */
    unsigned char begin_head;
    unsigned char begin_sector;
    unsigned char begin_cylinder;
    unsigned char partition_type;   /* 0x83 = Linux, 0x82 = Swap Linux */
    unsigned char end_head;
    unsigned char end_sector;
    unsigned char end_cylinder;
    unsigned char start_sector;
    unsigned char sector_count;
}
```

Il ne peut y avoir qu'une seule partition primaire active (i.e. celle depuis laquelle l'ordinateur démarre).

En plus de cette table, le MBR contient, sur les 446 premiers octets, un mini-programme de démarrage ou IPL (*Initial Program Loader*) qui cherche dans la table de partitions le secteur de début de la partition « active », et charge en mémoire le secteur de démarrage du système d'exploitation correspondant. Si on dispose de plusieurs systèmes d'exploitation, il faut un programme qui demande à l'utilisateur, au moment du boot sur quel système d'exploitation il veut démarrer. Par exemple avec Linux, il y a le *lilo* ou *Linux Loader* qui peut aussi être intégré dans le MBR. Avec Windows NT, on peut modifier le `boot.ini` (avec le programme `bootpart.exe`).

8.2 Le système de gestion de fichier FAT

Le système FAT est rencontré sur la plupart des PCs ; il a été conçu à l'origine pour des disques de faible capacité ou des disquettes.

Avec un système FAT, le disque dur est composé de *clusters*[1] qui sont des groupes de secteurs contigus. Un disque formaté avec FAT est divisé en *clusters*, dont la taille est déterminée par la taille du volume. Tous les *clusters* ont la même taille. Un fichier doit être contenu sur un nombre entier de *clusters*. On est ainsi confronté au *slack space problem* : l'espace inutilisé à la fin d'un *cluster* ne peut pas être utilisé par d'autres fichiers. Par exemple, soit un *cluster* composé de 8 secteurs de 512 octets : 4096 octets. Si on a un fichier de 10 octets, on perd 4086 octets, ... Avec un disque FAT, il faut éviter les petits fichiers ou bien utiliser la compression du disque.

Différents types de FATs existent : `FAT12`, `FAT16`, `FAT32` qui définissent le nombre de *clusters* que l'on peut adresser. Avec une `FAT12` on dispose de 12 bits, soit 4096 *clusters*, ce qui concerne de petits volumes comme les disquettes. Avec une `FAT16` on peut référencer 65536 *clusters* par volume. On parle aussi de `VFAT` (Virtual FAT ou FAT étendue) qui permet de gérer des noms longs.

Un système FAT comprend :

- un enregistrement d'amorçage avec la table des partitions (sur les disques durs) ;
- une zone réservée au secteur de chargement (secteur de Boot) ;

1. En anglais, groupe.

- un exemplaire de la FAT ;
- une copie optionnelle de la FAT ;
- le répertoire principal (*Root Directory*) ou dossier racine ;
- la zone des données et sous-répertoires.

8.2.1 Le secteur de BOOT

Il contient des informations sur le format du disque et un programme chargeant le DOS en mémoire (*BOOT strap*).

8.2.2 La FAT

Le système de fichier FAT est caractérisé par la table d'allocation de fichiers (*File Allocation Table*, FAT) qui est une table qui réside en haut du volume. Pour protéger le volume, deux copies de la FAT sont gardées au cas où l'une est endommagée. En plus, les tables FAT et le répertoire racine doivent être rangés dans un endroit fixe de façon à ce que les fichiers de boot du système puissent être localisés correctement.

À la création d'un fichier, une entrée est créée dans le répertoire et le numéro du premier *cluster* contenant des données est établi. Les entrées 0 et 1 sont réservées et contiennent des informations spécifiques. Les entrées suivantes indiquent l'état du *cluster* correspondant. Une valeur de 0 spécifie que le *cluster* représenté par l'entrée de la FAT est libre ; une valeur de FFFF indique que le *cluster* est le dernier de la chaîne et une autre valeur pointe sur le numéro du prochain *cluster* qui contient les données du fichier.

Mettre à jour la table FAT est très important et prend beaucoup de temps. Si la table FAT n'est pas mise à jour régulièrement, cela peut provoquer des pertes de données. Cela prend beaucoup de temps car les têtes de lecture du disque doivent être repositionnées sur la piste logique zéro du lecteur chaque fois que la table FAT est mise à jour.

Il n'y a pas d'organisation pour la structure du répertoire FAT, et la première allocation disponible sur le lecteur est allouée aux fichiers. En plus FAT supporte seulement les attributs de fichiers lecture seule, caché et archive.

8.2.3 Le répertoire principal (*Root Directory*)

Pour les volumes `FAT12` et `FAT16`, il se situe immédiatement derrière la deuxième copie de la FAT et ne peut être étendu. Sa taille est déterminée au formatage.

Sur les volumes `FAT32`, le répertoire racine est stocké comme les fichiers ordinaires ; un champ dans le secteur de Boot indique le *cluster* initial.

Un répertoire est divisé en petites structures appelées entrées de répertoires qui contiennent pour chaque fichier ou répertoires : le nom, les attributs, la taille, la date, le temps et le numéro du *cluster* initial.

8.2.4 Le Nommage

FAT utilise la convention de nom traditionnelle 8 + 3 et tous les noms de fichiers doivent être créés avec le jeu de caractères ASCII. Le nom d'un fichier ou d'un répertoire peut avoir jusqu'à 8 caractères de long, puis un point (.) en séparateur, puis jusqu'à 3 caractères d'extension. Le nom doit commencer avec une lettre ou un chiffre et peut contenir tous les caractères sauf les caractères suivants : `. " / \ [ ] : ; | = ,`.

Si l'un de ces caractères est utilisé, des résultats inattendus peuvent survenir. Le nom ne peut contenir d'espaces. Les noms suivants sont réservés : `CON`, `AUX`, `COM1`, `COM2`, `COM3`, `COM4`, `LPT1`, `LPT2`, `LPT3`, `PRN`, `NUL`.

Tous les caractères seront convertis en capitales.

Les noms de fichiers longs sont supportés par la `VFAT` et sont stockés en `UNICODE`, successeur 16 bits de l'`ASCII`, dans une entrée `LFN` (*Long File Name*). Chaque entrée peut contenir 13 caractères ; si le nom est plus long une autre entrée est créée.

8.2.5 Conclusion

Il vaut mieux lors de l'utilisation de lecteurs ou de partitions de plus de 200 Mo ne pas utiliser le système de fichier FAT. En effet, avec l'augmentation de la taille du volume, les performances de FAT vont diminuer rapidement. Il n'est pas possible d'établir des permissions sur des fichiers qui sont sur des partitions FAT. De plus, les partitions FAT sont limitées en taille à un maximum de 4 Go sous Windows NT et 2 Go sous MS-DOS.

8.3 Windows NT

8.3.1 Historique

Windows NT est un système d'exploitation récent dont les premières spécifications remontent à 1989. En fait, l'histoire de Windows NT date du début des années 80 quand Microsoft a travaillé avec IBM pour créer un successeur plus puissant à DOS et qui fonctionnerait sur les Intel x86. Le résultat fut OS/2. Parallèlement au développement d'OS/2, Microsoft était en train de travailler sur un nouveau système d'exploitation plus puissant que le système Windows. Ce système « New Technology » devait fonctionner sur des plate-formes avec différents processeurs. Fin Octobre 1988, Microsoft embaucha David Cutler qui était un gourou de chez Digital Equipment Corporation (maintenant Compaq) pour les aider à concevoir leur nouveau système. Le nom d'origine était OS/2 NT car dans le même temps, Microsoft était en train d'aider à développer OS/2 et d'intégrer des parties dans son nouveau système d'explotation (NT). Au début des années 1990, Bill Gates critiqua NT le trouvant trop gourmand et trop lent durant une *review*. La décision fut prise en 1991 de baser NT (API, interface utilisateur) sur le système Windows courant de Microsoft, version 3.0 et non plus sur OS/2. Le nom OS/2 NT fut supprimé et le nouveau nom devint Windows NT. Bill Gates et son équipe Windows NT, dirigée par David Cutler, poussa en avant le développement de NT. Microsoft

coupa les ponts avec IBM ainsi que leur développement sur OS/2. Le codage et le test de NT ont continué dans les mois suivants et la version Windows NT 3.1 fut disponible le 17 Juillet 1993.

Bien que ce fut la première version de Windows NT, Microsoft pris la décision de la nommer version 3.1 au lieu de 1.0 afin de l'intégrer, d'une certaine manière avec Windows 3.1, qui était déjà sur le marché. La version 3.5 de Windows NT suivit peu de temps après. Une révision majeure, version 4.0, fut commercialisée en août 1996 avec l'interface utilisateur de Windows 95. La prochaine version de Windows NT, Windows 2000 ou Windows NT 5.0, est actuellement en bêta version et promet de supporter de nouvelles technologies émergentes. Windows NT 4.0 est venu en deux versions : NT Server et NT Workstation. NT Server peut être utilisé comme serveur de fichiers sur un réseau local ou comme serveur internet en fournissant le courrier électronique, le web, le ftp et toutes les combinaisons de services basés sur TCP. NT Workstation est un système d'exploitation 32 bits qui agit comme un client parfait vers le serveur NT et qui convient également pour tout ordinateur personnel.

8.3.2 Caractéristiques

La portabilité Le système peut s'exécuter sur différentes plate-formes matérielles. Dans cet objectif, Windows NT a été écrit entièrement en C et C++ (16 millions de lignes de code). Microsoft a isolé la part du système qui est spécifique au matériel dans une couche appelée HAL (*Hardware Abstraction Layer*). À chaque nouvelle architecture, il suffit de recompiler le code source et de créer une nouvelle HAL. Windows NT fonctionne sur les architectures Intel x86, MIPS RISC, Digital Alpha et Motorola PowerPC RISC[2].

La sécurité. Le gouvernement américain a donné la certification de niveau C2 à `NT`. Ce standard C2 fournit :

- une protection de contrôle d'accès pour tous les objets partageables (fichiers, répertoires, processus, ...). Elle permet au propiétaire d'une ressource de déterminer qui peut avoir accès à la ressource et de quelle manière (lecture, écriture). Il y a cependant aussi besoin d'un contrôle d'accès privilégié (si le propriétaire n'est plus joignable) par l'intermédaire d'un administrateur.
- une surveillance de la sécurité qui offre la possibilité de détecter et d'enregistrer des événements relatifs à la sécurité ainsi que des tentatives pour créer, accéder ou détruire des ressources système.
- une protection de la mémoire qui empêche des processus non autorisés d'accéder à la mémoire d'autres processus. De plus Windows NT garantit que lorsqu'une page de mémoire physique est allouée à un processus utilisateur, cette page ne contiendra pas des données provenant d'un autre processus.
- l'authentification des mots de passe au login. Quand vous tapez la séquence `Ctrl-Alt-Del`, aucun autre processus (autre que le `WinLogon`) ne peut accéder aux données entrées. Cela pour se protéger contre les programmes qui peuvent essayer de voler les mots de passe en faisant des captures d'écran.

Le support du multi-tâches. Le multi-tâches permet à un ordinateur d'exécuter plus d'une tâche en même temps. Comme un processeur ne peut pas travailler sur plusieurs choses à la fois,

2. Windows 2000 devrait fonctionner uniquement sur des processeurs x86 et Digital Alpha.

il faut partager le processeur. Windows NT met les tâches dans des files d'attente ; chacune ayant un niveau de priorité. NT a 32 niveaux de priorité différents (de 0 à 31). Ensuite en se basant, entre autres, sur cette information, le système fait un peu de la tâche 1, un peu de la tâche 2, un peu de la tâche 3, et à nouveau un peu de la tâche 1, etc. Cela donne l'illusion que l'ordinateur est en train de faire plusieurs choses à la fois. Les tâches sont bien isolées en mémoire et les programmes ne sont pas autorisés à utiliser les zones mémoires du système, ni celles des autres programmes.

Le support du multi-processeurs En utilisant une technique appelée *Symmetrical Multiprocessing* (SMP), Windows NT est capable d'utiliser plusieurs processeurs sur le même système et d'affecter n'importe quelle tâche à n'importe quel processeur. Des versions spéciales de Windows NT ont la possibilité de supporter jusqu'à 32 processeurs. Microsoft ne supporte pas officiellement NT Server avec plus de 8 processeurs (pour NT Server Enterprise) et 4 processeurs pour NT Server.

Le support de RAID Windows NT supporte une caractéristique matérielle appelée RAID (*Redundant Array of Inexpensive Disks*). RAID permet une grande capacité de stockage, accroît les performances et surtout la fiabilité. À un niveau faible, RAID est capable de faire une copie miroir d'un disque. À des niveaux plus élevés, RAID gardera 2 bits de chaque octets sur des disques différents. Windows NT utilise des disques `SCSI` pour implanter le RAID.

La stabilité et la robustesse Plus d'attention a été donnée à la stabilité de Windows NT 4.0. C'était essentiel que NT soit stable pour concurrencer sérieusement UNIX. Windows 95 et 98 sont notoirement instables et donc inacceptables pour du matériel de haute performance (plusieurs processeurs, des gigaoctets de RAM), et pour des applications réseaux très surchargées (serveurs WEB).

8.3.3 Quelques définitions

Win32 API (*Application Programming Interface*) Ensemble de fonctions documentées décrivant l'interface de programmation de la famille Microsoft (Windows NT, Windows 9x) et utilisées par un développeur d'application.

DLL (*Dynamic-link library*) Ensemble de routines qui sont dynamiquement chargées par les applications qui les utilisent.

Base de registres On ne peut pas parler de la structure interne de NT sans passer par la base de registres (`regedit`). Elle contient les informations nécessaire pour booter et configurer le système, la base de données de sécurité et les profils par utilisateurs.

Windows NT Resource Kits Ce sont des *packages* essentiels pour des utilisateurs expérimentés, des administrateurs et même des développeurs. En plus d'outils utiles pour l'affichage de l'état du système interne, ces kits contiennent des documentations internes dans le *Windows NT Resource Guide*. Ce guide traite de l'architecture du système, de la structure de la base des registres et du système de fichier, de la gestion des performances, ...

Programme, processus et *threads* Windows NT est un système multi-tâches qui différencie les notions de programme, processus et *thread*. Un *programme* est une séquence statique d'instructions. Un *processus* est un ensemble de ressources réservées pour les *threads* qui exécutent le programme.

Un processus NT comprend :

- un programme exécutable avec du code et des données ;
- un espace d'adressage virtuel privé que peut utiliser le processus ;
- des ressources systèmes (sémaphores, descripteurs de fichiers, ...) que le système d'exploitation alloue au processus quand les *threads* les ouvrent durant l'exécution du programme ;
- un identificateur unique *process ID* (ou en interne, *client ID*).

Un *thread* est l'entité dans un processus que Windows NT ordonnance pour l'exécution. C'est l'unité d'ordonnancement et d'exécution pour Windows NT. Il comprend :

- le contenu d'un ensemble de registres représentant l'état du processeur ;
- deux piles, une pour le *thread* quand il s'exécute en mode noyau et une quand il s'exécute en mode utilisateur ;
- une zone utilisée par les librairies (ou routines) ;
- un identificateur appelé *thread ID* (aussi *client ID* en interne).

Les trois premiers points représentent le contexte du *thread*. Les *threads* ont leur propre contexte d'exécution ; chaque *thread* dans un processus partage l'espace d'adressage virtuel du processus (en plus du reste des ressources appartenant au processus).

8.3.4 NTFS

En 1988, Microsoft supportait deux systèmes de fichiers : le système FAT et HPFS. On a déjà vu que le système FAT montrait ses limitations pour gérer de larges configurations de fichiers. Le système d'exploitation OS/2 avait introduit HPFS pour faire face aux limitations de FAT et avait amélioré les temps d'accès aux fichiers pour de grands répertoires. Mais HPFS ne présentait pas de garanties pour des applications très critiques notamment en termes de reprises sur erreurs, tolérance aux pannes, sécurité. Il a fallu donc créer un nouveau système de fichier appelé NTFS (*Windows NT File System*). Windows NT peut certes fonctionner sur des système FAT16 mais de manière moins optimisée. Par contre Windows NT 4.0 ne peut être installé sur un système FAT 32. [3]

NTFS gère ainsi :

La reprise sur erreur NTFS fournit un système basé sur un modèle transactionnel (principe du tout ou rien). Si un programme modifie, par une opération d'entrée/sortie, la structure de NTFS (changement de la structure d'un répertoire, allocation d'espace pour un nouveau fichier), NTFS traite cette opération de manière atomique. De plus NT utilise un espace de stockage redondant pour sauvegarder des informations vitales du système. NTFS est un système de fichier *restaurable* car il garde trace des transactions par rapport au système de fichier. Quand un `CHKDSK` (*check disk*) est réalisé sur FAT et HPFS, la cohérence des pointeurs dans le répertoire, les allocations et les tables de fichiers sont vérifiées. Sous NTFS, un *log* de transactions par rapport à ces composants est maintenu de façon à ce que `CHKDSK` ait seulement besoin de repasser les transactions depuis le dernier *commit point* de façon à retrouver la cohérence dans le système de fichier.

3. Cela devrait être possible avec NT 5.0.

La tolérance aux pannes NTFS peut communiquer avec un pilote de tolérance aux pannes qui à son tour communique avec un pilote de disque matériel pour écrire les données sur le disque. Ce pilote de tolérance aux pannes peut dupliquer les données d'un disque vers un autre. Ce support est appelé *RAID level 1.* D'autres niveaux de disques RAID peuvent être supportés comme le niveau 5 (le pilote peut reconstruire le contenu d'un disque par des opérations de OU logique sur des données réparties sur trois ou plusieurs disques).

La sécurité Un fichier ouvert est implanté comme un objet fichier avec un descripteur de sécurité stocké sur le disque comme une part du fichier. Avant qu'un processus puisse ouvrir un objet fichier, le système de sécurité NT vérifie que le processus a l'autorisation pour le faire. Un utilisateur peut ajouter à un fichier des attributs qu'il aura lui-même déterminés.

NTFS revient également au concept FAT des *clusters* de façon à éviter le problème de HPFS d'une taille de secteur fixée. Ceci a été fait car Windows NT est un système d'exploitation devant être portable et que des technologies disques différentes peuvent être rencontrées. C'est pourquoi on a considéré que 512 octets par secteur risquaient de ne pas être toujours adaptés à l'allocation. La taille du *cluster* est définie comme un multiple de la taille d'allocation naturelle du matériel. Enfin, dans NTFS tous les noms de fichiers sont basés sur UNICODE et les noms de fichiers $8 + 3$ sont conservés en même temps que les noms de fichier longs.

Conventions de nommage de NTFS Les noms de fichiers et de répertoires peuvent avoir jusqu'à 255 caractères de long, y compris l'extension. Les noms conservent la casse (majuscule ou minuscule) mais n'y sont pas sensible. NTFS ne fait aucune différence sur les noms de fichiers par rapport à la casse. Les noms peuvent contenir tous les caractères sauf les suivants : `? " / \ < > * | :`.

À l'heure actuelle, à partir de la ligne de commande, vous pouvez seulement créer des noms de fichier jusqu'à 253 caractères.

Inconvénients de NTFS

- Il n'est pas recommandé d'utiliser NTFS sur un volume qui est plus petit que 400 Mo environ en raison de la déperdition d'espace. Cette déperdition d'espace provient de la forme des fichiers système de NTFS qui généralement utilisent au moins 4MB d'espace disque sur une partition de 100MB.
- À l'heure actuelle, il n'y a pas de cryptage de fichiers construit dans NTFS. C'est pourquoi quelqu'un peut booter sous MS-DOS ou un autre système d'exploitation et utiliser un utilitaire disque d'édition de bas niveau pour voir les données stockées dans un volume NTFS.
- Il n'est pas possible de formater des disquettes avec le système de fichier NTFS. Windows NT formate toutes les disquettes avec le système de fichier FAT car la déperdition causée par NTFS ne tiendrait pas sur une disquette.

8.3.5 La structure interne de NTFS

Le pilote NTFS se trouve dans la partie gestionnaire des entrées sorties et interagit avec 3 autres composants de NT proches du système de fichiers :

- Le service du fichier journal LFS (*Log File Service*) est une partie de NTFS qui maintient un journal des écritures sur disques. Ce fichier est utilisé pour restaurer un volume formaté NTFS en cas de panne du système.
- Le gestionnaire de cache et la mémoire virtuelle. Le cache fournit une interface spécialisée au gestionnaire de mémoire virtuelle de Windows NT. Quand un programme essaie d'accéder à une partie d'un fichier qui n'est pas dans le cache (on parle de *cache miss*) le gestionnaire de la mémoire appelle NTFS pour accéder au pilote du disque et obtenir le contenu du fichier. Le gestionnaire de cache optimise les entrées/sorties du disque en utilisant un ensemble de *threads* système qui appellent le gestionnaire de mémoire pour vider le contenu du cache vers le disque en tâche de fond.

NTFS utilise le modèle objet de Windows NT en implantant les fichiers comme des objets. Cette implantation permet aux fichiers d'être partagés et protégés par le gestionnaire d'objets. Une application crée ou accède à un fichier au moyen d' *object handles*. Quand la demande d'entrées/sorties arrive au pilote NTFS, le gestionnaire d'objets et le système de sécurité ont déjà vérifié que le processus appelant peut accéder au fichier. Le système de sécurité a analysé la liste de contrôle d'accès associée au fichier. Ensuite le gestionnaire d'entrées/sorties transforme le *handle* de fichier en un pointeur sur un objet fichier qui pointe à son tour sur un SCB (*Stream Control Block*). Tous les SCBs d'un même fichier pointent sur une structure commune appelée FCP (*File Control Block*) qui contient la référence du fichier permettant de localiser le fichier dans la *Master File Table* (cf ci-dessous).

8.3.6 La structure d'un volume NTFS

Cette section explique comment l'espace disque est divisé et organisé en *clusters*, comment les fichiers sont structurés en répertoires et comment les données et les attributs des fichiers sont réellement stockés sur disque.

Les volumes

Un volume est une partition logique d'un disque qui est créée quand on formate un disque. Un disque peut avoir un ou plusieurs volumes. NTFS gère chaque volume de manière indépendante. Un volume est une série de fichiers plus de l'espace non alloué restant sur la partition du disque. Si dans le système FAT, un volume comporte des zones préalablement formatées, un volume NTFS stocke toutes les données y compris les répertoires, le *bootstrap* du système comme des fichiers ordinaires.

Les *clusters*

NTFS comme pour FAT utilise la notion de *cluster* comme l'unité d'allocation du disque. La taille du *cluster cluster factor* est établie quand un utilisateur formate le disque soit avec la commande `format` ou avec l'utilitaire d'administration de disque. La taille du *cluster* est toujours une puissance de 2 (512, 1024, 2048, ...). NTFS se réfère aux localisations physiques sur un disque au moyen d'un numéro de *cluster* logique ou LCN (*Logical Cluster Number*). Pour effectuer la conversion, NTFS multiplie le LCN par le *cluster factor*. NTFS se refère aux données d'un fichier au moyen d'un numéro de *cluster* virtuel ou VCNs (*Virtuel Cluster Number*) : les *clusters* appartenant à un fichier particulier sont numérotés de 0 à m. Les VCNs ne sont pas forcément contigus et sont mappés sur des LCNs.

La *Master File Table* ou MFT

La MFT est le cœur de la structure d'un volume NTFS. Elle est implantée comme un tableau d'enregistrement de fichier. La taille de chaque enregistrement est fixée à 1Ko quelle que soit la taille du *cluster*. Logiquement la MFT contient une ligne pour chaque fichier et une ligne pour la MFT elle-même. En plus de la MFT, chaque volume inclut un ensemble de fichiers méta-données (*metadata files*) qui contiennent les informations utilisées pour implanter la structure du système de fichiers.

Pour accéder à un volume, NTFS doit d'abord le monter, i.e. le préparer pour son utilisation. Pour cela NTFS recherche dans le fichier de boot où trouver l'adresse physique de la MFT. L'enregistrement de la MFT est le premier dans la table ; le second enregistrement pointe sur un fichier rangé au milieu du disque appelé le *MFT Mirror* qui est une copie partielle de la MFT. Une fois l'enregistrement de la MFT lu, NTFS obtient l'information de correspondance VCN vers LCN dans les attributs, la décompresse et la range en mémoire. Cela permet de savoir où sont rangés les enregistrements composant la MFT. Ensuite NTFS décompresse ces enregistrements pour plusieurs fichiers de méta-données et ouvre ces fichiers. Parmi ces fichiers de méta-données, on trouve :

- Le *log file*. NTFS utilise ce fichier pour enregistrer toutes les opérations qui affectent le volume NTFS y compris la création du fichier et toutes les commandes commme `copy`, ... Le *log file* est utilisé pour récupérer un volume NTFS après une panne éventuelle.
- Une autre entrée dans la MFT est réservée au répertoire racine (« / »). Quand NTFS veut ouvrir un fichier pour la première fois, il commence par chercher dans l'enregistrement MFT du répertoire racine. Ensuite quand le fichier a été ouvert, NTFS range la référence à l'enregistrement MFT du fichier pour pouvoir y accéder directement la prochaine fois.
- NTFS enregistre l'état de l'allocation du volume dans un fichier *bitmap*. Chaque bit de ce fichier représente un *cluster* identifiant si le *cluster* est libre ou non.
- Le fichier boot range le code de *boot strap* de Windows NT.
- NTFS maintient aussi un *bad cluster file* (fichier de mauvais *clusters*) pour enregistrer les problèmes sur les différents *clusters* du disque.
- Un fichier *volume* contient le nom du volume, la version de NTFS pour laquelle le volume est formaté et un bit qui précise qu'une vérification du disque doit être effectuée.

– NTFS gère aussi un fichier contenant une table de définition des attributs qui définit les types d'attributs supportés sur le volume et indique s'ils peuvent être indexés et/ou récupérés après une panne.

Un fichier dans un volume NTFS est identifié par une valeur de 64 bits appelée la référence du fichier. Cette référence inclut le numéro du fichier qui correspond à la position de l'enregistrement du fichier dans la MFT − 1. Dans le cas où la taille du fichier excède celle d'un enregistrement de la table MFT, NTFS lui alloue des unités d'allocations supplémentaires appelées *runs* ou extensions (de 2Ko à 4Ko) et situées en dehors de la MFT. Ces extensions vont contenir les valeurs des attributs (par exemple les données) qui sont alors dits non-résidents. NTFS placera dans les attributs résidents dans la MFT les informations suffisantes pour permettre de localiser les attributs non-résidents.

Bibliographie

[1] Blaess (Christophe). – *Programmation système en C sous Linux*. – Paris, Eyrolles, 2000.

[2] Divay (Michel). – *Unix et les systèmes d'exploitation*. – Paris, Dunod, 2000.

[3] Goupille (Pierre-Alain). – *Technologie des ordinateurs et des réseaux*. – Paris, Dunod, 1998, 5e édition.

[4] Rifflet (Jean-Marie). – *La programmation sous UNIX*. – Paris, Ediscience international, 1997, 3e édition.

[5] Solomon (David A.). – *Inside Windows NT*. – Microsoft Press, 1998, 2nd édition.

[6] Welsh (Matt). – *Le système Linux* . – O'Reilly, 1997.

[7] Wrobel (Brigitte). – Cours de licence d'informatique, 1999. URL: http://www.loria.fr/~wrobel/.