Méthodes de programmation systèmes

UE n°NSY103

Notes de cours

Code de l'UE :	**NSY103**
Titre de la formation :	**Méthodes de programmation systèmes**
Ouvert :	Ouvert
Type de diplôme :	Unité de valeur CNAM.
Nombre d'heures :	55h (~18 + 1 cours de 3 heures)
Durée de la formation :	Un semestre - Période : deuxième semestre
Jours de cours	Lundi & Mercredi
Heures de cours	18h00-21h00
Lieu de la formation :	CUCES UNIVERSITES
Salle :	S05 LE LUNDI ET S18 LE MERCREDI (à confirmer)
URL :	http://cours.desvigne.org/
Animateur	Emmanuel DESVIGNE <*emmanuel@desvigne.org*>

1 Avant propos

Ce document contient les notes de cours de l'UE du CNAM « *NSY103 : méthodes de programmation systèmes* ».

Il a été élaboré à partir des connaissances acquises lors de ma formation initiale (DESS ingénierie des réseaux et systèmes, obtenu à Nancy en 1995), de diverses expériences professionnelles (je travaille actuellement comme responsable informatique à la maternité régionale de Nancy), à l'aide de la mine d'or que représente Internet, et à l'aide de ces quelques livres :

- Joëlle DELACROIX – Linux : programmation système et réseau, Dunod 2003 ;
- Andrew TANENBAUM – Systèmes d'exploitation, Pearsoneducation 2003 ;
- Jean-Marie RIFFLET – La programmation sous Unix - 3ème édition, Ediscience 1993 ;
- Jean-Marie RIFFLET – La communication sous Unix - 2ème édition, Ediscience 1994.

Dans ces conditions, difficile de revendiquer la paternité du présent melting pot. Aussi, ce document est mis sous licence libre GNU-FDL. Vous pouvez le copier, l'imprimer, le diffuser, voire l'apprendre librement.

Le programme officiel invite le formateur à ne voir que les systèmes Linux et Linux temps réel. Ces systèmes d'exploitation proposent effectivement tous les mécanismes décrits dans ce document. Ils sont stables, efficaces... Néanmoins, même s'ils ont une certaine préférence des personnes ayant rédigé le programme du CNAM pour des raisons éducatives (et du rédacteur du présent document, avouons-le), il est difficile de faire l'impasse des autres systèmes d'exploitation que le marché propose, comme les Unix, XXX-BSD, Mac OS (qui s'en inspire), et MS-Windows. C'est pourquoi, parfois, les philosophies de ces derniers OS seront indiquées, afin d'illustrer qu'il peut y avoir d'autres solutions/implémentations, et pour que le candidat acquière une culture générale des systèmes d'exploitation.

Ces systèmes d'exploitations sont eux-mêmes très largement écrit en C (et parfois en C++). De plus, les travaux dirigés, travaux pratiques, projets devront être rédigés dans ces langages. Aussi, le premier cours sera en grande partie réservé à une mise à niveau sur ces langages, afin de donner les pointeurs pour que tous les candidats possèdent des bases identiques. Si le volume (en nombre de pages) de cette révision est importante, s'agissant d'un pré-requis, le formateur ne doit pas y passer plus d'une séance. Cette mise à niveau pourra demander une quantité plus ou moins importante de travail personnel de la part des candidats.

2 Table des matières

3 Programme officiel

3.1 Public concerné et conditions d'accès

Avoir des bases sur le fonctionnement des systèmes d'exploitation (cette UE intervient dans des diplômes et certifications de niveau supérieur à Bac + 2).

3.2 Finalités de l'unité d'enseignement

3.2.1 Objectifs pédagogiques

Approches qualitative et quantitative des systèmes d'exploitation et de communication. Conception et fonctionnement des systèmes d'exploitation centralisés et répartis, spécificités des systèmes temps réels. Introduction à la programmation système.

Exemples dans les systèmes UNIX, LINUX et LINUX-RT

3.2.2 Capacité et compétences acquises

Savoir développer une application multi-processus utilisant des outils de communication et de synchronisation en C sous Linux/Unix.

Appréhender les mécanismes fondamentaux des systèmes d'exploitation Comprendre la problématique des systèmes temps réels et les particularités de ces systèmes

3.3 Organisation

3.3.1 Description des heures d'enseignements

Cours : 60 heures (Nancy : 51 heures).

3.3.2 Modalités de validation

Examen final (**pour Nancy :** projet = ¼, examen final = ¾, Cf. réforme 2006-2007).

3.4 Contenu de la formation

Introduction générale

- Structure des systèmes informatiques.
- Structure des systèmes d'exploitation.
- Spécificités des systèmes temps réel

Gestion de processus

- Processus : concepts, opérations sur les processus. Processus coopératifs, threads, communications inter-processus (tubes, files de messages, segments de mémoire partagée).

Ordonnancement de l'unité centrale

- Concepts et critères d'ordonnancement.
- Ordonnancement temps réel

Synchronisation de processus

- Section critique, sémaphores, problèmes classiques.

Interblocage, inversion de priorités

- Prévention, détection, correction, héritage de priorités...

Gestion de la mémoire

- pagination, segmentation. Mémoire virtuelle.

Systèmes de fichiers

- Interfaces des systèmes de fichiers et implémentation.

Systèmes distribués

- Structure des réseaux et structure des systèmes répartis. Programmation socket

Exemple d'un système : LINUX, LINUX-RT

3.5 Bibliographie

Auteur	Titre
Joëlle Delacroix*	Linux : programmation système et réseau, Dunod 2003
Nils Schaefer	*Programmation système sous Unix, sN Informatique*
Andrew Tanenbaum	*Systèmes d'exploitation, Pearsoneducation 2003*
Jean-Marie Rifflet	*La programmation sous Unix - 3ème édition, Ediscience 1993*
Jean-Marie Rifflet	*La communication sous Unix - 2ème édition, Ediscience 1994*

(*) : le livre de Joëlle Delacroix est le seul livre de la bibliographie officielle du CNAM.

3.6 Contacts

3.6.1 Coordonnées du secrétariat du responsable national :

Accès 37 0 36Case courrier : 432

Service d'Informatique cycle A - 2 rue Conté - Paris 3e

Tél : 01 40 27 27 02 - Fax : 01 58 80 84 93

Contact : Virginie Moreau et Françoise Carrasse
Courriel : sec-cycleA.informatique@cnam.fr

3.6.2 Coordonnées du responsable de cette formation à NANCY :

M. Emmanuel DESVIGNE

Tel : 06 33 65 13 35

Courriel : emmanuel@desvigne.org

URL des cours : http://cours.desvigne.org/

3.7 Nouveautés à partir de la session 2006-2007

3.7.1 Origine de la réforme

Suite à des discussions avec la CTI (commission des titres de l'ingénieur) et ses équivalents pour la Licence et les titres RNCP, le CNAM doit garantir que les programmes enseignés de partout en France sont identiques. Pour cela, il a été crée un "référentiel de cours" plus efficace que les simples descriptions d'UE en une page auxquelles nous étions habitués.

Un site "http://ne.pleiad.net" regroupe ces infos (à destination des formateurs) pour les 4 cours expérimentés cette année : RSX101, RSX102, NSY103, et NFP107.

3.7.2 Résumé du principe de la réforme

Les formateurs seront invités à déposer/proposer un sujet d'examen. Le coordinateur du cours pour le Nord-Est gèrera une "discussion" visant à faire émerger UN seul sujet pour tout le Nord-Est. Ce sujet est ensuite transmis au Professeur responsable du cours au niveau national (à Paris) qui donne son accord ou refuse le sujet en suggérant des modifications.

Il est demandé aux enseignants de suivre le plan du cours et de participer à la discussion avec les autres enseignants du Nord-Est faisant le même cours que vous au même semestre.

Pour vos cours, le référentiel est maintenant en place. Il va donc être plus facile de travailler dès le début en harmonie avec la réforme. Logiquement ce plan de cours doit être proche des vôtres. Il ne devrait varier que dans la chronologie et l'enveloppe horaire consacrée à chaque partie (représentative des questions qui seront posées et des points associés).

En résumé : le programme est imposé, mais le sujet doit faire l'objet d'une discussion commune.

3.7.3 Plan imposé par cette réforme pour le cours NSY103

Introduction
THEME 1 : Rappels d'architecture machine (interruptions, fonctionnement de caches) Structure des systèmes d'exploitation. Notions de bases (commutation de contexte, trappes, appels système)

Gestion de processus
THEME 2 : Processus : concepts, opérations sur les processus.
THEME 3 : Processus Linux : création d'un processus, recouvrement de code, fin de processus, états et structure d'un processus Linux. (fork, exec, wait, exit.)
THEME 4 : Concepts et critères d'ordonnancement de l'unité centrale. Algorithmes usuels. Notion de préemption. Ordonnancement sous Linux. Fonctionnement des signaux. Communication entre processus
THEME 5 : Communication centralisée : outils Linux (tubes anonymes, files de messages)
THEME 6 : Communication répartie : les sockets
THEME 7 : Les schémas de synchronisation : exclusion mutuelle, producteurs/consommateurs, sémaphores.

Gestion de la mémoire
THEME 7 : Pagination. Mémoire virtuelle. Segments de mémoire partagé sous Linux.

Systèmes de fichiers
THEME 8 : Interfaces des systèmes de fichiers et implémentation. Allocation du support de masse. Notions de répertoires, de partition. Gestion des accès disque (politique d'ordonnancement du bras)
THEME 9 : Systèmes de gestion de fichiers sous Linux (inode, structure d'un fichier Linux, structure d'une partition Linux, commandes Linux liées à la gestion de fichiers)

THEME 10 : Révision

Bibliographie
Joëlle Delacroix Linux : programmation système et réseau, Dunod 2003

Référentiel d'évaluation : L'évaluation de première et deuxième session est axée autour :
1/ d'un projet de mise en œuvre des outils de communication est donné à réaliser aux auditeurs. Ce projet conduit à la spécification et programmation d'une application multiprocessus simple communicant via les outils étudiés (tubes, MSQ, sockets, etc…).
Ce projet est obligatoire ; il compte pour un quart de la note finale de première et deuxième session.
2/ d'un examen écrit comptant pour ¾ de la note finale.

4 Mise à niveau en C/C++

La majorité des systèmes d'exploitation modernes sont écrits dans le langage C et/ou en C++. Les travaux pratiques/travaux dirigés de ce cours demanderont à connaître ces langages. Il serait irraisonnable d'espérer obtenir ce module NSY103 en faisant l'impasse de la maîtrise du C (et d'avoir quelques notions de C++).

Aussi, ce chapitre doit permettre à chacun de voir ou de revoir les principes et la syntaxe de ces langages.

4.1 Bref historique du C

Le C est un langage procédural conçu en 1972 par Dennis Richie et Ken Thompson, chercheurs aux Bell Labs, afin de développer un système d'exploitation : UNIX sur un DEC PDP-11.

En 1978, Brian Kernighan et Dennis Richie publient la définition classique du C dans le livre « The C Programming language ». Le C devenant de plus en plus populaire dans les années 80, plusieurs groupes mirent sur le marché des compilateurs comportant des extensions particulières.

En 1983, l'ANSI (American National Standards Institute) décida de normaliser le langage ; ce travail s'acheva en 1989 par la définition de la norme ANSI C. Celle-ci fut reprise telle quelle par l'ISO (International Standards Organization) en 1990.

4.2 La compilation

Le C est un langage compilé : le code (compréhensible par un être humain) doit être passé dans une moulinette (le compilateur) qui transformera ce code source en code exécutable directement par le microprocesseur (nous y reviendrons, c'est justement un point important de ce cour). Classiquement, la compilation se décompose en fait en 4 phases successives :

- **Le traitement par le préprocesseur (preprocessing) :** le fichier source est analysé par le « préprocesseur » qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers sources ...). Le résultat de ce prétraitement est toujours du C ;
- **La compilation :** la compilation traduit le texte généré par le préprocesseur en assembleur, c'est-à-dire en une suite d'instructions du microprocesseur qui utilisent des « *mnémoniques* » rendant la lecture possible par un être humain ;
- **L'assemblage :** cette opération transforme le code assembleur en un fichier binaire, c'est-à-dire en instructions directement compréhensibles par le processeur. Généralement, la compilation et l'assemblage se font dans la foulée, sauf si l'on spécifie explicitement que l'on veut le code assembleur. Le fichier produit par l'assemblage est appelé « fichier objet » ;
- **L'édition de liens :** un programme est souvent séparé en plusieurs fichiers source, pour des raisons de clarté mais aussi parce qu'il fait généralement appel à des librairies de fonctions standard déjà écrites. Une fois chaque code source assemblé, il faut donc lier entre eux les différents fichiers objets. L'édition de liens produit alors un fichier dit exécutable.

Par convention, les différents types de fichiers utilisés lors de la compilation sont distingués par leur suffixe. Les fichiers source sont suffixés par **.c**, les fichiers prétraités par le préprocesseur par **.i**, les fichiers assembleur par **.s**, et les fichiers objet par **.o**. Les fichiers objets correspondant aux bibliothèques pré-compilées ont pour suffixe « **.a** ». Enfin, sous

Unix/Linux, les exécutables produits n'ont pas de suffixe (« **.com** » ou « **.exe** » sous Windows).

Remarque importante (sous réserve de se faire taper sur les doigts par les puristes) : afin de ne pas réinventer la roue à chaque projet, certaines fonctions et routines sont compilées en fichiers objets (extension « **.o** »), puis classées dans des fichiers d'archives (extension « **.a** ») appelés « bibliothèques » en français (*libraries* en anglais). Il est interdit de traduire le mot anglais « library » en « librairie », mais bien en « bibliothèque ».

A noter qu'il existe une version moderne des bibliothèques : classiquement, lors de l'édition de liens, toutes les fonctions et routines utilisées par un programme étaient ajoutées à l'intérieur même de du fichier exécutable. Ainsi, si 10 programmes qui utilisent la même bibliothèque étaient exécutés en parallèle, chaque programme chargé en mémoire contient le même bout de code (ce qui prend inutilement de la place). Aujourd'hui, les fonctions qui sont mutualisées sont rangées dans des « bibliothèque dynamiques partagées ». On parlera de « shared library » sous Unix/Linux (suffixe « **.so** ») ou de « *dynamically linked library* » (suffixe « **.dll** ») sous Windows.

Classiquement, le compilateur C sous UNIX s'appelle **cc**. Il existe des alternatives, comme le compilateur du projet GNU : gcc. Ce compilateur peut-être obtenu gratuitement avec sa documentation et ses sources. Par défaut, gcc active toutes les étapes de la compilation. On le lance par la commande

```
gcc [options] fichier.c [-llibrairies]
```

Par défaut, le fichier exécutable s'appelle **a.out**. Le nom de l'exécutable peut être modifié à l'aide de l'option `-o`.

De façon classique (sans l'utilisation de bibliothèque dynamique partagée), les éventuelles bibliothèques sont déclarées par l'option `-lbibliothèque`. Dans ce cas, le système recherche le fichier **bibliothèque.a** dans le répertoire contenant les bibliothèques pré-compilées (généralement `/usr/lib/` sous Unix/Linux). Par exemple, pour lier le programme avec la librairie mathématique, on spécifie `-lm`. Le fichier objet correspondant est `libm.a`. Lorsque les librairies pré-compilées ne se trouvent pas dans le répertoire usuel, on spécifie leur chemin d'accès par l'option `-L`.

Les options les plus importantes du compilateur gcc sont les suivantes :

- `-c` : supprime l'édition de liens ; produit un fichier objet ;
- `-E` : n'active que le préprocesseur (le résultat est envoyé sur la sortie standard) ;
- `-g` : produit des informations symboliques nécessaires au débogueur ;
- `-Inom-de-répertoire` : spécifie le répertoire dans lequel doivent être recherchés les fichiers en-têtes à inclure (en plus du répertoire courant) ;
- -Lnom-de-répertoire : spécifie le répertoire dans lequel doivent être recherchées les librairies précompilées (en plus du répertoire usuel) ;
- `-o nom-de-fichier` : spécifie le nom du fichier produit. Par défaut, le fichier exécutable fichier produit s'appelle « **a.out** ».
- `-O, -O1, -O2, -O3` : options d'optimisations. Sans ces options, le but du compilateur est de minimiser le coût de la compilation. En rajoutant l'une de ces options, le compilateur tente de réduire la taille du code exécutable et le temps d'exécution. Les options correspondent à différents niveaux d'optimisation : `-O1` (similaire à `-O`) correspond à une faible optimisation, `-O3` à l'optimisation maximale ;
- `-S` : n'active que le préprocesseur et le compilateur ; produit un fichier assembleur ;
- `-v` : imprime la liste des commandes exécutées par les différentes étapes de la compilation ;

- `-W` : imprime des messages d'avertissement (warning) supplémentaires ;
- `-Wall` : imprime tous les messages d'avertissement.

4.3 Les composants élémentaires du C

Un programme en langage C est constitué des six groupes de composants élémentaires suivants :

- les identificateurs,
- les mots-clefs,
- les constantes,
- les chaînes de caractères,
- les opérateurs,
- les signes de ponctuation.

On peut ajouter à ces six groupes les commentaires, qui sont enlevés par le préprocesseur.

4.3.1 Les identificateurs

Le rôle d'un identificateur est de donner un nom à une entité du programme. Plus précisément, un identificateur peut désigner :

- un nom de variable ou de fonction,
- un type défini par `typedef`, `struct`, `union` ou `enum`,
- une étiquette (ou label).

Un identificateur est une suite de caractères parmi :

- les lettres (minuscules ou majuscules, mais non accentuées),
- les chiffres (sauf pour le premier caractère),
- le « blanc souligné » ou « *underscore* » : « _ ».

! Le langage C est « *case sensitive* » (tient compte des majuscules/minuscules).

Exemples : `var1`, `type_2` ou `_deb` sont des identificateurs valides.

Remarque : Il est déconseillé d'utiliser le *underscore* « _ » comme premier caractère d'un identificateur. En effet, par convention, il est souvent employé pour définir les variables globales de l'environnement C (prédéfinies par le compilateur).

Si la norme n'impose rien, pour éviter des surprises avec certains compilateurs, il est conseillé d'éviter de créer des identificateurs de plus de 31 caractères.

4.3.2 Les mots-clefs

Un certain nombre de mots, appelés mots-clefs, sont réservés pour le langage lui-même et ne peuvent pas être utilisés comme identificateurs. L'ANSI C compte 32 mots clefs :

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

que l'on peut ranger en catégories :

- les spécificateurs de stockage : `auto`, `register`, `static`, `extern`, `typedef` ;
- les spécificateurs de type : `char`, `double`, `enum`, `float`, `int`, `long`, `short`, `signed`, `struct`, `union`, `unsigned`, `void` ;
- les qualificateurs de type : `const`, `volatile` ;
- les instructions de contrôle : `break`, `case`, `continue`, `default`, `do`, `else`, `for`, `goto`, `if`, `switch`, `while` ;
- divers : `return`, `sizeof`.

4.3.3 Les commentaires

Un commentaire débute par `/*` et se termine par `*/`. Par exemple :

```
/*  Ceci est un commentaire */
```

On ne peut pas imbriquer des commentaires. Les compilateurs acceptent de plus en plus souvent les commentaires à la façon C++ : le commentaire commence par un double « / », et se continue jusqu'à la fin de la ligne. Exemple :

```
Ceci n'est pas un commentaire ; // Ceci est un commentaire
```

4.4 Structure d'un programme C

Une ***expression*** est une suite de composants élémentaires syntaxiquement correcte, par exemple :

```
x = y + 4
```

ou bien

```
(i >= 0) && (i < 10) || (p[i] != 0)
```

Une ***instruction*** est une expression suivie d'un point-virgule. Le point-virgule est donc le « séparateur d'instruction » ; il signifie en quelque sorte « évaluer cette expression ».

Plusieurs instructions peuvent être rassemblées par des accolades { et } pour former une instruction composée ou bloc qui est syntaxiquement équivalent à une instruction (le « { » signifie « début de bloc », ou « *begin* » dans d'autres langages comme le Pascal), et « } » signifie « fin de bloc » - le « *end* » - dans certains langages). Par exemple :

```
if (x != 0)
{
      z = y / x;
      t = y % x;
}
```

Une instruction composée d'un spécificateur de type et d'un identificateur ou d'une liste d'identificateurs séparés par une virgule est une ***déclaration***. Par exemple :

```
int a;
int b = 1, c;
double x = 2.38e4;
```

La déclaration d'un tableau se fait avec les accolades. Exemple :

```
int b[10] ;
/* déclare un tableau b de 10 entiers. Le premier élément du
tableau sera b[0], et le 10ième et dernier sera b[9] */
```

Remarque : en C, nous ne pouvons définir que des tableaux de taille connue lors de la compilation. Par exemple, le code suivant est interdit :

```
int  a=f() ; /* la variable a est initialisée avec le
résultat de l'appel à la fonction f() */
int  b[a] ; /* à la compilation, la valeur de « a » est
inconnue ; cette ligne engendrera une erreur de la part du
compilateur */
/* Par contre, la ligne suivante est correcte : */
int  c[4*4+1] ;
```

Important : en C, toute variable doit faire l'objet d'une déclaration avant d'être utilisée.

Typiquement, un programme C se présente de la façon suivante :

```
[directives au préprocesseur]
[déclarations de variables externes]
[fonctions secondaires]
main()
{
déclarations de variables internes
instructions
}
```

Les fonctions secondaires peuvent être placées indifféremment avant ou après la fonction principale `main()`. Si elle n'est pas définie, il est fortement conseillé qu'une fonction soit déclarée au moment où elle est utilisée. Une fonction secondaire peut s'écrire de la manière suivante :

```
type nom_de_ma_fonction (arguments)
{
     déclarations de variables internes
     instructions
}
```

Ainsi, le code suivant est correct :

```
void i()
{
     printf("bonjour.\n") ;
}

main ()
{
     i() ; /* ceci est un appel à la fonction i() */
}
```

Par contre, le code suivant n'est pas propre :

```
main ()
{
     i() ; /* ceci n'est pas propre : on appelle la fonction
i(), alors qu'à ce stade, on ne sait pas quels sont les
```

```
arguments acceptés, ni le type retourné. Ces informations ne
seront connues qu'après... */
}
void i()
{
     printf("bonjour.\n") ;
}
```

Pour faire propre, il faut déclarer la fonction i() avant qu'elle soit utilisée. Exemple :

```
/* déclaration de la fonction i() : */
void i(); /* ceci indique au compilateur les arguments et le
type retourné par la fonction i(), sans préciser son contenu
*/
main ()
{
     i() ; /* ici, i() est connue */
}
void i()
{
     printf("bonjour.\n") ;
}
```

La déclaration des arguments d'une fonction obéit à une syntaxe voisine de celle des déclarations de variables : on met en argument de la fonction une suite d'expressions type objet séparées par des virgules. Par exemple :

```
int somme(int a, int b)
{
     int resultat;
     resultat = a * b;
     return(resultat);
}
```

Remarque : la déclaration ci-dessus est moderne. Historiquement, Richie et Thompson utilisaient une autre syntaxe, toujours reconnue par les compilateurs, mais moins propre :

```
int somme(a, b)
int a;
int b;
{
     int resultat;
     resultat = a * b;
     return(resultat);
}
```

4.5 Les types

4.5.1 Les types prédéfinis

Le C est un langage « typé » : toute variable, constante ou fonction est d'un type précis. Le type d'un objet définit la façon dont il est représenté en mémoire.

Les types de base en C concernent les caractères, les entiers et les flottants (nombres réels). Ils sont désignés par les mots-clefs suivants :

- `char` (un octet : représente un caractère suivant son code ASCII),
- `int` (un entier ; sa taille dépend de la machine sur laquelle le code est compilé : 16 bits, 32 bits, 64 bits…),
- `float` (nombre en virgule flottante représenté sur 32 bits suivant le codage de l'IEEE),

- `double` (nombre en virgule flottante représenté sur 64 bits suivant le codage de l'IEEE),
- `short` (demi entier ; « `short` » employé seul est équivalent à « `short int` »),
- `long` (double le nombre de bit du type `int` ou `double` qui suit ; employé seul, il signifie « `long int` »),
- `unsigned` (par défaut, un type entier ou flottant est signé, sauf s'il est précédé du mot clé `unsigned`),
- `signed` (précise - bien que ce soit le cas par défaut – que le type qui suit est signé),
- `void` (type vide) : il s'agit d'un type très particulier, qui signifie « pas de type ». C'est principalement utilisé pour indiquer qu'une fonction ne retourne aucune valeur (c'est ainsi qu'en C, nous définissons une procédure : la notion de procédure n'existant pas, nous définissons une fonction qui retourne du vide).

Exemple de procédure en C :

```
void f(int a, int b)
{
     int i = a + b ;
     g(i);
     /* ici, il n'y a aucune instruction return(valeur) */
}
```

Pour savoir le nombre d'octet utilisé en mémoire pour stocker un type, il faut utiliser le mot clé prédéfini « `sizeof()` ». Exemple : `sizeof(int)`.

Remarque : en C, le type booléen n'existe pas. On utilise intrinsèquement le type `int`.

4.5.2 Les types composés

Voici les différents types composés en C :

- Nous avons précédemment vu les **tableaux**. Exemple : `float b[5] ;`
  Nous pouvons composer les symboles `[]` afin de définir des tableaux de tableau, des tableaux de tableaux de tableau, etc. Par exemple, pour définir une matrice n de 10x10 nombres à virgule flottante, nous pouvons utiliser :
  ```
  double    n[10][10] ;
  ```
  Dans cet exemple, `n[4]` est un tableau de 10 `double`.
- Une **structure** est une suite finie d'objets de types différents. Contrairement aux tableaux, les différents éléments d'une structure n'occupent pas nécessairement des zones contiguës en mémoire. Chaque élément de la structure, appelé *membre* ou *champ*, est désigné par un identificateur.
  On distingue la déclaration d'un modèle de structure de celle d'un objet de type structure correspondant à un modèle donné. La déclaration d'un modèle de structure dont l'identificateur est `modele_de_struct` suit la syntaxe suivante :

  ```
  struct modele_de_struct
  {
    type1 membre1;
    type2 membre2;
      ...
    typeN membreN;
  };

  /* une variable de type modele_de_struct se déclare
  ainsi : */
  ```

```
struct modele_de_struct var1 ;
/* on accède aux champs de var1 ainsi : */
var1.membre1=blablabla ;
var1.membre2=blablabla ;
```

- Une **union** désigne un ensemble de variables de types différents susceptibles d'occuper alternativement une même zone mémoire. Une union permet donc de définir un objet comme pouvant être d'un type au choix parmi un ensemble fini de types. Si les membres d'une union sont de longueurs différentes, la place réservée en mémoire pour la représenter correspond à la taille du membre le plus grand.

  Les déclarations et les opérations sur les objets de type union sont les mêmes que celles sur les objets de type struct. Dans l'exemple suivant, la variable `hier` de type `union jour` peut être soit un entier, soit un caractère :

```
union jour
{
  char lettre;
  int numero;
};
union jour hier;
...
jour.numero=4 ;
/* à ce stade, il serait incohérent (sauf cas
particulier) de vouloir utiliser jour.lettre */
```

- Les **champs de bits** (bitfields) permettent, en C, de spécifier la longueur des champs d'une structure au bit près si ce champ est de type entier (int ou unsigned int). Cela se fait en précisant le nombre de bits du champ avant le `;` qui suit sa déclaration. Par exemple, la structure suivante

```
struct registre
{
      unsigned int actif : 1;
      unsigned int valeur : 31;
};
```

  possède deux membres : `actif` qui est codé sur un seul bit, et `valeur` qui est codé sur 31 bits. Tout objet de type `struct registre` est donc codé sur 32 bits. Toutefois, l'ordre dans lequel les champs sont placés à l'intérieur de ce mot de 32 bits dépend de l'implémentation. Le champ `actif` de la structure ne peut prendre que les valeurs `0` et `1`. Aussi, si `r` est un objet de type `struct registre`, l'opération « `r.actif += 2;` » ne modifie pas la valeur du champ. Remarques : la taille totale d'un champ de bits doit être inférieure au nombre de bits d'un entier. De plus, un champ de bits n'a pas d'adresse ; on ne peut donc pas lui appliquer l'opérateur « & » ;

- Les **énumérations** permettent de définir un type par la liste des valeurs qu'il peut prendre. Un objet de type énumération est défini par le mot-clef `enum` et un identificateur de modèle, suivis de la liste des valeurs que peut prendre cet objet. Exemple :

```
enum modele { constante1, constante2, ...,constanteN };
```

  En réalité, les objets de type `enum` sont représentés comme des `int`. Les valeurs possibles constante1, constante2,... ,constanteN sont codées par des entiers de 0 à N-1. De plus, on peut modifier le codage par défaut des valeurs de la liste lors de la déclaration du type énuméré, par exemple : `enum booleen { faux = 0, vrai = 23 };`

- Enfin, pour alléger l'écriture des programmes, il est possible de définir des types composés avec `typedef`. La syntaxe est : `typedef type synonyme ;`

  Par exemple :

```
struct complexe_struct
{
      double reelle;
      double imaginaire;
};
typedef struct complexe_struct complexe;

main()
{
      complexe z;
      z.reelle=z.imaginaire=0 ;
      ...
}
```

4.6 Les constantes

Une constante est une valeur qui apparaît littéralement dans le code source d'un programme. Le type de la constante étant déterminé par la façon dont la constante est écrite. Les constantes peuvent être de 4 types : entier, flottant (nombre réel), caractère, ou énumération. Ces constantes vont être utilisées, par exemple, pour initialiser une variable.

4.6.1 Les constantes entières

Une constante entière peut être représentée de 3 manières différentes suivant la base dans laquelle elle est écrite :

- **décimale :** par exemple, `-10`, `0` et `2437348` sont des constantes entières décimales ;
- **octale :** la représentation octale d'un entier correspond à sa décomposition en base 8. Les constantes octales doivent commencer par un zéro. Par exemple, les représentations octales des entiers `0` et `255` sont respectivement `00` et `0377` ;
- **hexadécimale :** la représentation hexadécimale d'un entier correspond à sa décomposition en base 16. Les lettres de a à f sont utilisées pour représenter les nombres de 10 à 15. Les constantes hexadécimales doivent commencer par `0x` ou `0X`. Par exemple, les représentations hexadécimales de `14` et `255` sont respectivement `0xe` et `0xff`.

Par défaut, une constante décimale est représentée avec le format interne le plus court permettant de la représenter parmi les formats des types `int`, `long int` et `unsigned long int` tandis qu'une constante octale ou hexadécimale est représentée avec le format interne le plus court permettant encore de la représenter parmi les formats des types `int`, `unsigned int`, `long int` et `unsigned long int`.

On peut cependant spécifier explicitement le format d'une constante entière en la suffixant par **`u`** ou **`U`** pour indiquer qu'elle est non signée, ou en la suffixant par **`l`** ou **`L`** pour indiquer qu'elle est de type long. Par exemple :

Constante	Type
`123456`	`int`
`02311`	`int /* octal */`
`0x04d2`	`int /* hexadécimal */`

`123456789L`	`long`
`1234U`	`unsigned int`
`123456789UL`	`unsigned long int`

4.6.2 Les constantes réelles

Les constantes réelles sont représentées par la notation classique par mantisse et exposant. L'exposant est introduit par la lettre **e** ou **E** ; il s'agit d'un nombre décimal éventuellement signé.

Par défaut, une constante réelle est représentée avec le format du type double. On peut cependant influer sur la représentation interne de la constante en lui ajoutant un des suffixes **`f`** (indifféremment **F**) pour forcer le type `float`, ou **`l`** (indifféremment **L**) pour forcer le type **`long double`**. Par exemple :

Constante	Type
`12.34`	`double`
`12.3e-4`	`double`
`12.34F`	`float`
`12.34L`	`long double`

4.6.3 Les constantes caractères

Pour désigner un caractère imprimable, il suffit de le mettre entre apostrophes (par ex. `'A'` ou `'$'`).

Les seuls caractères imprimables qu'on ne peut pas représenter de cette façon sont l'antislash (« \ ») et l'apostrophe (« ' »), qui sont respectivement désignés par « `\\` » et « `\'` ». Le point d'interrogation et les guillemets peuvent aussi être désignés par les notations « `\?` » et « `\"` ». Les caractères non imprimables peuvent être désignés par `'\code-octal'` où code-octal est le code en octal du caractère. On peut aussi écrire `'\xcode-hexa'` où code-hexa est le code en hexadécimal du caractère. Par exemple, `'\33'` et `'\x1b'` désignent le caractère escape. Toutefois, les caractères non-imprimables les plus fréquents disposent aussi d'une notation plus simple :

Caractère	Valeur
`'\n'`	nouvelle ligne
`'\r'`	retour chariot
`'\t'`	tabulation horizontale
`'\f'`	saut de page
`'\v'`	tabulation verticale
`'\a'`	signal d'alerte
`'\b'`	retour arrière

4.7 Les opérateurs

4.7.1 L'affectation

En C, l'affectation est un opérateur à part entière. Elle est symbolisée par le signe « = ». Sa syntaxe est la suivante : `variable = expression`

Le terme de gauche de l'affectation peut être une variable simple, un élément de tableau mais pas une constante. Cette expression a pour effet d'évaluer *`expression`* et d'affecter la valeur obtenue à variable. Remarque importante : cette expression « `variable = expression` » possède elle même une valeur, qui est celle de `expression`. Ainsi, l'expression « `i = 5` » vaut `5`. Cette subtilité permet des écritures du style :

```
i = j = 20 + 4 ;
```

Dans ce cas, l'expression de droite est évaluée (`20 + 4` vaut `24`), et le résultat est affecté à la variable `j`. Puis, le résultat de « `j = 20 + 4` », c'est à dire `24`, est affecté à la variable `i`.

L'affectation effectue une conversion de type implicite : la valeur de l'expression (terme de droite) est convertie dans le type du terme de gauche. Par exemple, le programme suivant :

```
main()
{
      int i;
      int j = 2;
      float x = 2.5;
      i = j + x; /* i vaut 2 + 2 = 4 */
      x = x + i; /* x vaut 2.5 + 4 */
      /* à ce stade, x vaut 6.5 */
}
```

4.7.2 Les opérateurs arithmétiques

Les opérateurs arithmétiques classiques sont l'opérateur unaire moins « - » (changement de signe) ainsi que les opérateurs binaires :

+	addition
-	soustraction
*	multiplication
/	division
%	reste de la division (modulo)

Ces opérateurs agissent de la façon attendue sur les entiers comme sur les flottants. Leurs seules spécificités sont les suivantes :

- Contrairement à d'autres langages, le C ne dispose que de la notation `/` pour désigner à la fois la division entière et la division entre flottants. Si les deux opérandes sont de type entier, l'opérateur `/` produira une division entière (quotient de la division). Par contre, il délivrera une valeur flottante dès que l'un des opérandes est un flottant. Par exemple :

```
float x;
x = 3 / 2; /* ici, x vaut 1 */
x = 3 / 2.0; /* ici, x vaut 1.5 */
```

- L'opérateur `%` ne s'applique qu'à des opérandes de type entier. Si l'un des deux opérandes est négatif, le signe du reste dépend de l'implémentation, mais il est en général le même que celui du dividende.

- Notons enfin qu'il n'y a pas en C d'opérateur effectuant l'élévation à la puissance. Il faut utiliser des fonctions prédéfinies dans certaines bibliothèques pour réaliser cette opération.

4.7.3 Les opérateurs relationnels

Leur syntaxe est `expression1` **op** `expression2` avec **op** parmi :

>	strictement supérieur
>=	supérieur ou égal
<	strictement inférieur
<=	inférieur ou égal
==	égal
!=	différent

Les deux expressions sont évaluées puis comparées. Comme il n'existe pas de booléen, la valeur rendue est de type `int`, et elle vaut `1` si la condition est vraie, et `0` sinon.

Remarque importante, la faute du débutant : ne pas confondre l'opérateur de test d'égalité « == » avec l'opérateur d'affection « = ». Ainsi, le programme :

```
main()
{
      int a = 0;
      int b = 1;
      if (a = b)
            printf("\n a et b sont égaux\n");
      else
            printf("\n a et b sont différents\n");
}
```

affiche que a et b sont égaux !

4.7.4 Les opérateurs logiques booléens

Les opérateurs logiques sont :

&&	et logique
||	ou logique
!	négation logique

Comme pour les opérateurs de comparaison, la valeur retournée par ces opérateurs est de type `int` qui vaut `1` si la condition est vraie et `0` sinon.

Dans une expression de type

```
expression1 op1 expression2 op2 ...expressionN
```

l'évaluation se fait de gauche à droite et s'arrête dès que le résultat final est déterminé. Par exemple dans :

```
int i;
if ((i >= 0) && (i <= 9) && !(f(i) == 0))
...
```

la dernière clause « `!f(i) == 0` » ne sera évaluée (et par conséquent, la fonction `f()` ne sera appelée) que si `i` a une valeur entre `0` et `9`.

4.7.5 Les opérateurs logiques bit à bit

Les six opérateurs suivants permettent de manipuler des entiers au niveau du bit. Ils s'appliquent aux entiers de toute longueur (`short`, `int` ou `long`), signés ou non :

`&`	et
`|`	ou inclusif
`^`	ou exclusif
`~`	complément à 1
`<<`	décalage à gauche
`>>`	décalage à droite

Le décalage à droite et à gauche effectuent respectivement une multiplication et une division par une puissance de 2. Notons que ces décalages ne sont pas des décalages circulaires (le bit qui sort disparaît).

4.7.6 Les opérateurs d'affectation composée

Les opérateurs d'affectation composée sont :

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`
- `&=`
- `^=`
- `|=`
- `<<=`
- `>>=`

Pour tout opérateur op, l'expression `expression1 op= expression2` est équivalente à : `expression1 = expression1 op expression2`

Exemple : `i += j * 4` revient à `i = i + j * 4`.

4.7.7 Les opérateurs d'incrémentation et de décrémentation

Il s'agit des opérateurs :

- `++` : incrémentation,
- `--` : décrémentation.

Les opérateurs d'incrémentation ++ et de décrémentation `--` s'utilisent aussi bien en suffixe (`i++`) qu'en préfixe (`++i`). Dans les deux cas la variable `i` sera incrémentée.

Attention : dans la notation suffixe la valeur retournée sera l'ancienne valeur de i alors que dans la notation préfixe se sera la nouvelle. Par exemple :

```
int a = 3 ;
int b ;
int c ;
```

```
b = ++a;      /* a vaut 4 puis b prend la valeur de a, ie 4 */
c = b++;      /* c vaut la valeur de b (4), puis b est
incrémenté, sa valeur passe à 5 */
```

4.7.8 L'opérateur virgule

Une expression peut être constituée d'une suite d'expressions séparées par des virgules :

```
expression1, expression2, ..., expressionN
```

Cette expression est alors évaluée de gauche à droite. Sa valeur sera la valeur de l'expression de droite. Exemple :

```
main()
{
  int a, b;
  b = ((a = 3), (a + 2));
  /* a prend la valeur 3, puis on évalue a+2, le résultat est
mis dans b */
}
```

La virgule séparant les arguments d'une fonction ou les déclarations de variables n'est pas l'opérateur virgule. En particulier l'évaluation de gauche à droite n'est pas garantie. Par exemple, dans « `f(++a, ++b)` », nous ne savons pas si l'incrémentation de a se fera avant ou après celle de b. Pire encore, nous ne savons même pas si ces incrémentations se feront avant ou après l'appel de la fonction `f()`. Un conseil : ne pas utiliser ces notations. Utiliser plutôt : « `++a ; ++b ; f(a, b) ;` ».

4.7.9 L'opérateur conditionnel ternaire

L'opérateur conditionnel `?` est un opérateur ternaire. Sa syntaxe est la suivante :

```
condition ? expression1 : expression2
```

Cette expression est égale à `expression1` si `condition` est satisfaite, et à `expression2` sinon. Par exemple, l'expression :

```
x >= 0 ? x : -x
```

correspond à la valeur absolue d'un nombre `x`. De même l'instruction :

```
m = ((a > b) ? a : b);
```

affecte à `m` le maximum de a et de b.

4.7.10 L'opérateur de conversion de type

L'opérateur de conversion de type, appelé « cast », permet de modifier explicitement le type d'un objet. On écrit : `(type)objet`

```
float x = 2.1 ;
float y ;
y = (int)x * 2 ; /* y prend comme valeur 4 */
```

4.7.11 L'opérateur adresse / pointeur / indirection

Attention !!! C'est à partir de là que le C peut devenir obscur... La compréhension de ce paragraphe est capitale pour comprendre certains codes décortiqués dans ce cour.

L'opérateur d'adresse « `&` » appliqué à une variable retourne l'adresse mémoire où est stockée la valeur de cette variable. La syntaxe est « `&variable` ». Exemple :

```
int i ;
```

```
/* à ce stade, &i vaut l'emplacement dans la mémoire où est
stocké la valeur de la variable i */
```

L'opérateur « & » ne peut s'appliquer qu'à une variable (on parle de « *lvalue* »). En effet, « &(y+3) » n'a pas de sens : il signifierait « à quel endroit de la mémoire est stocké le résultat de la somme de y et de 3 ».

On dit que &y est le **pointeur** de y. Par exemple, si on écrit :

```
int i ;
a = &i ;
```

a sera un pointeur qui pointe sur l'emplacement où est stocké le contenu de i. Dans ce cas, a doit être de type « *pointeur vers un entier* ». Un pointeur se déclare ainsi :

```
type *nom_du_pointeur ;
```

Dans notre exemple ci-dessus, il faut déclarer a ainsi :

```
int  *a ; /* a est un pointeur vers un entier */
```

L'opérateur d'**indirection** « * », appliqué à un pointeur vers une variable v, correspond à la variable v. Exemple :

```
int  i ;
int  *a ;
a = &i; /* à ce stade, a va pointer sur la variable i */
*a = 3 ; /* cette instruction est strictement équivalente à
« i = 3 » */
```

Utilisation des pointeurs comme argument de fonction : en C, le passage d'argument se fait uniquement par valeur. Exemple :

```
int somme(int a, int b) ;

main()

{

     int x=4,y=2 ;

     somme(x,y) ;

     ...

}

int somme(int a, int b)

{

     a  +=  b ;  /*  c'est  la  variable  locale  a  qui  est
modifiée ; mais la valeur de la variable y de la fonction
main() n'est pas modifiée */

     return(a) ;

}
```

Les pointeurs sont donc utilisés pour effectuer un passage de paramètre par variable. Exemple :

```
void doubler(int *b)
{
     /* ici, b est un pointeur vers un entier */
     *b = *b * 2 ; /* on aurait pu aussi écrire *b *= 2 ; */
     /* dans la ligne ci-dessus, la variable entière pointée
par b est doublée */
}
main()
{
```

```
        int a=4 ;
        int *p ;
        p = &a ;
        /* à ce stade, la variable « a » vaut 4 */
        doubler( p ) ; /* on aurait pu écrire « doubler(&a) » */
        /* à ce stade, la variable « a » vaut 8 */
}
```

Attention !!! Ca n'est pas parce que nous définissons un pointeur vers un objet que pour autant, une place est réservée pour stocker un tel objet. Par exemple, le code suivant est incorrect :

```
float *p ;
*p = 4. ; /* faux : on ne sait pas sur quoi pointe p */
```

Remarque : Lorsque nous définissons une variable a comme étant un tableau de 10 éléments, a est le pointeur sur le premier élément. Exemple :

```
main()
{
     int a[10] ; /* définit un tableau de 10 entier. a[0] est
le premier élément de ce tableau, et a[9] est le 10ième et
dernier */
     *a = 4 ;
     /* La ligne ci-dessus est tout à fait acceptable en C,
bien que pas propre. Signifie « affecte 10 à la variable
pointée par a, qui est le premier élément du tableau. Pourrait
s'écrire : « a[0]=4 ; » */
}
```

L'opérateur « `->` » est une abréviation qui permet d'accéder aux champs d'une variable composée (i.e. de type `struct` ou `union`) à travers son pointeur. Exemple :

```
typedef struct { double im ; double re ; } complexe ;
complexe z ;
complexe *ptrz = &z ; /* définition de ptrz comme pointeur
vers un complexe, et on l'initialise en le faisant pointer
vers la variable z */
ptrz->im = ptrz->re = 1.0 ;
/* la ligne précédente équivaut à écrire : (*ptrz).im =
(*ptrz).re = 1.0 ; */
```

Attention !!! L'arithmétique des pointeurs est particulière. En effet, si `p` est un pointeur qui pointe vers une variable de type `t`, « `p++` » revient en réalité à ajouter à `p` le nombre d'octets pris en mémoire par un objet de type `t` (i.e. `sizeof(t)`). Exemple :

```
double t[2] ;
double *p = &t[0] ; /* p est défini comme pointeur sur un
double, et est initialisé pour pointer sur t */
*p=1.0 ; /* équivaut à t[0]=1.0 */
p++ ; /* p ne pointe pas vers l'octet suivant en mémoire, mais
vers l'objet de type double suivant */
p=4.0 ; /* équivaut à t[1]=4.0 */
```

4.7.12 Règles de priorité des opérateurs

Le tableau suivant classe les opérateurs par ordres de priorité décroissants. Les opérateurs placés sur une même ligne ont même priorité. Si dans une expression figurent plusieurs opérateurs de même priorité, l'ordre d'évaluation est définie par la flèche de la seconde colonne du tableau. On préférera toutefois mettre des parenthèses en cas de doute... :

() [] -> .	→
! ~ ++ -- -(*unaire*) (type) *(indirection) &(adresse) sizeof()	←
* / %	→
+ -(*binaire*)	→
<< >>	→
< <= > >=	→
== !=	→
&(*et bit-à-bit*)	→
^	→
|(*ou bit-à-bit*)	→
&&	→
||	→
? :	→
= += -= *= /= %= &= ^= |= <<= >>=	←
,	→

4.7.13 Les constantes chaînes de caractères

Une constante chaîne de caractères est une suite de caractères entourés par des guillemets « " ». Par exemple :

```
"Ceci est une chaîne de caractères"
```

Une chaîne de caractères peut contenir des caractères non imprimables, désignés par les représentations vues précédemment. Par exemple :

```
"ligne 1 \n ligne 2"
```

A l'intérieur d'une chaîne de caractères, le caractère « " » doit être désigné par `\"`. Enfin, le caractère `\` suivi d'un passage à la ligne est ignoré. Cela permet de faire tenir de longues chaînes de caractères sur plusieurs lignes. Par exemple :

```
"ceci est une longue longue  longue longue longue longue \
longue longue chaîne de caractères"
```

Attention !!! Encore un piège du C : en C, le type « chaîne de caractères » n'existe pas. Le C utilise des tableaux de caractères pour stocker une chaîne. En mémoire, le caractère nul « `\0` » permet de terminer une chaîne de caractères. Par exemple, la chaîne `"ABC"` prendra 4 octets en mémoire : le `65` (code ASCII du `'A'`), 66 (code ASCII du `'B'`), 67 (code ASCII du `'C'`), puis `'\0'`. Lorsque nous affectons une chaîne de caractères à une constante, il est inutile de compter combien il y a de caractères dedans :

```
char str[] = "ABC" ;
```

Avec ce que nous avons vu dans le chapitre sur les pointeurs, nous savons maintenant que nous pouvons écrire aussi :

```
char *str = "ABC" ;
```

4.8 Les instructions de branchement conditionnel

On appelle instruction de contrôle toute instruction qui permet de contrôler le fonctionnement d'un programme. Parmi les instructions de contrôle, on distingue les

instructions de branchement et les boucles. Les instructions de branchement permettent de déterminer quelles instructions seront exécutées et dans quel ordre.

4.8.1 Branchement conditionnel `if... else`

La forme la plus générale est celle-ci :

```
if (expression)
      instruction_executée_si_expression_est_vraie
else
      instruction_executée_si_expression_est_fausse
```

La clause « `else instruction_executée_si_expression_est_fausse` » est optionnelle.

Le code suivant :

```
if (expression1)
      instruction1
else if (expression2)
      instruction2
else
      instruction3
```

est équivalent à :

```
if (expression1)
      instruction1
else
{
      if (expression2)
            instruction2
      else
            instruction3
}
```

Autrement dit, instruction3 fait référence au dernier `else`, c'est à dire au deuxième `if`.

4.8.2 Branchement multiple `switch`

Sa forme la plus générale est celle-ci :

```
switch (expression)
{
      case constante1 :
            liste d'instructions 1
            break;
      case constante2 :
            liste d'instructions 2
            break;
      ...
      case constanteN :
            liste d'instructions N
            break;
      default:
            liste d'instructions si aucun cas rencontré
            break; /* ce dernier break est optionnel */
}
```

Si la valeur de `expression` est égale à l'une des constantes, la liste d'instructions correspondant est exécutée. Sinon la liste d'instructions « si aucun cas rencontré » correspondant à « `default` » est exécutée. L'instruction « `default` » est facultative.

Remarque importante : si on omet un « `break` », les instructions correspondant aux cas suivants seront aussi exécutées. Exemple :

```
switch (i)
{
      case 0 :
            liste d'instructions exécutées si i == 0 ;
            break;
      case 1 :
            liste d'instructions exécutées si i == 1 ;
      case 2 :
            liste d'instructions exécutées si i==1 ou i==2 ;
            break ;
      default:
            liste d'instructions si i différent de 0,1 et 2 ;
}
```

4.9 Les boucles

Les boucles permettent de répéter une série d'instructions tant qu'une certaine condition n'est pas vérifiée.

4.9.1 Boucle `while`

La syntaxe de `while` est la suivante :

```
while (expression)
      instruction
```

L'expression `expression` est tout d'abord évaluée une première fois. Si elle est vraie, `instruction` est effectuée. Puis, `expression` est de nouveau évaluée, etc. Exemple :

```
i = 1;
while (i < 10)
{
    g(i);
    i++;
}
```

4.9.2 Boucle `do... while`

Il peut arriver que l'on ne veuille effectuer le test de continuation qu'après avoir exécuté l'instruction. Dans ce cas, on utilise la boucle do... while. Sa syntaxe est :

```
do
      instruction
while (expression);
```

Ici, `instruction` sera toujours exécutée au moins une fois. La boucle continuera tant que `expression` est non nulle. Exemple :

```
int a=0;
do
{
      a=f(); /* sera toujours effectué */
} while (a > 0) && (a <= 10));
```

4.9.3 Boucle `for`

La syntaxe de `for` est :

```
for (expr1 ; expr2 ; expr3)
     instruction
```

Cette écriture est strictement équivalente à :

```
expr1 ;
while (expr2)
{
     instruction
     expr3
}
```

Exemple :

```
for (f = 1, i=1; i <= 10 ; ++i)
{
     f *= i ;
}
/* à ce stade, f vaut 10!=1*2*3*...*10 */
/* on pourrait même intégrer « f *= i »  dans la boucle.
   Aussi, le code ci-dessous est équivalent : */
for (f = 1, i=1; i <= 10 ; f *= i , ++i);
/* à éviter néanmoins, pour rendre le programme lisible */
```

4.10 Les instructions de branchement non conditionnel

4.10.1 Branchement non conditionnel `break`

Nous avons vu le rôle de l'instruction `break;` au sein d'une instruction de branchement multiple `switch`. L'instruction `break` peut, plus généralement, être employée à l'intérieur de n'importe quelle boucle. Elle permet d'interrompre le déroulement de la boucle, et passe à la première instruction qui suit la boucle. En cas de boucles imbriquées, `break` fait sortir de la boucle la plus interne. Exemple :

```
main()
{
  int i;
  for (i = 0; i < 5; i++)
  {
    g(i);
    if (i == 3)
      break;
  }
  /* à ce stade, i vaut 3 */
}
```

4.10.2 Branchement non conditionnel `continue`

L'instruction `continue` permet de passer directement au tour de boucle suivant, sans exécuter les autres instructions de la boucle. Exemple :

```
main()
{
  int i;
  for (i = 0; i <= 5; i++)
```

```
  {
    if (i == 3)
      continue;
    g(i);
  }
}
/* la fonction g(i) ne sera jamais appelée quand i vaut 3 */
```

4.10.3 Branchement non conditionnel `goto`

L'instruction `goto` permet d'effectuer un saut jusqu'à l'instruction `etiquette` correspondant. En général, elle est à proscrire de tout programme C digne de ce nom. Il existe quelques exceptions, pour écrire du code efficace (sortie de boucles imbriquées complexes en évitant quelques tests). Exemple :

```
int  i,j,k;
debut :
for (i=0 ;i<10 ;++i)
{
     do
     {
          k=g(i);
          while (f(k)!=0)
          {
               j=h(k);
               if (recommence(j))
                    goto debut;
               k++;
          }
     } while (k != 4);
}
```

4.11 Les fonctions

4.11.1 Définition d'une fonction

Nous avons déjà vu comment définir une fonction :

```
type nom-fonction (type1 arg1,... ,typeN argN)
{
  [déclarations de variables locales ]
  liste d'instructions
}
```

Et que la syntaxe suivante, bien que désuète et lourde, reste acceptable :

```
type nom-fonction (arg1, ..., argN)
type1 arg1 ;
... ;
typeN argN ;
{
  [déclarations de variables locales ]
  liste d'instructions
}
```

La première ligne de cette définition est *l'en-tête* de la fonction. Dans cet en-tête, `type` désigne le type retourné par la fonction. Contrairement à d'autres langages, il n'y a pas en C de notion de procédure ou de sous-programme. Une telle fonction sera définie en retournant un type `void`.

Les arguments de la fonction sont appelés paramètres formels, par opposition aux paramètres effectifs qui sont les paramètres avec lesquels la fonction est effectivement appelée. Les paramètres formels peuvent être de n'importe quel type. Leurs identificateurs n'ont d'importance qu'à l'intérieur de la fonction. Enfin, si la fonction ne possède pas de paramètres, on remplace la liste de paramètres formels par le mot-clef `void`.

Le corps de la fonction débute éventuellement par des déclarations de variables, qui sont locales à cette fonction. Il se termine par l'instruction de retour à la fonction appelante, return, dont la syntaxe est `return(expression);`

La valeur de `expression` est la valeur que retourne la fonction. Son type doit être le même que celui qui a été spécifié dans l'en-tête de la fonction. Si la fonction ne retourne pas de valeur (fonction de type `void`), sa définition s'achève par « `return;` » (qui est implicite).

Plusieurs instructions return peuvent apparaître dans une fonction. Le retour au programme appelant sera alors provoqué par le premier return rencontré lors de l'exécution.

4.11.2 Appel d'une fonction

L'appel d'une fonction se fait par l'expression `nom-fonction(param1, ...,paramN)`

En C, l'ordre et le type des paramètres effectifs de la fonction doivent concorder avec ceux donnés dans l'en-tête de la fonction. Les paramètres effectifs peuvent être des expressions. La virgule qui sépare deux paramètres effectifs est un simple signe de ponctuation ; il ne s'agit pas de l'opérateur virgule. Cela implique en particulier que l'ordre d'évaluation des paramètres effectifs n'est pas assuré et dépend du compilateur.

4.11.3 Déclaration d'une fonction

Le C n'autorise pas les fonctions imbriquées. La définition d'une fonction secondaire doit donc être placée soit avant, soit après la fonction principale main. Par exemple, le code suivant est interdit :

```
void une_fonction()
{
     int une_sous_fonction() /* la définition de cette sous-
fonction, ou fonction imbriquée, est interdite à l'intérieur
de une_fonction() ; notez que d'autres langages, comme le
Pascal, autorise cette forme de programmation */
     {
     }
}
```

Il est indispensable que le compilateur « *connaisse* » la fonction au moment où celle-ci est appelée. Si une fonction est définie après son premier appel, elle doit impérativement être déclarée au préalable. Une fonction secondaire est déclarée par son *prototype*, qui donne le type de la fonction et celui de ses paramètres, sous la forme :

```
type nom-fonction(type1,... ,typeN);
```

Notez que la déclaration ressemble à la définition : le code compris entre « `{...}` » est remplacé par un « `;` ». Remarque : si l'ordre et le type des paramètres doivent être respectés entre la déclaration et la définition, le nom des arguments peut être différent. Exemple :

```
int f(int a, float b) ;
...
int f(int i, float j)
{
     blablabla;
}
```

Il est même inutile de mettre un nom aux paramètres. La déclaration suivante est toute aussi bonne :

```
int f(int, float) ;
```

Même si la déclaration est parfois facultative (par exemple quand les fonctions sont définies avant la fonction main et dans le bon ordre), elle seule permet au compilateur de vérifier que le nombre et le type des paramètres utilisés dans la définition concordent bien avec le protype. De plus, la présence d'une déclaration permet au compilateur de mettre en place d'éventuelles conversions des paramètres effectifs, lorsque la fonction est appelée avec des paramètres dont les types ne correspondent pas aux types indiqués dans le prototype.

4.11.4 Cas particulier de la fonction `main()`

La fonction principale `main()` est une fonction comme les autres. Nous avons jusqu'à présent considéré qu'elle était de type `void`, ce qui est toléré par le compilateur (mais génère normalement un message d'alerte).

En fait, la fonction `main()` est de type `int`. Elle doit retourner un entier dont la valeur est transmise à l'environnement d'exécution. Cet entier indique si le programme s'est ou non déroulé sans erreur. La valeur de retour 0 correspond à une terminaison correcte, toute valeur de retour non nulle correspond à une terminaison sur une erreur. L'appel à « `return(statut);` » dans la fonction `main()`, où statut est un entier spécifiant le type de terminaison du programme, peut être remplacée par un appel à la fonction `exit()` (nous verrons plus tard comment faire appel à des fonctions prédéfinies ailleurs).

La fonction `main()` peut également posséder des paramètres formels. En effet, un programme C peut recevoir une liste d'arguments au lancement de son exécution. La ligne de commande qui sert à lancer le programme est, dans ce cas, composée du nom du fichier exécutable suivi par des paramètres. La fonction main reçoit tous ces éléments de la part de l'interpréteur de commandes. En fait, la fonction main possède deux paramètres formels, appelés par convention `argc` (*argument count*) et `argv` (*argument vector*). `argc` est une variable de type `int` dont la valeur est égale au nombre de mots composant la ligne de commande (y compris le nom de l'exécutable). Elle est donc égale au nombre de paramètres effectifs de l'éxécutable + 1. `argv` est un tableau de chaînes de caractères correspondant chacune à un mot de la ligne de commande. Le premier élément, `argv[0]`, contient donc le nom de la commande (du fichier exécutable), le second `argv[1]` contient le premier paramètre...

Le prototype valide de la fonction main est donc :

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

4.11.5 Pointeur sur une fonction

Il est parfois utile de passer une fonction comme paramètre d'une autre fonction. Cette procédure permet en particulier d'utiliser une même fonction pour différents usages. Pour cela, on utilise un mécanisme de pointeur. Un pointeur sur une fonction correspond à l'adresse du début du code de la fonction. Un pointeur sur une fonction ayant pour prototype « `type fonction(type1,... ,typeN);` » est de type :

```
type (*)(type1,... ,typeN);
```

Ainsi, une fonction `operateur_binaire` prenant pour paramètres deux entiers et une fonction de type `int`, qui prend elle-même deux entiers en paramètres, sera définie par :

```
int operateur_binaire(int, int, int (*)(int, int)) ;
```

et sera définie par :

```
int operateur_binaire(int a, int b, int (*f)(int, int))
{
     int i, j ;
```

```
        ...
        (*f)(i,j) ; /* appel de la fonction f passée en
    paramètre */
    }
```

Notons qu'on n'utilise pas la notation « &nom_fonction », mais simplement « nom_fonction » comme paramètre effectif de d'une fonction. Exemple :

```
int exemple(int, int);

int operateur_binaire(int, int, int(*)(int, int));

main()

{

        int i, j ;

        operateur_binaire(i,  j,  exemple) ;  /*  et  non  pas
    &exemple */

}
```

4.11.6 Fonctions avec un nombre variable de paramètres

Il est possible en C de définir des fonctions qui ont un nombre variable de paramètres. Une fonction possédant un nombre variable de paramètre doit posséder au moins un paramètre formel fixe. La notation « `...` » (obligatoirement à la fin de la liste des paramètres d'une fonction) spécifie que la fonction possède un nombre quelconque de paramètres (éventuellement de types différents) en plus des paramètres formels fixes. Ainsi, une fonction ayant pour prototype :

```
int f(int a, char c, ...);
```

prend comme paramètre un entier, un caractère, et un nombre quelconque d'autres paramètres.

Un appel à une fonction ayant un nombre variable de paramètres s'effectue comme un appel à n'importe quelle autre fonction.

Pour accéder à la liste des paramètres de l'appel, on utilise les macros définies dans le fichier en-tête « `stdarg.h` » de la librairie standard (nous reviendrons sur ce point ultérieurement).

Il faut tout d'abord déclarer dans le corps de la fonction une variable pointant sur la liste des paramètres de l'appel. Cette variable a pour type `va_list`. Par exemple :

```
va_list liste_parametres;
```

Cette variable est tout d'abord initialisée à l'aide de la macro `va_start`, dont la syntaxe est :

```
va_start(liste_parametres, dernier_parametre);
```

où `dernier_parametre` désigne l'identificateur du dernier paramètre formel fixe de la fonction. Après traitement des paramètres, on libère la liste à l'aide de la macro `va_end` :

```
va_end(liste_parametres);
```

On accède aux différents paramètres de liste par la macro `va_arg()` qui retourne le paramètre suivant de la liste:

```
va_arg(liste_parametres, type)
```

où `type` est le type supposé du paramètre auquel on accède.

Notons que l'utilisateur doit lui-même gérer le nombre de paramètres de la liste. Pour cela, on utilise généralement un paramètre formel qui correspond au nombre de paramètres de la liste, ou une valeur particulière qui indique la fin de la liste.

Cette méthode est utilisée dans le programme d'exemple suivant, où la fonction `add` effectue la somme de ses paramètres en nombre quelconque.

```
/* les 3 instruction #include ci-dessous seront expliquées
dans un chapitre ultérieur */
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>

int add(int, ...);
int add(int nb,...)
{
  int res = 0;
  int i;
  va_list liste_parametres;

  /* on suppose que le programme appelant envoie en premier
paramètre le nombre de variables à additionner ; le
compilateur n'a aucun moyen de vérifier si le programmeur
applique toujours bien cette convention : */
  va_start(liste_parametres, nb);
  for (i = 0; i < nb; i++)
    res += va_arg(liste_parametres, int);
  va_end(liste_parametres); /* on libère la place prise par la
liste en mémoire */
  return(res);
}

int main(void)
{
  int n1, n2 ;
  n1=add(4,10,2,8,5); /* n1 vaut 25 */
  n2=add(6,10,15,5,2,8,10); /* n2 vaut 50 */
  return(0);
}
```

4.12 Durée de vie des variables

Les variables manipulées dans un programme C ne sont pas toutes traitées de la même manière. En particulier, elles n'ont pas toutes la même durée de vie. On distingue deux catégories de variables :

- **Les variables permanentes (ou statiques) :** occupent un emplacement en mémoire qui reste le même durant toute l'exécution du programme. Cet emplacement est alloué une fois pour toutes lors de la compilation. La partie de la mémoire contenant les variables permanentes est appelée **segment de données** (nous y reviendrons fréquemment dans ce cour). Par défaut, les variables permanentes sont initialisées à zéro par le compilateur. Elles sont caractérisées par le mot-clef `static` ;
- **Les variables temporaires :** elles se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme. Elles ne sont pas initialisées par défaut. Leur emplacement en mémoire est libéré par exemple à la fin de l'exécution d'une fonction secondaire.

Par défaut, les variables temporaires sont situées dans la partie de la mémoire appelée **segment de pile** (même remarque : nous y reviendrons fréquemment par la suite). Dans ce cas, la variable est dite *automatique*. Le spécificateur de type correspondant, `auto`, est

rarement utilisé puisqu'il ne s'applique qu'aux variables temporaires qui sont automatiques par défaut.

Une variable temporaire peut également être placée dans un *registre* de la machine. Un registre est une zone mémoire particulière (généralement, gravé dans le microprocesseur) sur laquelle sont effectuées les opérations machine. Il est donc beaucoup plus rapide d'accéder à un registre qu'à toute autre partie de la mémoire. On peut demander au compilateur de ranger une variable très utilisée dans un registre, à l'aide de l'attribut de type `register`. Le nombre de registres étant limité, cette requête ne sera satisfaite que s'il reste des registres disponibles. Cette technique permettant d'accélérer les programmes a aujourd'hui perdu tout son intérêt. Grâce aux performances des optimiseurs de code intégrés au compilateur (cf. options -On de gcc), il est maintenant plus efficace de compiler un programme avec une option d'optimisation que de placer certaines variables dans des registres.

La durée de vie des variables est liée à leur portée, c'est-à-dire à la portion du programme dans laquelle elles sont définies.

4.12.1 Variables globales

On appelle variable globale une variable déclarée en dehors de toute fonction. Une variable globale est connue du compilateur dans toute la portion de code qui suit sa déclaration. Les variables globales sont systématiquement permanentes. Dans le programme suivant, n est une variable globale :

```
int n;
/* c'est implicite : à ce stade, n=0 */
void incremente_n();
void incremente_n ()
{
  n++;
  return; /* ce return n'est pas obligatoire */
}
```

4.12.2 Variables locales

On appelle variable locale une variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instructions) du programme. Par défaut, les variables locales sont temporaires. Quand une fonction est appelée, elle place ses variables locales dans la pile à l'exécution. A la sortie de la fonction, les variables locales sont dépilées et donc perdues.

Remarque : Les variables locales n'ont en particulier aucun lien avec des variables globales de même nom. Par exemple, le programme suivant

```
int n = 10;
void f();
void f()
{
  int n = 0;
  n++; /* ici, c'est le n local qui est modifié ; vu que nous
avons appelé une variable locale « n », il n'est plus possible
d'accéder à la variable global n initialisée à 10 au dessus */
}
```

Remarque : les variables locales à une fonction ont une durée de vie limitée à une seule exécution de cette fonction. Leurs valeurs ne sont pas conservées d'un appel au suivant.Il

est toutefois possible de créer une variable locale de classe statique en faisant précéder sa déclaration du mot-clef `static`. Exemple : `static type nom_de_variable;`

Une telle variable reste locale à la fonction dans laquelle elle est déclarée (impossible d'y accéder depuis une autre fonction), mais sa valeur est conservée d'un appel au suivant. Elle est également initialisée à zéro à la compilation. Exemple :

```
void f();
void f()
{
  static int n;
  n++;
  affiche_un_entier(n);
}

main()
{
  f();
  f();
}
```

Si la fonction « affiche_un_entier » est définie ailleurs et affiche le contenu de l'entier passé en paramètre, le programme afficherait :

```
1

2
```

4.12.3 Transmission des paramètres d'une fonction

Comme nous l'avons déjà vu, les paramètres d'une fonction sont traités de la même manière que les variables locales de classe automatique : lors de l'appel de la fonction, les paramètres effectifs sont copiés dans le segment de pile. La fonction travaille alors uniquement sur cette copie. Cette copie disparaît lors du retour au programme appelant. Cela implique en particulier que, si la fonction modifie la valeur d'un de ses paramètres, seule la copie sera modifiée ; la variable du programme appelant, elle, ne sera pas modifiée (passage des paramètres par valeurs).

Remarque : attention, un tableau est un pointeur. En passant un tableau en paramètre d'une fonction, le contenu du tableau sera modifiable.

4.12.4 Les qualificateurs de type const et volatile

Les qualificateurs de type `const` et `volatile` permettent de réduire les possibilités de modifier une variable :

- Une variable dont le type est qualifié par `const` ne peut pas être modifiée. Ce qualificateur est utilisé pour se protéger d'une erreur de programmation. On l'emploie principalement pour qualifier le type des paramètres d'une fonction afin d'éviter de les modifier involontairement ;
- Une variable dont le type est qualifié par `volatile` ne peut pas être impliquée dans les optimisations effectuées par le compilateur. On utilise ce qualificateur pour les variables susceptibles d'être modifiées par une action extérieure au programme.

Les qualificateurs de type se placent juste avant le type de la variable.

4.13 Les directives au préprocesseur

Comme nous l'avons déjà vu, le préprocesseur est un programme exécuté lors de la première phase de la compilation. Il effectue des modifications textuelles sur le fichier source à partir de directives.

A noter que le préprocesseur est totalement idiot : il ne connaît rien au C. Il ne fait qu'appliquer des directives comme « importer le contenu d'un autre fichier » (`#include`), remplacer partout un mot clé par un autre (`#define`), ou effectue des compilations conditionnelles (`#ifdef` ... `#if` ... `#else` ... `#endif`).

Remarque : en général, dans beaucoup de compilateurs, le # d'une directive doit toujours être le **premier** caractère d'une ligne. Par exemple, le code suivant pourra être refusé par certains compilateurs :

```
#if aaa
      #if bbb
            …
      #endif
#endif
```

Par contre, nous pourrons écrire :

```
#if aaa
#     if bbb
            …
#     endif
#endif
```

4.13.1 La directive `#include`

La directive `#include` permet d'incorporer dans le fichier source le texte figurant dans un autre fichier. Ce dernier peut être un fichier en-tête (qui introduit des prototypes de fonctions externes au programmes) ou n'importe quel autre fichier.

La directive `#include` possède deux syntaxes voisines mais au comportement différent :

- `#include <nom-de-fichier>` : recherche le fichier mentionné dans un ou plusieurs répertoires systèmes définis par l'implémentation (par exemple, `/usr/include/`) ;
- `#include "nom-de-fichier"` : recherche le fichier dans le répertoire courant (celui où se trouve le fichier source). On peut spécifier d'autres répertoires à l'aide de l'option « `-I` » du compilateur.

La première syntaxe est généralement utilisée pour inclure les fichiers en-tête des fonctions des bibliothèques livrées en standard avec le compilateur, tandis que la seconde est plutôt destinée aux fichiers créés par l'utilisateur.

4.13.2 La directive `#define`

La directive `#define` permet de définir :

- des constantes symboliques,
- des macros avec paramètres.

Exemple de définition de constantes symboliques : la directive « `#define nom reste-de-la-ligne` » demande au préprocesseur de substituer toute occurrence de `nom` par la chaîne de caractères `reste-de-la-ligne` dans la suite du fichier source. Son utilité principale est de donner un nom parlant à une constante, qui pourra être aisément modifiée. Par exemple :

```
#define TAILLE_ECRAN_X 1024
```

```
#define TAILLE_ECRAN_Y 768
#define NB_COULEUR 256
#define NB_PIXELS_ECRAN TAILLE_ECRAN_X * TAILLE_ECRAN_Y
```

Il n'y a toutefois aucune contrainte sur la chaîne de caractères reste-de-la-ligne (par exemple, il peut s'agir de mots clés du langage). Par exemple, on peut écrire (beurk) :

```
#define BEGIN {
#define END }
```

Attention au piège !!! Comme déjà indiqué, le remplacement d'une chaîne se fait bêtement, sans réfléchir aux conséquences. Exemple :

```
#define    a    1

#define    b    2

#define    somme_a_plus_b  a + b

...

int i = somme_a_plus_b ; /* i vaut 3 */

int j = somme_a_plus_b * 3 ; /* Par contre, j ne vaut pas 9 !
En effet, j vaut 1 + 2 * 3, et comme la multiplication est
prioritaire sur l'addition, on a : j = 7 */
```

N'hésitez pas à parenthèser pour éviter ces effets de bord :

```
#define    somme_a_plus_b  ( a + b )
```

Il est aussi possible d'utiliser la directive `#define` afin de définir des macros avec la syntaxe suivante :

```
#define nom(liste-de-paramètres) corps-de-la-macro
```

où `liste-de-paramètres` est une liste d'identificateurs séparés par des virgules. Par exemple, avec la directive :

```
#define MAX(a,b) (a > b ? a : b)
```

le processeur remplacera dans la suite du code toutes les occurrences du type « `MAX(x,y)` » où x et y sont des symboles quelconques par « `(x > y ? x : y)` ».

Attention, ici aussi, pour éviter les effets de bors, comme a et b peuvent être n'importe quelle expression complexe, il vaut mieux parenthéser. La bonne définition de la macro, qui évite tout effet de bord, est :

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

Une macro a donc une syntaxe similaire à celle d'une fonction, mais son emploi permet en général d'obtenir de meilleures performances en temps d'exécution (mais pas en place prise par le code en mémoire).

La distinction entre une définition de constante symbolique et celle d'une macro avec paramètres se fait sur le caractère qui suit immédiatement le nom de la macro : si ce caractère est une parenthèse ouvrante, c'est une macro avec paramètres, sinon c'est une constante symbolique. Il ne faut donc jamais mettre d'espace entre le nom de la macro et la parenthèse ouvrante. Ainsi, si l'on écrit par erreur :

```
#define CARRE (a) a * a
```

la chaîne de caractères « `CARRE(2)` » sera remplacée par

```
(a) a * a (2)
```

Ici aussi, pour éviter tout effet de bord avec la priorité des opérateurs, la bonne syntaxe serait :

```
#define CARRE(a) ( (a) * (a) )
```

En effet, en C, il n'est jamais gênant de surparenthèser. Le code suivant fonctionnera :

```
int i=((2)*(3)-(*(p))); /* notez que p est un pointeur */
```

4.13.3 La compilation conditionnelle

La compilation conditionnelle a pour but d'incorporer ou d'exclure des parties du code source dans le texte qui sera généré par le préprocesseur. Elle permet d'adapter le programme au matériel ou à l'environnement sur lequel il s'exécute, ou d'introduire dans le programme des instructions de débogage.

Les directives de compilation conditionnelle se répartissent en deux catégories, suivant le type de condition invoquée :

- la valeur d'une expression
- l'existence ou l'inexistence de symboles.

Exemples :

```
#if condition1
  partie-du-programme-1
#elif condition2
  partie-du-programme-2
     ...
#elif conditionN
  partie-du-programmeN
#else
  partie-du-programme-defaut
#endif
```

Le nombre de `#elif` est quelconque et le `#else` est facultatif. Chaque conditionX doit être une expression constante.

Une seule « `partie-du-programme` » sera compilée : celle qui correspond à la première `conditionX` non nulle, ou bien la `partie-du-programme-defaut` si toutes les conditions sont nulles.

Par exemple, on peut écrire :

```
#define PROCESSEUR ALPHA
...
#if PROCESSEUR == ALPHA
  taille_long = 64;
#elif PROCESSEUR == PC
  taille_long = 32;
#endif
```

L'autre directive de compilation conditionnelle est un test lié à l'existence d'un symbole. Sa syntaxe est :

```
#ifdef symbole
  partie-du-programme-1
#else
  partie-du-programme-2
#endif
```

Si `symbole` est défini au moment où l'on rencontre la directive `#ifdef`, alors `partie-duprogramme-1` sera compilée et `partie-du-programme-2` sera ignorée. Dans le cas contraire, c'est `partie-du-programme-2` qui sera compilée. La directive `#else` est évidemment facultative.

Da façon similaire, on peut tester la non-existence d'un symbole par : `#ifndef symbole`

Ce type de directive est utile pour rajouter des instructions destinées au débogage du programme :

```
#define DEBUG
  ...
#ifdef DEBUG
```

```
  code qui affiche le contenu de certaines variables, etc.
#endif /* DEBUG */
```

Il suffit alors de mettre en commentaire la directive « `#define DEBUG` » pour que les instructions liées au débogage ne soient pas compilées. Cette dernière directive peut être remplacée par l'option de compilation `-Dsymbole`, qui permet de définir un symbole lorsqu'on lance la compilation. Ainsi, « `gcc -DDEBUG fichier.c` » équivaut à un « `gcc fichier.c` » avec un fichier « `fichier.c` » dont la première ligne serait : « `#define DEBUG` ».

Remarque : ces directives conditionnelles sont très utilisées pour éviter d'importer un fichier plusieurs fois avec la directive « `#include` ». Exemple d'un fichier « test.h » :

```
#ifndef TEST_H
#define TEST_H
blablabla...
#endif
```

Ainsi, si dans le "*blablabla*" ci-dessus, il venait à y avoir un « `#include "fichier2.h"` », et que `fichier2.h` venait à faire un « `#include "test.h"` », le compilateur ne tournerait pas en boucle.

4.14 Les bibliothèques standards

Contrairement à d'autres langages, il n'existe pas en C de fonctions prédéfinies pour écrire à l'écran, ouvrir/lire/écrire/fermer un fichier, comparer deux chaînes de caractères, trier un tableau d'entier, etc.

Par contre, tout compilateur propose des « *bibliothèques de fonctions* », rangées dans des *archives* ou *bibliothèques dynamiques*. Or, si certaines de ces fonctions sont propres à chaque compilateur et/ou à chaque environnement, certaines fonctions sont obligatoires, et inscrites dans la norme ANSI, dans une bibliothèque qui est toujours utilisée lors de la phase d'édition de liens durant la compilation, sans que l'utilisateur n'ait besoin de la spécifier explicitement : la bibliothèque **`libc`**. D'autres fonctions sont disponibles si votre environnement répond à d'autres normes, comme par exemple la norme POSIX. Certaines fonctions de la norme POSIX sont déjà présentes dans la bibliothèque `libc`. D'autres ont besoin que l'utilisateur précise l'utilisation de bibliothèques externes particulières lors de la phase d'édition de lien, avec l'option « `-lnom_de_la_bibliothèque` » du compilateur.

Exemple : pour afficher une chaîne de caractères `"bonjour\n"` à l'écran, nous pouvons utiliser la fonction standard AINSI « `puts()` » existant dans la bibliothèque `libc`.

Or, le compilateur risque de nous afficher une alerte si nous écrivons le code suivant :

```
char s[]="ceci est un test\n" ;
main()
{
      put(s) ;
}
```

En effet, la fonction `puts()` est définie nulle part. Par contre, en lisant la documentation de cette fonction `puts()` (`man puts` sous Unix/Linux par exemple), nous apprenons que le prototype de la fonction `puts()` est définie dans le fichier `stdio.h`. Il suffit donc de commencer notre programme par : `#include <stdio.h>`

Ainsi, nous pourrons utiliser `puts()` (et beaucoup d'autre fonctions, constantes, macos) dont les déclarations se trouvent dans le fichier `/usr/include/stdio.h`.

Pour les fonctions qui ne font par partie de la bibliothèque standard `libc`, il faudra certainement fournir l'option « `-lnom_d'une_bibliothèque_externe` » au compilateur pour que ce dernier sache où aller chercher les fonctions dont le programme a besoin.

Exemple : il existe une fonction mathématique « `double pow(double x, double y);` » qui retourne le nombre flottant `x` à la puissance `y`. Le manuel de cette fonction nous apprend :

- que la définition de cette fonction se trouve dans le fichier `math.h`. Avant d'utiliser cette fonction dans notre programme, il faudra bien penser à faire un : `#include <math.h>`
- de plus, il faut penser à indiquer au compilateur d'aller chercher cette fonction dans la bibliothèque des fonctions mathématiques « m », en donnant l'option de compilation « `-lm` ». Exemple : `cc test.c –o test –lm`

Sans cette dernière option, nous aurions certainement une erreur comme celle-ci à la compilation :

```
In function `main':
: undefined reference to `pow'
collect2: ld returned 1 exit status
```

4.15 Les conventions d'écriture d'un programme C

Il existe très peu de contraintes dans l'écriture d'un programme C. Par exemple, en dehors d'une composante élémentaire, il est toujours possible d'ajouter des espaces, des tabulations, des retours à la ligne...

Toutefois ne prendre aucune précaution aboutirait à des programmes illisibles. Aussi existe-t-il un certain nombre de conventions.

- on n'écrit qu'une seule instruction par ligne : le point virgule d'une instruction ou d'une déclaration est toujours le dernier caractère de la ligne ;
- les instructions sont disposées de telle façon que la structure modulaire du programme soit mise en évidence. En particulier, une accolade ouvrante marquant le début d'un bloc doit être seule sur sa ligne ou placée à la fin d'une ligne. Une accolade fermante est toujours seule sur sa ligne ;
- on laisse un blanc :
  - entre les mots-clefs `if`, `while`, `do`, `switch` et la parenthèse ouvrante qui suit,
  - après une virgule,
  - de part et d'autre d'un opérateur binaire (pas toujours suivi) ;
- on ne met pas de blanc entre un opérateur unaire et son opérande, ni entre les deux caractères d'un opérateur d'affectation composée.
- les instructions doivent être indentées afin que toutes les instructions d'un même bloc soient alignées.

4.16 Le C++

4.16.1 Généralités

Le langage C++ a été conçu par *Bjarne Stroustrup*, afin d'ajouter (entre autre) au C la notion de « *programmation objet* ».

La Programmation Orientée Objet (POO), parfois appelée Programmation Par Objets (PPO). Cette technique permet d'introduire le concept d'« *objet* », qui consiste en un ensemble de données **et** de procédures qui agissent sur ces données.

Lorsque l'objet est parfaitement bien écrit, il introduit la notion fondamentale d'« *encapsulation des données* ». Ceci signifie qu'il n'est plus possible pour l'utilisateur de l'objet, d'accéder directement aux données : il doit passer par des « *méthodes* » spécifiques écrites par le concepteur de l'objet, et qui servent d'interface entre l'objet et ses utilisateurs. L'intérêt de cette technique est évident. L'utilisateur ne peut pas intervenir directement sur l'objet, ce qui diminue les risques d'erreur, ce dernier devenant une *boîte noire*.

Une autre notion importante en P.O.O. est l'héritage. Elle permet la définition d'une nouvelle « *classed'objet* » à partir d'une « *classe d'objet* » existante. Il est alors possible de lui adjoindre de nouvelles données, de nouvelles fonctions membres (procédures) pour la spécialiser.

4.16.2 Différences entre C et C++

Nous allons parler ici d'un certain nombre de différences existant entre le C et le C++. Nous pourrions d'ailleurs plutôt utiliser le terme d'incompatibilités.

- **Les fonctions :** la définition historique des fonctions n'est plus acceptée en C++. La syntaxe :

```
int calcule_somme(a, b)
int a;
int b;
{
     return(a+b) ;
}
```

  sera refusée par le compilateur. En C++, seule la syntaxe suivante reste valide :

```
int calcule_somme(int a, int b)
{
     return(a+b) ;
}
```

- **Const :** le C++ a quelque peu modifié l'utilisation "C" de ce qualificatif `const`. Pour rappel, « `const` » est utilisé pour définir une constante. C'est une bonne alternative à un `#define`. La portée en C++ est désormais plus locale. En C, un `const` permettait pour une variable globale d'être visible partout. C++ limite quant à lui la portée d'une telle variable, au fichier source contenant la déclaration ;

- **Compatibilité de pointeurs :** en C ANSI, un pointeur de type « `void *` » est compatible avec tout autre type de pointeurs, et inversement. Le code suivant est légal en C :

```
void *p_generique;    /* Pointeur générique */
int *p_entier;    /* Pointeur sur un entier */
[...]
p_entier = p_generique;
[...]
p_generique = p_entier;
```

  Ces affectations font intervenir des conversions implicites. En C++, seule la conversion int*->void* est implicite. L'autre reste possible, mais nécessite un "cast" :

```
void *p_generique;    /* Pointeur générique */
int *p_entier;    /* Pointeur sur un entier */
[...]
p_entier = (int *) p_generique; /* cast obligatoire */
[...]
p_generique = p_entier; /* reste OK */
```

- **Nouvelle forme de commentaire :** en C++, tout ce qui suit un double slash (« `//` ») jusqu'à la fin de la ligne est considéré comme un commentaire ;

- **Déclaration des variables :** en C, les déclarations de variables devaient impérativement être en début de bloc. C++ lève cette obligation. Exemple :

```
int f(int a, int b)
{
     int i ;
     i = a + b ;
     b++;
     int j ;  // interdit en C, légal en C++
     j = a + b + i ;
     return(j) ;
}
```

- **Valeur par défaut d'un paramètre :** il est dorénavant possible de donner une valeur par défaut à un paramètre. Exemple :

```
// Si aucun paramètre transmis, choisi 1 par défaut.
int double(int a=1)
{
    return(2 * a) ;
}
[...]
int i = double(2) ; // i vaut 4 ;
int j = double() ; // j vaut double(1), i.e. : 2
```

- **Passage de paramètres par référence** : nous avons vu qu'en C, il était impossible de passer des variables par référence. Si nous voulions qu'une variable passée en paramètre soit « modifiable », il fallait passer en paramètre non pas la variable elle-même, mais un pointeur sur cette variable. Une nouvelle syntaxe apparaît en C++ pour permettre de passer une variable en paramètre afin qu'elle puisse être modifiée par la fonction appelée : dans les paramètres, le nom de la variable doit être précédé par le symbole « & ». Exemple :

```
void  double_parametre(int &a)
{
     a *= 2 ; /* on modifie le contenu de la variable de
la fonction appelante */
}
[...]
int i = 4;
double_parametre(i);
// à ce stade, i vaut 8
```

  Important !!! Evidement, le paramètre passé à une fonction prototypée pour accepter un paramètre par référence doit être une « *lvalue* » (variable modifiable). En effet, dans l'exemple ci-dessus, le code « `double_parametre(4);` » n'aurait aucun sens.

- **Opérateurs `new` et `delete` :** classiquement, pour allouer de la mémoire afin de stocker des valeurs d'un type quelconque, il fallait utiliser les fonctions de la bibliothèque standard `malloc()` et `free()`. Le C++ propose un nouveau jeu d'opérateurs d'allocation/désallocation de mémoire : `new` et `delete`. Ils ont été créés principalement pour la gestion dynamique des objets, mais on peut les utiliser également pour des variables simples. Exemple :

```
int *p ; // à ce stade, p pointe on ne sait où...
p = new int ; /* dorénavant, p pointe vers un espace
mémoire réservé pour stocker une valeur de type int */
*p = 4 ;
[...]
delete p ; // libère la mémoire réservée par p
```

- **Les flux :** C++ introduit la notion de « flux » pour gérer les entrées/sorties. Deux opérateurs binaires sont introduits pour gérer les flux : « >> » permet de récupérer de l'information en provenance d'un flux, et « << » permet d'envoyer une information à un flux. Ces opérateurs se lisent de gauche à droite, et le résultat est un flux. Exemple :

```
// include indispensable pour cout, cin, et endl
#include <iostream.h>
int main()
{
     int n;
     cout << "Entrez un entier : ";
     cin >> n;
     cout << "Vous avez entré : " << n << endl;
     // rq : endl vaut en fait '\n'
     return(0) ;
}
```

- **Les classes :** la programmation par objet en C++ se fait à l'aide de l'utilisation du mot clé `class`. Son emploi ressemble à celui de `struct`, sauf qu'il permet non seulement de définir des membres (appelés variables d'instance en PPO), mais aussi des fonctions (appelées méthodes). Exemple :

```
class complexe
{
  public :
     double re;
     double im;
     void init(double r, double i) // Initialisation
     {
          re = r ;
          im = i ;
     }
     void ajoute(complexe);
};
```

Le mot clé `public` signifie que les variables d'instance qui suivent sont accessibles depuis la routine qui crée un objet. Ainsi, si `z` est un objet de type `complexe`, nous pourrons écrire « `z.re=4. ;` ». Si nous avions mis le mot clé `protected`, cet accès de l'extérieur de la classe aurait été interdit. Dans l'exemple ci-dessu, la fonction addition() est déclarée, mais pas définie. En effet, par souci de lisibilité, toutes les méthodes ne sont pas nécessairement définies dans la classe. La syntaxe pour la définir ailleurs (par exemple dans un autre fichier) est :

```
void complexe::addition(complexe z)
{
  [...]
}
```

Remarque : bien que réagissant exactement de la même façon, ces deux présentations possèdent une subtile différente. La première forme (quand la méthode est définie dans la classe) est dite « **inline** ». L'utilisation de la méthode ne se fera pas par un appel à une fonction, mais le code de la méthode sera inséré à chaque appel (comme pour les macros). Le résultat sera moins bon en terme de place occupée en mémoire (pas de mutualisation du code), mais sera meilleur en terme de performance.

L'utilisation d'une classe se fait ainsi :

```
complexe z1, z2, z3 ;
z1.init(2., 3.) ;
z2.init(-1., 0) ;
z1.ajoute(z2) ; // signifie z1 += z2
```

- **Constructeur, destructeur :** la méthode qui porte le même nom que la classe s'appelle le constructeur. Elle est appelée à chaque fois qu'un objet de ce type est créé (permettant ainsi l'initialisation automatique de l'objet). La méthode qui porte le même nom, et qui est précédé d'un « ~ » s'appelle le destructeur. Elle est appelée dès qu'un objet est détruit. Constructeur et destructeur ne doivent pas retourner de type. Il est possible d'avoir plusieurs constructeur, pouvant avoir des paramètres en nombre et de type différents. Exemple :

```
class complexe
{
  public :
     double re;
     double im;
     // crée un complexe initialisé à 0 :
     complexe()
     {
          re = im = 0. ;
     }
     // crée un complexe initialisé aux valeurs données:
     complexe(double r, double i)
     {
          re = r ;
          im = i ;
     }
     // le destructeur (qui ne fait rien dans ce cas) :
     ~complexe()
     {
     }
};
```

Le constructeur est utilisé par exemple lors de l'utilisation du créateur new :

```
complexe *ptr_z1, *ptr_z2 ;
ptr_z1=new complexe ; // c'est complexe() qui est appelé
ptr_z2=new complexe(1., 2.) ;
```

- **Le mot clé `this` :** ce mot clé, utilisé dans une classe, est le pointeur qui pointe vers l'objet lui-même. Exemple :

```
void complexe::addition(complexe z)
{
   this->re += z.re ;
   this->im += z.im ;
   // Ce qui équivaut tout simplement à :
   // re += z.re ;
   // im += z.im ;
}
```

- **Surdéfinition d'un opérateur :** il est possible de redéfinir les opérateurs sur une classe avec le mot clé `operator`. Par exemple, pour définir l'opérateur « + » sur deux complexes, la synthaxe est :

```
class complexe
{
  public :
     double re;
     double im;
     // crée un complexe initialisé à 0 :
     complexe()
     {
          re = im = 0. ;
     }
     complexe operator + (complexe z)
```

```
        {
            im += z.im;
            re += z.re;
        }
        complexe operator = (complexe z)
        {
            im = z.im;
            re = z.re;
        }

};
[...]
// Utilisation :
complexe z1, z2, z3;
[...]
z1 = z2 + z3 ;
```

- L'héritage : l'héritage se fait assez simplement en C++. Il utilise la notation « : ». Exemple :

```
// Définition d'une classe point :
class point
{
    int x ;
    int y ;
    int est_allume ;
    point()
    {
        x = y = est_allume = 0 ;
    }
    point(int x0, int y0)
    {
        x = x0 ;
        y = y0 ;
        est_allume = 0 ;
    }
    ~point()
    {
    }
    void allume()
    {
        est_alume = 1 ;
    }
    void eteint ()
    {
        est_alume = 0 ;
    }
} ;
// Définition d'une classe point_colore qui hérite
// de la classe point :
class point_colore : public point
{
    int rouge;
    int vert;
    int bleu;
    point_colore()
    {
        rouge = vert = bleu = x = y = est_allume = 0 ;
    }
```

```
        point_colore(int x0, int y0, int r0, int v0,
                                                int b0)
        {
            x = x0 ;
            y = y0 ;
            rouge = r0 ;
            vert = v0 ;
            bleu = b0 ;
            est_allume = 0 ;
        }
        void change_couleur(int r, int v, int b)
        {
            rouge = r ;
            vert = v ;
            bleu = b ;
        }
};
```

Dans cet exemple, `point_colore` hérite publiquement de `point` (tous les membres publics de `point` restent publics dans `point_colore`). Les constructeurs ont été redéfinis. Une nouvelle méthode `change_couleur()` est définie. Par contre, le destructeur, et les méthodes `allume()` et `eteint()` n'ont pas été redéfinies. Aussi, elles peuvent être utilisées sur un objet de classe point_colore de la même façon qu'elles étaient utilisées avec un objet de classe point (c'est le même code qui est utilisé).

Ces quelques pages vous auront permis de survoler les possibilités offertes par le C++. Il serait certainement possible de rentrer encore bien plus dans les détails dans les subtilités de ce langage (règles d'héritage en fonction de l'utilisation de l'utilisation de `public`/`protected`, utilisation de *fonction virtuelles*, définitions de `template`, etc.).

Néanmoins, nous dépasserions l'objectif que nous nous étions fixé, qui sont :

- connaître toutes les subtilités du langage C (en particulier, les spécificités liées aux pointeurs, aux tableaux, aux structures, etc.), afin de pouvoir lire mais aussi concevoir du code en rapport avec la programmation système ;
- et avoir les notions de C++ qui nous permettent de lire le code source de certaines parties de systèmes d'exploitation écrits dans ce langage.

4.17 Travaux dirigés

1) Ecrire en C un programme capable de simuler une pile d'entiers (`int`). La pile contiendra au maximum 100 entiers ; aussi, le candidat pourra utiliser un tableau de 100 entiers pour stocker le contenu de la pile. Les fonctions qui devront être définies seront :
    - `cree_pile()` : crée une pile d'entier ;
    - `detruit_pile()` : détruit une pile, même s'il reste des éléments dedans ;
    - `empile()` : permet d'empiler un entier dans une pile existante ;
    - `depile()` : dépile le dernier entier qui a été ajouté à une pile. Retourne la valeur de cet entier ;
    - `est_vide()` : retourne 0 si la pile est vide, et 1 sinon.
2) Même question, mais cette fois-ci, sans présumer d'une taille maximale de la pile. La pile pourra potentiellement contenir un très grand nombre d'éléments entiers, sans qu'il soit possible d'estimer ce nombre à l'avance. Le candidat aura besoin de lire la

documentation des fonctions `malloc()` et `free()`, qui permettent l'allocation dynamique et la restitution de mémoire.

3) Ecrire en C un programme capable de simuler une pile d'objets de n'importe quel type (simple entier, double, structure, union, etc.). Ce code utilisera la notion de « pointeur de fonction » afin de fournir en paramètre la fonction qui permet de copier les objets qui sont empilés.

Correction *rapide* et peu commentée de l'exercice numéro 2)

Fichier « pile.h » :

```
#ifndef __PILE_H_
#define __PILE_H_

struct struct_pile
{
        struct struct_pile  *suivant;
        int                 valeur_stockee;
};
typedef struct struct_pile pile;
#define PILE_VIDE       ((struct struct_pile *)NULL)
pile    *cree_pile();
void    detruit_pile(pile *);
pile    *empile(pile *, int);
int     depile(pile **);
int     est_vide(pile *);

#endif
```

Fichier « pile.c » :

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

pile    *cree_pile()
{
        return(PILE_VIDE);
}

void    detruit_pile(pile *p)
{
        struct struct_pile *temp_p;
        while (p != PILE_VIDE)
        {
                temp_p = p;
                p = p->suivant;
                free(temp_p);
        }
}

pile    *empile(pile *p, int nouvelle_val)
{
        pile *temp_p;
        temp_p = (pile *)malloc(sizeof(pile));
        if (temp_p == NULL)
```

```
        {
                printf("Plus assez de mémoire !!!");
                exit(-1);
        }
        temp_p->suivant = p;
        temp_p->valeur_stockee = nouvelle_val;
        return(temp_p);
}

int     depile(pile **p)
{
        pile *temp_p;
        int  val_ret;
        if ( (*p) == PILE_VIDE )
                return(0);
        temp_p = *p;
        val_ret = temp_p->valeur_stockee;
        (*p) = (*p)->suivant;
        free(temp_p);
        return(val_ret);
}

int     est_vide(pile *p)
{
        return(p == PILE_VIDE);
}
```

Fichier « main.c » :

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

int     main()
{
        pile    *p;
        p = cree_pile();
        printf("Est-ce qu'une pile nouvellement crée est vide : ");
        if (est_vide(p))
                printf("oui\n");
        else
                printf("non\n");
        printf("J'empile 2\n");
        p = empile(p, 2);
        printf("J'empile -1\n");
        p = empile(p, -1);
        printf("J'empile 0\n");
        p = empile(p, 0);
        printf("Valeur dépilée : %d\n", depile(&p));
        printf("Est-ce que la pile est maintenant vide : ");
        if (est_vide(p))
                printf("oui\n");
        else
                printf("non\n");
        printf("Valeur dépilée : %d\n", depile(&p));
        printf("Est-ce que la pile est maintenant vide : ");
        if (est_vide(p))
                printf("oui\n");
```

```
        else
                printf("non\n");
        printf("Valeur dépilée : %d\n", depile(&p));
        printf("Est-ce que la pile est maintenant vide : ");
        if (est_vide(p))
                printf("oui\n");
        else
                printf("non\n");
        detruit_pile(p);
        return(0);
}
```

Compilation et utilisation sous Linux :

```
$> cc -Wall -O2 pile.c main.c -o test

$> ./test
Est-ce qu'une pile nouvellement crée est vide : oui
J'empile 2
J'empile -1
J'empile 0
Valeur dépilée : 0
Est-ce que la pile est maintenant vide : non
Valeur dépilée : -1
Est-ce que la pile est maintenant vide : non
Valeur dépilée : 2
Est-ce que la pile est maintenant vide : oui

$>
```

5 Introduction : architecture des ordinateurs

5.1 Schéma de base

Dans sa version moderne, un ordinateur est constitué des éléments suivants :

- un ou plusieurs microprocesseurs (CPU) ;
- de la mémoire vive (RAM),
- des périphériques, qui dialoguent avec le CPU et la RAM via des « contrôleurs de périphérique ».

Le dialogue entre ces différents éléments se fait via des « *bus* » (bus = canal de communication entre plusieurs composants). Comme il existe plusieurs type de bus (PCI, ISA, USB, etc.), l'interconnexion entre bus se fait à travers des « *ponts* ». Aujourd'hui, ces ponts sont implémentés dans un « *chipset* ».

5.2 Le microprocesseur (CPU)

5.2.1 Présentation

Le processeur (CPU, pour *Central Processing Unit*, soit *Unité Centrale de Traitement*) est le cerveau de l'ordinateur. Il permet de manipuler des informations numériques, c'est-à-dire des informations codées sous forme binaire, et d'exécuter les instructions stockées en mémoire.

Le premier microprocesseur (Intel 4004) a été inventé en 1971. Il s'agissait d'une unité de calcul de 4 bits, cadencé à 108 kHz. Depuis, la puissance des microprocesseurs augmente exponentiellement (Cf. loi empirique de Moore : « la complexité des semi-conducteurs proposés en entrée de gamme double tous les dix-huit mois à coût constant depuis 1959, date de leur invention ».

5.2.2 Fonctionnement

Le CPU est un circuit électronique cadencé au rythme d'une « horloge interne », grâce à un cristal de quartz qui, soumis à un courant électrique, envoie des impulsions, appelées « **top** ». La **fréquence d'horloge** (appelée également cycle), correspondant au nombre d'impulsions par seconde, s'exprime en Hertz (Hz). Ainsi, un ordinateur à 200 MHz possède une horloge envoyant 200 000 000 de battements par seconde. La fréquence d'horloge est généralement un multiple de la fréquence du système (FSB, Front-Side Bus), c'est-à-dire un multiple de la fréquence de la carte mère

A chaque top d'horloge le processeur exécute une action, correspondant à une instruction ou une partie d'instruction. L'indicateur appelé **CPI** (Cycles Par Instruction) permet de représenter le nombre moyen de cycles d'horloge nécessaire à l'exécution d'une instruction sur un microprocesseur. La puissance du processeur peut ainsi être caractérisée par le nombre d'instructions qu'il est capable de traiter par seconde. L'unité utilisée est le **MIPS** (Millions d'Instructions Par Seconde) correspondant à la fréquence du processeur que divise le CPI.

Définition **:** une **instruction** est l'opération élémentaire que le processeur peut accomplir. Les instructions sont stockées dans la **mémoire principale** (RAM), en vue d'être traitée par le processeur. Une instruction est composée de deux champs :

- **le code opération**, représentant l'action que le processeur doit accomplir ;

- **le code opérande**, définissant les paramètres de l'action. Le code opérande dépend de l'opération. Il peut s'agir d'une donnée ou bien d'une adresse mémoire.

Le nombre d'octets d'une instruction est variable selon le type de donnée et selon de type de CPU (l'ordre de grandeur est de 1 à 4 octets).

Les instructions peuvent être classées en catégories dont les principales sont :

- **Accès à la mémoire** : des accès à la mémoire ou transferts de données entre registres.
- **Opérations arithmétiques** : opérations telles que les additions, soustractions, divisions ou multiplication.
- **Opérations logiques** : opérations ET, OU, NON, NON exclusif, etc.
- **Contrôle** : contrôles de séquence, branchements conditionnels, etc.

Définition : lorsque le processeur exécute des instructions, les données sont temporairement stockées dans de petites mémoires rapides de 8, 16, 32 ou 64 bits, située à l'intérieur du microprocesseur, que l'on appelle **registres**. Suivant le type de processeur le nombre global de registres peut varier d'une dizaine à plusieurs centaines.

Les principaux registres sont :

- le **registre accumulateur** (ACC), stockant les résultats des opérations arithmétiques et logiques ;
- le **registre d'état** (PSW, *Processor Status Word*), permettant de stocker des indicateurs sur l'état du système (retenue, dépassement, etc.) ;
- le **registre instruction** (RI), contenant l'instruction en cours de traitement ;
- le **compteur ordinal** (CO ou PC pour *Program Counter*), contenant l'adresse de la prochaine instruction à traiter ;
- le **registre tampon**, stockant temporairement une donnée provenant de la mémoire.

5.2.3 Mémoire cache

Définition : la **mémoire cache** (également appelée **antémémoire** ou **mémoire tampon**) est une mémoire rapide permettant de réduire les délais d'attente des informations stockées en mémoire vive.

En effet, depuis quelques années, la mémoire centrale de l'ordinateur possède une vitesse bien moins importante que le processeur. Il existe néanmoins des mémoires beaucoup plus rapides, mais dont le coût est très élevé. La solution consiste donc à inclure ce type de mémoire rapide à proximité du processeur et d'y stocker temporairement les principales données devant être traitées par le processeur. Les ordinateurs récents possèdent plusieurs niveaux de mémoire cache :

- La **mémoire cache de premier niveau** (appelée **Cache L1**, pour Level 1 Cache) est directement intégrée dans le processeur. Dans certains modèles de microprocesseurs (ça n'a pas toujours été le cas), elle se subdivise en 2 parties :
    - le **cache d'instructions**, qui contient les instructions issues de la mémoire vive décodées lors de passage dans les « pipelines » (Cf. ci-dessous) ;
    - la seconde est le **cache de données**, qui contient des données issues de la mémoire vive et les données récemment utilisées lors des opérations du processeur.

  Les caches du premier niveau sont très rapides d'accès : leur délai d'accès tend à s'approcher de celui des registres internes aux processeurs ;

- La **mémoire cache de second niveau** (appelée **Cache L2**, pour Level 2 Cache) est située au niveau du boîtier contenant le processeur (dans la puce). Le cache de second niveau vient s'intercaler entre le processeur avec son cache interne et la mémoire vive. Il est plus rapide d'accès que cette dernière mais moins rapide que le cache de premier niveau ;
- La **mémoire cache de troisième niveau** (appelée **Cache L3**, pour Level 3 Cache) est située au niveau de la carte mère.

Tous ces niveaux de cache permettent de réduire les temps de latence des différentes mémoires lors du traitement et du transfert des informations. Pendant que le processeur travaille, le contrôleur de cache de premier niveau peut s'interfacer avec celui de second niveau pour faire des transferts d'informations sans bloquer le processeur. De même, le cache de second niveau est interfacé avec celui de la mémoire vive (cache de troisième niveau), pour permettre des transferts sans bloquer le fonctionnement normal du processeur.

5.2.4 Signaux de commande

Les **signaux de commande** sont des signaux électriques permettant d'orchestrer les différentes unités du processeur participant à l'exécution d'une instruction. Les signaux de commandes sont distribués grâce à un élément appelé **séquenceur**. Le signal *Read/Write* (en français *lecture/écriture*), permet par exemple de signaler à la mémoire que le processeur désire lire ou écrire une information.

5.2.5 Unités fonctionnelles

En pratique, le microprocesseur est constitué d'un ensemble d'**unités fonctionnelles** reliées entre elles. L'architecture d'un microprocesseur est très variable d'une architecture à une autre. Cependant, les principaux éléments d'un microprocesseur sont les suivants :

- Une **unité d'instruction** (ou unité de commande, en anglais **control unit**) qui lit les données entrant dans le CPU, les décode, puis les envoie à l'unité d'exécution ;

  L'unité d'instruction est notamment constituée des éléments suivants :

  - le **séquenceur** (ou bloc logique de commande) chargé de synchroniser l'exécution des instructions au rythme d'une horloge. Il est ainsi chargé de l'envoi des signaux de commande ;
  - le **compteur ordinal** contenant l'adresse de l'instruction en cours ;
  - le **registre d'instruction** contenant l'instruction suivante.
- Une **unité d'exécution** (ou **unité de traitement**), qui accomplit les tâches que lui a données l'unité d'instruction. L'unité d'exécution est notamment composée des éléments suivants :
  - L'**unité arithmétique et logique** (notée **UAL** ou en anglais **ALU** pour *Arithmetical and Logical Unit*). L'UAL assure les fonctions basiques de calcul arithmétique et les opérations logiques (ET, OU, Ou exclusif, etc.),
  - L'**unité de virgule flottante** (notée **FPU**, pour Floating Point Unit), qui accomplit les calculs complexes non entiers que ne peut réaliser l'unité arithmétique et logique,
  - Le **registre d'état**,
  - Le **registre accumulateur**.
- Une **unité de gestion des bus** (ou **unité d'entrées-sorties**), qui gère les flux d'informations entrant et sortant, en interface avec la mémoire vive du système ;

Le schéma ci-dessous donne un schéma simplifié des éléments constituant le CPU :

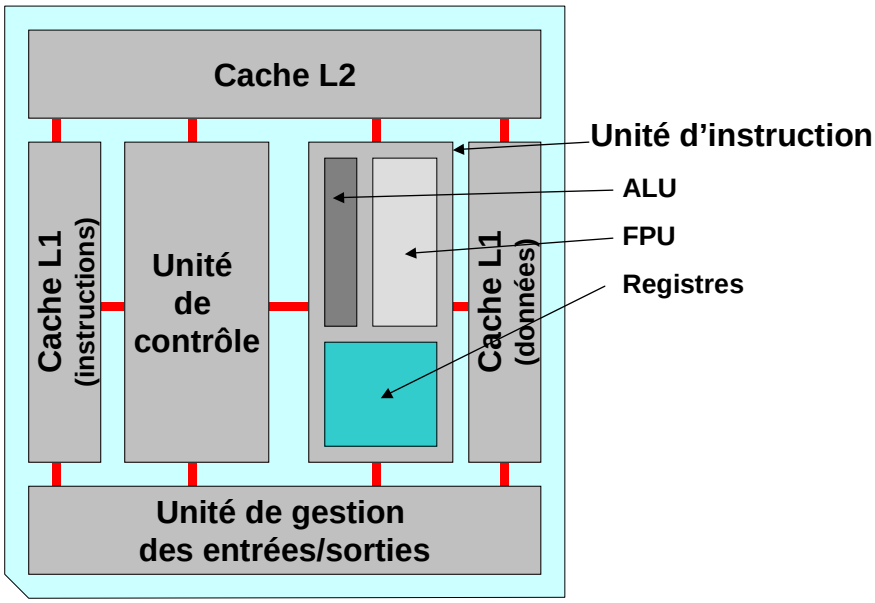


Schéma représentatif d'un processeur

5.2.6 Familles

Chaque type de CPU possède son propre jeu d'instructions (ensemble d'instructions reconnues par le CPU). On distingue ainsi les familles de processeurs suivants, possédant chacun un jeu d'instruction qui leur est propre :

- 80x86 : (« x » représente la famille). Exemple : 386, 486, 586, 686, Pentium, ...,
- ARM,
- IA-64,
- MIPS,
- Motorola 6800,
- PowerPC,
- SPARC,
- ...

Cela explique qu'un programme exécutable compilé et assemblé pour un type de processeur ne puisse fonctionner directement sur un système possédant un autre type de processeur.

5.2.7 Jeu d'instruction, architecture de CPU

On appelle jeu d'instructions l'ensemble des opérations élémentaires qu'un processeur peut accomplir. Le jeu d'instruction d'un processeur détermine ainsi son architecture, sachant qu'une même architecture peut aboutir à des implémentations différentes selon les constructeurs.

Le processeur travaille effectivement grâce à un nombre limité de fonctions, directement câblées sur les circuits électroniques. La plupart des opérations peuvent être réalisées à l'aide de fonctions basiques. Certaines architectures incluent néanmoins des fonctions évoluées courante dans le processeur.

Les deux grandes architectures de CPU sont :

- L'**architecture CISC** (Complex Instruction Set Computer, soit « ordinateur à jeu d'instruction complexe ») consiste à câbler dans le processeur des instructions complexes, difficiles à créer à partir des instructions de base. L'architecture CISC est utilisée en particulier par les processeurs de type 80x86. Ce type d'architecture possède un coût élevé dû aux fonctions évoluées imprimées sur le silicium. D'autre part, les instructions sont de longueurs variables et peuvent parfois nécessiter plus d'un cycle d'horloge. Or, un processeur basé sur l'architecture CISC ne peut traiter qu'une instruction à la fois, d'où un temps d'exécution conséquent ;
- L'**architecture RISC** (Reduced Instruction Set Computer, soit « ordinateur à jeu d'instructions réduit ») n'a pas de fonctions évoluées câblées. Les programmes doivent ainsi être traduits en instructions simples, ce qui entraîne un développement plus difficile et/ou un compilateur plus puissant. Une telle architecture possède un coût de fabrication réduit par rapport aux processeurs CISC. De plus, les instructions, simples par nature, sont exécutées en un seul cycle d'horloge, ce qui rend l'exécution des programmes plus rapide qu'avec des processeurs basés sur une architecture CISC. Enfin, de tels processeurs sont capables de traiter plusieurs instructions simultanément en les traitant en parallèle.

5.2.8 Améliorations technologiques

Ces dernières années, les constructeurs de microprocesseurs (appelés fondeurs), ont mis au point un certain nombre d'améliorations permettant d'optimiser le fonctionnement du processeur :

- **Le parallélisme** : consiste à exécuter simultanément, sur des processeurs différents, des instructions relatives à un même programme. Cela se traduit par le découpage d'un programme en plusieurs processus traités en parallèle afin de gagner en temps d'exécution.

- **Le pipeline** : technologie visant à permettre une plus grande vitesse d'exécution des instructions en parallélisant des étapes.

  Pour comprendre le mécanisme du pipeline, il est nécessaire au préalable de comprendre les phases d'exécution d'une instruction. Les phases d'exécution d'une instruction pour un processeur contenant un pipeline « classique » à 5 étages sont les suivantes :

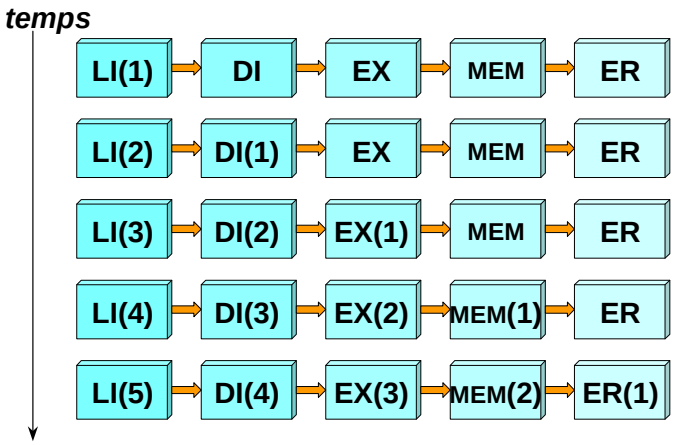
  - **LI** : (Lecture de l'Instruction (en anglais FETCH instruction) depuis le cache ;
  - **DI** : Décodage de l'Instruction (DECODe instruction) et recherche des opérandes (Registre ou valeurs immédiate);
  - **EX** : Exécution de l'Instruction (EXECute instruction) (si ADD, on fait la somme, si SUB, on fait la soustraction, etc.);
  - **MEM** : Accès mémoire (MEMory access), écriture dans la mémoire si nécéssaire ou chargement depuis la mémoire ;
  - **ER** : Ecriture (Write instruction) de la valeur calculée dans les registres.

  Classiquement, les instructions sont organisées en file d'attente dans la mémoire, et sont chargées les unes après les autres.

  Dans un pipeline, le traitement des instructions nécessite au maximum les cinq étapes précédentes. Dans la mesure où l'ordre de ces étapes est invariable (LI, DI, EX, MEM et ER), il est possible de créer dans le processeur un certain nombre de circuits spécialisés pour chacune de ces phases.

  L'objectif du pipeline est d'être capable de réaliser chaque étape en parallèle avec les étapes amont et aval, c'est-à-dire de pouvoir lire une instruction (LI) lorsque la

précédente est en cours de décodage (DI), que celle d'avant est en cours d'exécution (EX), que celle située encore précédemment accède à la mémoire (MEM) et enfin que la première de la série est déjà en cours d'écriture dans les registres (ER).



Enchainement de 5 instructions (numérotées de 1 à 5) dans un pipeline

Il faut compter en général 1 à 2 cycles d'horloge (rarement plus) pour chaque phase du pipeline, soit 10 cycles d'horloge maximum par instruction. Pour deux instructions, 12 cycles d'horloge maximum seront nécessaires (10+2=12 au lieu de 10*2=20), car la précédente instruction était déjà dans le pipeline. Les deux instructions sont donc en traitement dans le processeur, avec un décalage d'un ou deux cycles d'horloge). Pour 3 instructions, 14 cycles d'horloge seront ainsi nécessaires, etc.

- Technologie **superscalaire** (en anglais superscaling) consiste à disposer plusieurs unités de traitement en parallèle afin de pouvoir traiter plusieurs instructions par cycle ;
- La technologie **HyperThreading** (traduisez HyperFlots ou HyperFlux) consiste à définir deux processeurs logiques au sein d'un processeur physique. Ainsi, le système d'exploitation reconnaît deux processeurs physiques et se comporte en système multitâche en envoyant deux threads simultanés (nous reparlerons longuement de ce qu'est un thread dans les prochains chapitres). On parle alors de SMT (Simultaneous Multi Threading). Cette « supercherie » permet d'utiliser au mieux les ressources du processeur en garantissant que des données lui sont envoyées en masse.

5.3 Le bus

5.3.1 Description

Un **bus informatique** désigne l'ensemble des « **lignes de communication** » connectant les différents composants d'un ordinateur.

- **le bus système** (aussi appelé **bus interne**) : il relie le micro-processeur à la mémoire vive ;
- **le bus d'extension** (aussi appelé **bus d'entrées/sorties**) : il relie le micro-processeur aux connecteurs d'entrées/sorties et aux connecteurs d'extension.

Un bus qui n'interconnecte que deux dispositifs est appelé un « **port** ».

Un bus est souvent caractérisé par une *fréquence* et le *nombre de bits* d'informations qu'il peut transmettre simultanément (via des « *lignes* »). Lorsqu'un bus peut transmettre plus d'un bit d'information simultanément, on parlera d'un **bus parallèle**, sinon, d'un **bus série**.

Remarque : la fréquence donnée est tantôt la fréquence du signal électrique sur le bus, tantôt la cadence de transmission des informations, qui peut être un multiple de la fréquence du signal (vitesse de transmission = fréquence d'horloge * nombre de lignes).

Exercice : quelle est la vitesse de transmission (en méga octets par seconde – Mo/s –) d'un bus de 32 bits ayant une fréquence de 66 MHz ?

A ne pas savoir par cœur, voici quelques exemples de vitesse de bus :

Norme	Largeur du bus (bits)	Vitesse du bus (MHz)	Bande passante (Mo/sec)
ISA 8-bit	8	8.3	7.9
ISA 16-bit	16	8.3	15.9
EISA	32	8.3	31.8
VLB	32	33	127.2
PCI 32-bit	32	33	127.2
PCI 64-bit 2.1	64	66	508.6
AGP	32	66	254.3
AGP(x2 Mode)	32	66x2	528
AGP(x4 Mode)	32	66x4	1056
AGP(x8 Mode)	32	66x8	2112
ATA33	16	33	33
ATA100	16	50	100
ATA133	16	66	133
Serial ATA (S-ATA)	1		180
Serial ATA II (S-ATA2)	2		380
USB	1		1.5
USB 2.0	1		60
Firewire	1		100
Firewire 2	1		200
SCSI-1	8	4.77	5
SCSI-2 – Fast	8	10	10
SCSI-2 – Wide	16	10	20
SCSI-2 - Fast Wide 32 bits	32	10	40
SCSI-3 – Ultra	8	20	20
SCSI-3 - Ultra Wide	16	20	40
SCSI-3 - Ultra 2	8	40	40
SCSI-3 - Ultra 2 Wide	16	40	80
SCSI-3 - Ultra 160 (Ultra 3)	16	80	160
SCSI-3 - Ultra 320 (Ultra 4)	16	80 DDR	320
SCSI-3 - Ultra 640 (Ultra 5)	16	80 QDR	640

5.3.2 Matériel

D'un point de vue physique, un bus est un ensemble de conducteurs électriques parallèles. À chaque cycle de temps, chaque conducteur transmet un bit.

En général, l'élément qui écrit sur le bus positionne une tension de 0 volt pour indiquer un « zéro binaire », et une tension choisie conventionnellement par le constructeur pour

indiquer le « un binaire ». Les éléments qui « écoutent » sur le bus mettent leur ligne en position « impédance infinie » pour ne pas entraver ce signal.

Pour les bus parallèles (exemple : ceux entre le CPU, la mémoire, les éléments d'entrée/sortie) on en général une taille de 8 bits, 16 bits, 32 bits, 64 bits ou plus.

Certains conducteurs supplémentaires sont affectés à la transmission des signaux de contrôles de l'état du bus (horloge, demande d'utilisation du bus, etc).

5.3.3 Dans un ordinateur

Dans un ordinateur, la lecture et l'écriture entre le CPU et la mémoire vive (ou entre un périphérique et la mémoire vive, ou entre le CPU et un périphérique) se fait sur trois bus distincts :

- un **bus de données**,
- un **bus d'adresse**,
- un **bus de contrôle**.

Le bus d'adresse est utilisé pour sélectionner la ou les cellules mémoires qui doivent être lues ou écrites. Le bus de données transmet le contenu de la mémoire elle-même (ou la donnée à inscrire en mémoire). Le bus de contrôle permet de gérer la communication sur le bus (demande d'utilisation du bus, horloge, etc).

5.4 Le chipset

On appelle **chipset** (en français « *jeu de composants* ») l'élément chargé d'aiguiller les informations entre les différents bus de l'ordinateur, afin de permettre à tous les éléments constitutifs de l'ordinateur de communiquer entre eux. Le chipset était originalement composé d'un grand nombre de composants électroniques, ce qui explique son nom. Aujourd'hui, dans un PC, il est généralement composé de deux éléments :

- le **NorthBridge** (Pont Nord ou Northern Bridge, appelé également contrôleur mémoire) est chargé de contrôler les échanges entre le processeur et la mémoire vive. C'est la raison pour laquelle il est situé géographiquement proche du processeur. Il est parfois appelé GMCH, pour *Graphic and Memory Controller Hub* ;
- le **SouthBridge** (Pont Sud ou Southern Bridge, appelé également contrôleur d'entrée-sortie ou contrôleur d'extension) gère les communications avec les périphériques d'entrée-sortie. Le pont sud est également appelé ICH (I/O Controller Hub).

On parle généralement de bridge (en français pont) pour désigner un élément d'interconnexion entre deux bus.

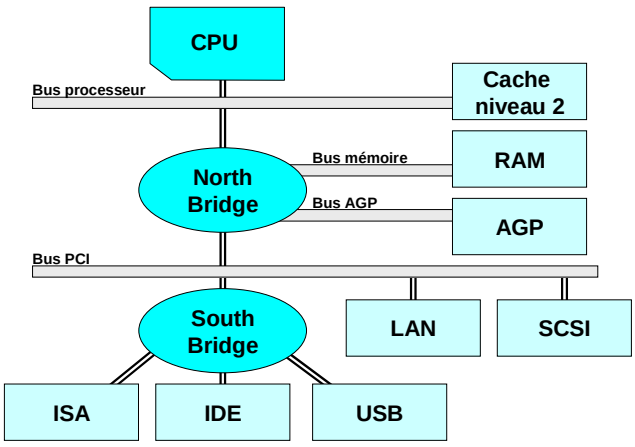


Schéma d'un chipset de PC

5.5 La mémoire

On appelle « *mémoire* » tout composant électronique capable de stocker temporairement des données. On distingue ainsi deux grandes catégories de mémoires :

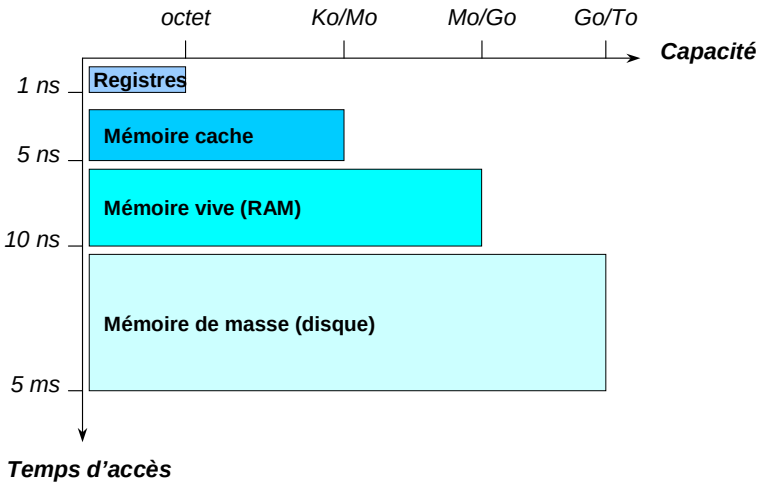
- la mémoire centrale (appelée également mémoire interne) permettant de mémoriser temporairement les données lors de l'exécution des programmes. La mémoire centrale est réalisée à l'aide de micro-conducteurs, c'est-à-dire des circuits électroniques spécialisés rapides. La mémoire centrale correspond à ce que l'on appelle la mémoire vive ;
- la mémoire de masse (appelée également mémoire physique ou mémoire externe) permettant de stocker des informations à long terme, y compris lors de l'arrêt de l'ordinateur. La mémoire de masse correspond aux dispositifs de stockage magnétiques, tels que le disque dur, aux dispositifs de stockage optique, correspondant par exemple aux CD-ROM ou aux DVD-ROM, ainsi qu'aux mémoires mortes.

5.5.1 Caractéristiques techniques

Les principales caractéristiques d'une mémoire sont les suivantes :

- **la capacité**, représentant le volume global d'informations (en bits) que la mémoire peut stocker ;
- **le temps d'accès**, correspondant à l'intervalle de temps entre la demande de lecture/écriture et la disponibilité de la donnée ;
- **le temps de cycle**, représentant l'intervalle de temps minimum entre deux accès successifs ;
- **le débit**, définissant le volume d'information échangé par unité de temps, exprimé en bits par seconde ;
- **la non volatilité**, caractérisant l'aptitude d'une mémoire à conserver les données lorsqu'elle n'est plus alimentée électriquement.

Ainsi, la mémoire idéale possède une grande capacité avec des temps d'accès et temps de cycle très restreints, un débit élevé et est non volatile.



Vitesse et temps d'accès des différents types de mémoire

Néanmoins les mémoires rapides sont également les plus onéreuses. C'est la raison pour laquelle des mémoire utilisant différentes technologiques sont utilisées dans un ordinateur, interfacées les unes avec les autres et organisées de façon hiérarchique.

5.5.2 Temps d'accès et capacité des différents types de mémoire

Les mémoires les plus rapides (aussi les plus coûteuses) sont situées en faible quantité à proximité du processeur et les mémoires de masse, moins rapides (et moins coûteuses), servent à stocker les informations de manière permanente.

Parmi les types de mémoires, nous pouvons noter :

- **La mémoire vive** : généralement appelée RAM (Random Access Memory, traduisez mémoire à accès direct), c'est la mémoire principale du système, c'est-à-dire qu'il s'agit d'un espace permettant de stocker de manière temporaire des données lors de l'exécution d'un programme ;
- **Mémoire morte :** la mémoire morte, appelée ROM pour Read Only Memory (traduisez mémoire en lecture seule) est un type de mémoire permettant de conserver les informations qui y sont contenues même lorsque la mémoire n'est plus alimentée électriquement. A la base ce type de mémoire ne peut être accédée qu'en lecture. Toutefois il est désormais possible d'enregistrer des informations dans certaines mémoires de type ROM.
- **Mémoire flash** : la mémoire flash est un compromis entre les mémoires de type RAM et les mémoires mortes. En effet, la mémoire Flash possède la non-volatilité des mémoires mortes tout en pouvant facilement être accessible en lecture ou en écriture. En contrepartie les temps d'accès des mémoires flash sont plus importants que ceux de la mémoire vive.

5.6 Communication périphériques/CPU

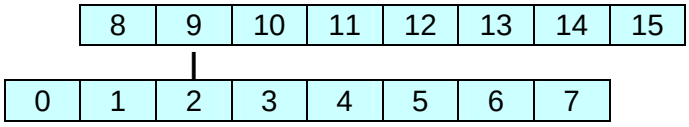
5.6.1 Les interruptions matérielles (et IRQ)

Puisque le CPU ne peut pas traiter plusieurs informations simultanément (il traite une information à la fois, le multitâche consiste à alterner des morceaux d'instructions de plusieurs tâches différentes de façon très rapide, afin que l'utilisateur ait une impression de simultanéité), un programme en cours d'exécution peut, grâce à une **interruption**, être momentanément suspendu, le temps que s'exécute une *routine d'interruption*. Le programme interrompu peut ensuite reprendre son exécution.

Une interruption devient une **interruption matérielle** lorsqu'elle est demandée par un composant matériel de l'ordinateur.

Ainsi, lorsque lorsqu'un périphérique a besoin d'une ressource, il envoie parfois au CPU une demande d'interruption pour que ce dernier lui prête son attention.

En pratique, dans un PC classique, les périphériques ont un numéro d'interruption, que l'on appelle **IRQ** (*Interruption request*, ce qui signifie «*requête d'interruption*»). Physiquement, chaque IRQ correspond à un signal sur une ligne. Par exemple, historiquement, sur un bus ISA, il y a une ligne qui signifie « demande d'interruption ». Et sur le bus de données, le ou les lignes à « 1 » sur le canal de données indiquai(en)t le numéro d'IRQ (de 0 à 7). Comme plusieurs demandes d'interruptions peuvent être faites simultanément, un « contrôleur d'interruption » permet de prendre en compte l'IRQ ayant la plus haute priorité. Avec l'arrivée des bus 16 bits est apparu un second contrôleur d'interruptions. La liaison entre les deux groupes d'interruptions se fait par l'intermédiaire de l'IRQ 2 reliée à l'IRQ 9 (et est appelée «cascade»). La cascade vient donc en quelque sorte "insérer" les IRQ 8 à 15 – sauf le 9 – entre les IRQ 1 et 3 :



Principe de cascade d'IRQ

Sur un PC, pour un contrôleur d'interruption donné, les IRQ ayant un le numéro le plus petit est prioritaire. Avec le principe de cascade, les priorités des IRQ sont classées ainsi :

0 > 1 > 8 > 9 > 10 > 11 > 12 > 13 > 14 > 15 > 3 > 4 > 5 > 6 > 7

5.6.2 Les adresses de base

Les périphériques ont besoin d'échanger des informations avec le CPU. Des plages d'adresses mémoire leur sont assignées pour l'envoi et la réception de données. Ces adresses sont appelées « *adresses de base* » (les termes suivants sont également parfois utilisés : « *ports d'entrée/sortie* », « *ports d'E/S* », « *adresse d'E/S* », « ports de base », ou en anglais « *I/O address* » ou «*Input/Output Address* »).

C'est par l'intermédiaire de cette adresse de base que le périphérique peut communiquer avec le système d'exploitation. Il n'existe qu'une adresse de base unique par périphérique.

5.6.3 Utilisation d'un canal DMA

Des périphériques ont régulièrement besoin d'accéder à une zone mémoire afin de s'en servir comme zone de tampon (en anglais *buffer*), c'est-à-dire une zone de stockage temporaire permettant d'enregistrer rapidement des données en entrée ou en sortie.

Un **canal d'accès direct à la mémoire**, appelé **DMA** (*Direct Memory Access* soit « *Accès direct à la mémoire* »), a ainsi été défini pour y remédier.

Le canal DMA désigne un accès à un emplacement de la mémoire vive (RAM) de l'ordinateur, repéré par une « adresse de début » (ou «RAM Start Address» en anglais) et une « adresse de fin ». Cette méthode permet à un périphérique d'emprunter des canaux spéciaux qui lui donnent un accès direct à la mémoire, sans faire intervenir le microprocesseur, afin de le décharger de ce dernier.

Un ordinateur de type PC possède 8 canaux DMA. Les quatre premiers canaux DMA ont une largeur de bande de 8 bits tandis que les DMA 4 à 7 ont une largeur de bande de 16 bits.

6 Le système d'exploitation

6.1 Définition

Le **système d'exploitation** (SE, en anglais « **Operating System** » ou **OS**) est un ensemble de programmes responsables de la liaison entre les ressources matérielles d'un ordinateur et les applications de l'utilisateur (traitement de texte, jeu vidéo...).

Il assure le démarrage de l'ordinateur, et fournit aux programmes applicatifs des points d'entrée génériques pour les périphériques.

Il a des objectifs principaux :

- la prise en charge, la gestion, et le partage des ressources d'un ordinateur,
- la construction au dessus de ces ressources d'une « interface standardisée » (au niveau des programmeurs, on parlera plus tard d'API : « Application Programming Interface », ou « Interface de programmation » en français) plus conviviale et plus facile d'emploi pour y accéder.

Le partage des ressources va principalement concerner :

- le partage du CPU,
- le partage de la mémoire centrale,
- le partage des périphériques d'entrée-sortie (clavier, écran, imprimante, webcam...).

Les questions auxquelles le système d'exploitation doit répondre sont :

- Dans le cadre du partage du processeur : parmi tous les programmes chargés en mémoire centrale, lequel doit s'exécuter ?
- Dans le cadre du partage de la mémoire centrale : comment allouer la mémoire centrale aux différents programmes? Comment disposer d'une quantité suffisante de mémoire pour y placer tous les programmes nécessaires à un bon taux d'utilisation du processeur ? Comment assurer la protection entre ces différents programmes utilisateurs (protection = veiller à ce qu'un programme donné n'accède pas à une plage mémoire allouée à un autre programme) ?
- Dans le cadre du partage des périphériques: dans quel ordre traiter les requêtes d'entrées-sorties pour optimiser les transferts?

6.2 Structure générale

Le système d'exploitation réalise donc une couche logicielle placée entre la machine matérielle et les applications. Le système d'exploitation peut être découpé en plusieurs grandes fonctionnalités. Dans une première approche, ces fonctionnalités qui seront étudiées plus en détail dans ce cour sont :

- la fonctionnalité de gestion du processeur : le système doit gérer l'allocation du processeur aux différents programmes pouvant s'exécuter. Cette allocation se fait par le biais d'un **algorithme d'ordonnancement** qui planifie l'exécution des programmes. Dans ce cadre, une exécution de programme est appelée **processus** ;
- la fonctionnalité de gestion des objets externes : comme la mémoire centrale est une mémoire volatile, toutes les données devant être conservées au-delà de l'arrêt de la machine doivent être stockées sur une mémoire de masse non volatile (disque dur, Clé USB, CD-Rom...). La gestion de l'allocation des mémoires de masse ainsi

que l'accès aux données qui y sont stockées s'appuient sur la notion de **fichiers** et de **système de gestion de fichiers** ;

- la fonctionnalité de gestion des entrées-sorties : le système doit gérer l'accès aux périphériques. Il doit faire la liaison entre les appels de haut niveau des programmes utilisateurs (exemple : `fgetc()`) et les opérations de bas niveau de l'unité d'échange responsable du périphérique (unité d'échange clavier) : c'est le **pilote d'entrées-sorties** (***driver***) qui assure ce travail ;
- la fonctionnalité de gestion de la mémoire : le système doit gérer l'allocation de la mémoire centrale entre les différents programmes pouvant s'exécuter, c'est-à-dire qu'il doit trouver une place libre suffisante en mémoire centrale pour que le chargeur puisse y placer un programme à exécuter, en s'appuyant sur les mécanismes matériels sous-jacents de **segmentation** et de **pagination**. Comme la mémoire physique est souvent trop petite pour contenir la totalité des programmes, la gestion de la mémoire se fait selon le principe de la mémoire virtuelle : à un instant donné, seules sont chargées en mémoire centrale les parties de code et données utiles à l'exécution, les autres étant temporairement stockées sur disque ;
- la fonctionnalité de gestion de la concurrence : comme plusieurs programmes coexistent en mémoire centrale, ceux-ci peuvent vouloir communiquer pour échanger des données. Par ailleurs, il faut synchroniser l'accès aux données partagées afin de maintenir leur cohérence. Le système d'exploitation offre des outils de communication et de synchronisation entre processus ;
- la fonctionnalité de gestion de la protection : le système doit fournir des mécanismes garantissant que ses propres ressources (processeur, mémoire, fichiers) ne peuvent être utilisées que par les programmes auxquels les droits nécessaires ont été accordés. Ii faut notamment protéger le système et la machine des programmes utilisateurs (**mode « exécution utilisateur »** et **mode « superviseur »**) ;
- la fonctionnalité d'accès au réseau : des exécutions de programmes placées sur des machines physiques distinctes doivent pouvoir échanger des données. Le système d'exploitation fournit des outils permettant aux applications distantes de dialoguer au travers une couche de protocoles réseau telle que TCP/IP.

Les fonctionnalités du système d'exploitation utilisent les mécanismes offerts par le matériel de la machine physique pour réaliser leurs opérations. Notamment, le système d'exploitation s'interface avec la couche matérielle, par le biais du mécanisme des interruptions (Cf. chapitre précédent), qui lui permet de prendre connaissance des événements survenant au niveau matériel.

Par ailleurs, le système d'exploitation s'interface avec les applications du niveau utilisateur par le biais des fonctions prédéfinies que chacune de ses fonctionnalités offre. Ces fonctions que l'on qualifie de **routines systèmes** constituent les points d'entrées des fonctionnalités du système d'exploitation. Il est possible de les appeler depuis les applications de niveau utilisateur. Ces appels peuvent se faire à deux niveaux :

- dans le code d'un programme utilisateur à l'aide d'un **appel système**, qui n'est autre qu'une forme d'appel de procédure amenant à l'exécution d'une routine système ;
- depuis le « prompt » de l'interpréteur de commandes, à l'aide d'une commande. L'interpréteur de commandes est un outil de niveau utilisateur qui accepte les commandes de l'utilisateur, les analyse et lance l'exécution de la routine système associée ;
- plus récemment, dans les systèmes d'exploitations graphique, en utilisant les mécanismes d'icônes (double clic, menu contextuel, clisser/déposer, etc.).

6.3 Les différents types de systèmes d'exploitation

Les systèmes d'exploitation « ayant à gérer plusieurs tâches » peuvent être classés selon différents types :

- les systèmes à *traitements par lots* ;
- les systèmes *multi-utilisateurs interactifs* ;
- les systèmes *temps-reels*.

6.3.1 Les systèmes à traitements par lots

Les systèmes à traitement par lots (ou systèmes *batch*) constituent en quelque sorte les ancêtres de tous les systèmes d'exploitation. Ils sont nés de l'introduction sur les toutes premières machines de deux programmes permettant une exploitation plus rapide et plus rentable du processeur, en vue d'automatiser les tâches de préparation des travaux à exécuter. Ces deux programmes sont :

- le chargeur dont le rôle a été initialement de charger automatiquement les programmes dans la mémoire centrale de la machine depuis les cartes perforées ou le dérouleur de bandes ;
- le moniteur d'enchaînement de traitements, dont le rôle a été de permettre l'enchaînement automatique des travaux soumis en lieu et place de l'opérateur de la machine.

Le principe du traitement par lots s'appuie sur la composition de lots de travaux ayant des caractéristiques ou des besoins communs, la formation de ces lots visant à réduire les temps d'attente du processeur en faisant exécuter les uns à la suite des autres ou ensemble, des travaux nécessitant les mêmes ressources.

La caractéristique principale d'un système de traitement par lots est qu'il n'y a pas d'interaction possible entre l'utilisateur et la machine durant l'exécution du programme soumis. Le programme est soumis avec ses données d'entrées et l'utilisateur récupère les résultats de l'exécution ultérieurement, soit dans un fichier, soit sous forme d'une impression. C'est le mode de fonctionnement typique des anciens MainFrame.

6.3.2 Les systèmes interactifs

La particularité d'un système d'exploitation interactif est que l'utilisateur de la machine peut interagir avec l'exécution de son programme. Typiquement, l'utilisateur lance l'exécution de son programme et attend, derrière le clavier et l'écran, le résultat de celle-ci. S'il s'aperçoit que l'exécution n'est pas conforme à son espérance, il peut immédiatement agir pour arrêter celle-ci, et analyser les raisons de l'échec.

Puisque l'utilisateur attend derrière son clavier et son écran et que, par nature, l'utilisateur de la machine est un être impatient, le but principal poursuivi par les systèmes interactifs va être d'offrir pour chaque exécution le plus petit temps de réponse possible. Pour parvenir à ce but, la plupart des systèmes interactifs travaillent en temps partagé (impression de multitâche donné en exécutant rapidement de courts fragments de plusieurs programmes les uns à la suite des autres).

En effet, un système en temps partagé permet aux différents utilisateurs de partager l'ordinateur simultanément, tout en ayant par ailleurs la sensation d'être seul à utiliser la machine. Ce principe repose notamment sur un partage de l'utilisation du processeur par les différents programmes des différents utilisateurs. Chaque programme occupe à tour de rôle le processeur pour un court laps de temps (le quantum) et les exécutions se succèdent suffisamment rapidement sur le processeur pour que l'utilisateur ait l'impression que son travail dispose seul du processeur.

6.3.3 Les systèmes « temps-réel »

Les systèmes temps réel sont des systèmes liés au contrôle de procédé pour lesquels la caractéristique primordiale est que les exécutions de programmes sont soumises à des contraintes temporelles, c'est-à-dire qu'une exécution de programme est notamment qualifiée par une date butoir de fin d'exécution, appelée « **échéance** », au-delà de laquelle les résultats de l'exécution ne sont plus valides.

Exemple : programme de contrôle d'un automate d'une chaîne de production, pilotage d'un missile, etc.

Pour garantir le respect de limites ou contraintes temporelles, il est nécessaire que :

- les différents services et algorithmes utilisés s'exécutent en temps borné. Un système d'exploitation temps réel doit ainsi être conçu de manière à ce que les services qu'il propose (accès hardware, etc.) répondent en un temps borné ;
- les différents enchaînements possibles des traitements garantissent que chacun de ceux-ci ne dépassent pas leurs limites temporelles. Ceci est vérifié à l'aide d'un test appelé « *test d'acceptabilité* ».

On distingue deux types de systèmes « temps réel » :

- le temps réel *strict* (ou dur, de l'anglais *hard real-time*) : il ne tolère aucun dépassement de ces contraintes, ce qui est souvent le cas lorsque de tels dépassements peuvent conduire à des situations critiques, voire catastrophiques : pilote automatique d'avion, système de surveillance de centrale nucléaire, etc. ;
- le temps réel *souple* (ou mou, de l'anglais *soft real-time*) : à l'inverse, le temps réel souple s'accommode de dépassements des contraintes temporelles dans certaines limites au-delà desquelles le système devient inutilisable : visioconférence, jeux en réseau, etc.

On peut ainsi considérer qu'un système temps réel strict doit respecter des limites temporelles données même dans la pire des situations d'exécution possibles. En revanche un système temps réel souple doit respecter ses limites pour une moyenne de ses exécutions. On tolère un dépassement exceptionnel, qui sera peut-être rattrapé à l'exécution suivante.

6.4 Le noyau

6.4.1 Définition

Un **noyau** (**kernel** en anglais), est la partie fondamentale de certains systèmes d'exploitation. Elle gère les ressources de l'ordinateur et permet aux différents composants - matériels et logiciels - de communiquer entre eux.

En tant que partie du système d'exploitation, le noyau fournit des mécanismes d'abstraction du matériel, notamment de la mémoire, du (ou des) processeur(s), et des échanges d'informations entre logiciels et périphériques matériels. Le noyau autorise aussi diverses abstractions logicielles et facilite la communication entre les processus.

Le noyau d'un système d'exploitation est lui-même un logiciel, mais ne peut cependant utiliser tous les mécanismes d'abstraction qu'il fournit aux autres logiciels. Son rôle central impose par ailleurs des performances élevées.

La majorité des systèmes d'exploitation sont construits autour de la notion de noyau. L'existence d'un noyau, c'est-à-dire d'un programme unique responsable de la communication entre le matériel et le logiciel, résulte de compromis complexes portant sur des questions de performance, de sécurité et d'architecture des processeurs.

L'existence d'un noyau présuppose une partition virtuelle de la mémoire vive physique en deux régions disjointes, l'une étant réservée au noyau (**l'espace noyau**) et l'autre aux applications (**l'espace utilisateur**). Cette division fondamentale de l'espace mémoire en un espace noyau et un espace utilisateur contribue beaucoup à donner la forme et le contenu actuels des systèmes généralistes (Linux, Windows, Mac OS X, etc.). Le noyau a de grands pouvoirs sur l'utilisation des ressources matérielles, en particulier de la mémoire. Elle structure également le travail des développeurs : le développement de code dans l'espace noyau est à priori plus délicat que dans l'espace utilisateur car la mémoire n'est pas protégée.

Diverses abstractions de la notion d'application sont fournies par le noyau aux développeurs. La plus courante est celle de processus (ou tâche). Le noyau du système d'exploitation n'est en lui même pas une tâche, mais un ensemble de routines pouvant être appelées par les différents processus pour effectuer des opérations requérant un certain niveau de privilèges. Les flots d'exécution dans le noyau sont des continuations des flots d'exécution des processus utilisateurs bloqués lorsqu'ils effectuent des appels systèmes. En général, un processus bloqué ne consomme pas de temps processeur, il est réveillé par le processus système lorsque celui-ci se termine.

Lorsque plusieurs tâches doivent être exécutées de manière parallèle, un noyau multitâche s'appuie sur les notions de :

- commutation de contexte ;
- ordonnancement ;
- temps partagé.

6.4.2 Les différents types de noyaux

Il existe toutes sortes de noyaux, plus ou moins spécialisés. Des noyaux spécifiques à une architecture, souvent monotâches, d'autres généralistes et souvent multitâches et multiutilisateurs. L'ensemble de ces noyaux peut être divisé en deux approches opposées d'architectures logicielles : les **noyaux monolithiques** (**modulaires** ou pas) et les **micro-noyaux**.

Définitions :

- « **Noyaux monolithiques non modulaires** » : certains OS, comme d'anciennes versions de Linux, certains BSD ou certains vieux Unix ont un noyau monolithique. C'est-à-dire que l'ensemble des fonctions du système et des pilotes sont regroupés dans un seul bloc de code et un seul bloc binaire généré à la compilation. De par la simplicité de leur concept mais également de leur excellente vitesse d'exécution, les noyaux monolithiques ont été les premiers à être développés et mis en œuvre. Cependant, au fur et à mesure de leurs développements, les codes des noyaux monolithiques ont augmenté en taille et il s'est avéré difficile de les maintenir. Le support par les architectures monolithiques des chargements à chaud ou dynamiques implique une augmentation du nombre de pilotes matériel compilés dans le noyau, et par suite, une augmentation de la taille de l'empreinte mémoire des noyaux. Celle-ci devint rapidement inacceptable. De plus, les multiples dépendances créées entre les différentes fonctions du noyau empêchaient la relecture et la compréhension du code ;
- **« Noyaux monolithiques modulaires »** : pour répondre aux problèmes des noyaux monolithiques, ces derniers sont devenus modulaires. Dans ce type de noyau, seules les parties fondamentales du système sont regroupées dans un bloc de code unique (monolithique). Les autres fonctions, comme les pilotes matériel, sont regroupées en différents modules qui peuvent être séparés tant du point de vue du code que du point de vue binaire. La très grande majorité des systèmes actuels utilise cette technologie : Linux, la plupart des BSD ou Solaris. Par exemple avec le noyau Linux, certaines parties peuvent être non compilées ou compilées en

tant que modules chargeables directement dans le noyau. La modularité du noyau permet le chargement à la demande de fonctionnalités et augmente les possibilités de configuration. Ainsi les systèmes de fichiers peuvent être chargés de manière indépendante, un pilote de périphérique changé, etc. Les distributions Linux, par exemple, tirent profit des modules chargeables lors de l'installation. L'ensemble des pilotes matériel sont compilés en tant que modules. Le noyau peut alors supporter l'immense variété de matériel trouvé dans les compatibles PC. Après l'installation, lors du démarrage du système, seuls les pilotes correspondant au matériel effectivement présent dans la machine sont chargés en mémoire vive. La mémoire est économisée.

Les noyaux monolithiques modulaires conservent les principaux atouts des noyaux monolithiques purs dont ils sont issus. Ainsi, la facilité de conception et de développement est globalement maintenue et la vitesse d'exécution reste excellente. L'utilisation de modules implique le découpage du code source du noyau en blocs indépendants. Ces blocs améliorent l'organisation et la clarté du code source et en facilitent également la maintenance.

Les noyaux monolithiques modulaires conservent également un important défaut des noyaux monolithiques purs : une erreur dans un module met en danger la stabilité de tout le système. Les tests et certification de ces composants doivent être plus poussés.

- **« Micro-noyaux » :** les systèmes à micro-noyaux cherchent à minimiser les fonctionnalités dépendantes du noyau en plaçant la plus grande partie des services du système d'exploitation à l'extérieur de ce noyau, c'est-à-dire dans l'espace utilisateur. Ces fonctionnalités sont alors fournies par de petits serveurs indépendants possédant souvent leur propre espace d'adressage. Un petit nombre de fonctions fondamentales sont conservées dans un noyau minimaliste appelé « micro-noyau ». L'ensemble des fonctionnalités habituellement proposées par les noyaux monolithiques est alors assuré par les services déplacés en espace utilisateur et par ce micro-noyau. Cet ensemble logiciel est appelé « micro-noyau enrichi ».

  Ce principe a de grands avantages théoriques : en éloignant les services « à risque » des parties critiques du système d'exploitation regroupées dans le noyau, il permet de gagner en robustesse et en fiabilité, tout en facilitant la maintenance et l'évolutivité. En revanche, les mécanismes de communication (IPC) qui deviennent fondamentaux pour assurer le passage de messages entre les serveurs, sont très lourds et peuvent limiter les performances.

6.4.3 Avantages et inconvénients des différents types de noyau

Les avantages théoriques des systèmes à micro-noyaux sont la conséquence de l'utilisation du mode protégé par les services qui accompagnent le micro-noyau. En effet, en plaçant les services non critiques dans l'espace utilisateur, ceux-ci bénéficient de la protection de la mémoire. La stabilité de l'ensemble en est améliorée : une erreur d'un service en mode protégé a peu de conséquences sur la stabilité de l'ensemble de la machine.

De plus, en réduisant les possibilités pour les services de pouvoir intervenir directement sur le matériel, la sécurité du système est renforcée. Le système gagne également en possibilités de configuration. Ainsi, seuls les services utiles doivent être réellement lancés au démarrage. Les interdépendances entre les différents serveurs sont faibles. L'ajout ou le retrait d'un service ne perturbe pas l'ensemble du système. La complexité de l'ensemble est réduite.

Le développement d'un système à micro-noyau se trouve également simplifié en tirant parti à la fois de la protection de la mémoire et de la faible interdépendance entre les services. Les erreurs provoquées par les applications en mode utilisateur sont traitées plus

simplement que dans le mode noyau et ne mettent pas en péril la stabilité globale du système. L'intervention sur une fonctionnalité défectueuse consiste à arrêter l'ancien service puis à lancer le nouveau, sans devoir redémarrer toute la machine.

Les micro-noyaux ont un autre avantage : ils sont beaucoup plus compacts que les noyaux monolithiques. Six millions de lignes de code pour le noyau Linux 2.6.0 contre en général moins de 50 000 lignes pour les micro-noyaux. La maintenance du code exécuté en mode noyau est donc simplifiée. Le nombre réduit de lignes de code peut augmenter la portabilité du système.

L'utilisation de nombreux services dans l'espace utilisateur engendre les deux problèmes suivants :

- la plupart des services sont à l'extérieur du noyau et génèrent un très grand nombre d'appels système ;
- les interfaces de communication entre les services (IPC) sont complexes et trop lourdes en temps de traitement.

Pour ces raisons de performance, les systèmes généralistes basés sur une technologie à micro-noyau, tels que Windows et Mac OS X, n'ont pas un « vrai » micro-noyau enrichi. Ils utilisent un **micro-noyau hybride** : certaines fonctionnalités qui devraient exister sous forme de mini-serveurs se retrouvent intégrées dans leur micro-noyau, utilisant le même espace d'adressage. Pour Mac OS X, cela forme XNU : le noyau monolithique BSD fonctionne en tant que « service » de Mach et ce dernier inclut du code BSD dans son propre espace d'adressage afin de réduire les latences.

Ainsi, les deux approches d'architectures de noyaux, les micro-noyaux et les noyaux monolithiques, considérés comme diamétralement différentes en terme de conception, se rejoignent quasiment en pratique par les micro-noyaux hybrides et les noyaux monolithiques modulaires.

6.5 Linux

Le système d'exploitation Linux est un système multiprogrammé, compatible avec la norme pour les systèmes d'exploitation IEEE-POSIX 1003.1, appartenant à la grande famille des systèmes de type Unix. C'est un système de type interactif qui présente également des aspects compatibles avec la problématique du temps réel faiblement contraint.

Le système Linux est né en 1991 du travail de développement de Linus Torvalds. Initialement créé pour s'exécuter sur des plates-formes matérielles de type IBM/PC/Intel 386, le système est à présent disponible sur des architectures matérielles très diverses telles que SPARC, Alpha, IBMSystem/390, Motorola, etc.

Une des caractéristiques principales de Linux est le libre accès à son code source, celui-ci ayant été déposé sous Licence Publique GNU (GPL). Le code source peut ainsi être téléchargé (http://kernel.org/), étudié et modifié par toute personne désireuse de le faire.

6.5.1 Structure de l'OS Linux

Le système Linux est structure comme un noyau monolithique modulaire qui peut être découpé en quatre grandes fonctions :

- une fonctionnalité de gestion des **processus** qui administre les exécutions de programmes, le **processus** étant l'image dynamique de l'exécution d'un programme ;
- une fonctionnalité de gestion de la mémoire ;
- une fonctionnalité de gestion des fichiers de haut niveau : le **VFS** (**Virtual File System**), qui s'interface avec des gestionnaires de fichiers plus spécifiques de type

Unix, DOS, etc., lesquels s'interfacent eux-mêmes avec les contrôleurs de disques, disquettes, CD-Rom, clés USB, etc.;

- une fonctionnalité de gestion du réseau qui s'interface avec le gestionnaire de protocole puis le contrôleur de la carte réseau.

L'architecture du système Linux peut être ajustée autour du noyau grâce au concept de « *modules chargeables* » (Cf. précédent chapitre).

Lors du paramétrage de la compilation du noyau, il est possible de définir pour chaque module s'il doit être liée de façon statique au noyau (il fait alors partie du noyau monolithique), où s'il doit être lié de façon dynamique (lors de l'insertion d'un périphérique « *plug & play* » par exemple).

Deux commandes permettent alors au super utilisateur (`root`) de gérer le l'activation dynamique d'un module :

- `insmod nom_module` (pour charger un module) ;
- `rmmod nom_module` (pour décharger un module).

6.5.2 Quelques notions fondamentales

Comme nous l'avons déjà évoqué précédemment, le système d'exploitation s'interface avec les applications du niveau utilisateur par le biais des fonctions prédéfinies qualifiées de **routines systèmes**, qui constituent les points d'entrées des fonctionnalités du système.

A ce titre, le noyau Linux ne doit pas être appréhendé comme étant un processus, mais un **gestionnaire de processus**, qui offre des services à ceux-ci.

L'exécution des routines systèmes s'effectue sous un mode privilégié, appelé « **mode superviseur** », ou « *mode maître* », ou « *mode noyau* » (« **kernel mode** »). Lorsqu'un programme est exécuté en mode, aucune restriction ne s'applique à lui. Il peut accéder à toute la mémoire, dialoguer directement avec les périphériques, etc.

Un processus utilisateur s'exécute par défaut selon un mode qualifié de « **mode esclave** » ou « *mode utilisateur* » (« *User Mode* »): ce mode d'exécution est un mode pour lequel les actions pouvant être entreprises par le programme sont volontairement restreintes afin de protéger la machine des actions parfois malencontreuses du programmeur. Notamment, le jeu d'instructions utilisables par le programme en mode utilisateur est réduit, et spécialement les instructions permettant la manipulation des interruptions est interdite.

Remarque : sur les microprocesseurs de la famille Intel 80386 (et tous les successeurs et leurs clones), les niveaux de privilège sont implémentés en attribuant une valeur de 0 à 3 aux objets reconnus par le microprocesseur (segments, tables, etc.), le niveau 0 correspondant au niveau le plus privilégié et le niveau 3 au niveau le moins privilégié. Ces niveaux de privilège affectés aux objets définissent des règles d'accès aux objets ainsi qu'un ensemble d'instructions privilégiées ne pouvant être exécuté que lorsque le niveau courant du processeur est égal à 0. Le système Linux n'utilise que deux de ces niveaux : le niveau 0 correspond au niveau superviseur tandis que le niveau 3 correspond au niveau utilisateur. Les niveaux 1 et 2 ne sont pas pris en compte.

6.5.3 La commutation de contexte

Lorsqu'un processus utilisateur demande l'exécution d'une routine du système d'exploitation par le biais d'un appel système, ce processus quitte son mode courant d'exécution (le mode esclave) pour passer en mode d'exécution du système, soit le mode superviseur. Ce passage du mode utilisateur au mode superviseur constitue une **commutation de contexte** : elle s'accompagne d'une opération de sauvegarde du contexte utilisateur, c'est-à-dire principalement de la valeur des registres du processeur (compteur ordinal, registre d'état), sur la pile noyau. Un contexte noyau est chargé

constitué d'une valeur de compteur ordinal correspondant à l'adresse de la fonction à exécuter dans le noyau, et d'un registre d'état en mode superviseur. Lorsque l'exécution de la **fonction système** (on parle aussi de **primitive**) est achevée, le processus repasse du mode superviseur au mode utilisateur. Il y a de nouveau une opération de commutation de contexte avec restauration du contexte utilisateur sauvegardé lors de l'appel système, ce qui permet de reprendre l'exécution du programme utilisateur juste après l'appel.

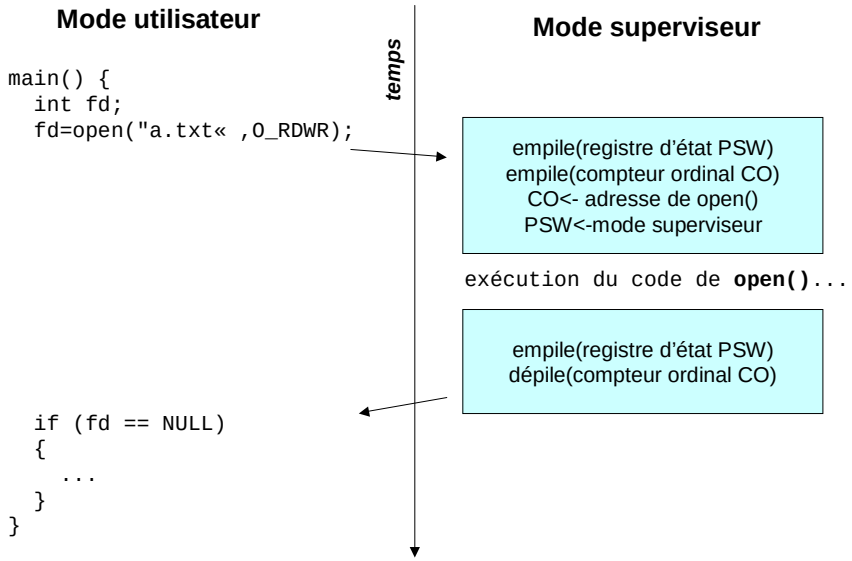
Trois causes majeures provoquent le passage du mode utilisateur au mode superviseur :

- le fait que le processus utilisateur appelle une fonction du système. C'est une demande explicite de passage en mode superviseur.

  Voici le mécanisme d'appel d'une fonction système sur un PC i386 : dans le registre %EAX, mettre le numéro de la fonction à appeler (en effet, chaque fonction système Linux possède un identifiant numérique unique), et mettre les arguments à transmettre dans les registres %EBX, %ECX, %EDX, %ESI et %ED. Appeler ensuite l'interruption 0x80. Exemple de fonctions systèmes :

%EAX	Nom	Code source	%EBX	%ECX	%EDX	%ESI	%EDI
1	sys_exit	Kernel/exit.c	int				
2	sys_fork	arch/i386/kernel/process.c	strucl pt_regs				
3	sys_read	fs/read_write.c	unsigned int	char *	size_t		
4	sys_write	fs/read_write.c	unsigned int	char *	size_t		
5	sys_open	fs/open.c	const char *	int	int		
6	sys_close	fs/open.c	unsigned int				

Nous reviendrons sur ce point dans un chapitre spécifique.



Exemple de commutation de contexte demandée par l'utilisateur

- l'exécution par le processus utilisateur d'une opération illicite (division par zéro, instruction machine interdite, violation mémoire...) : c'est la **trappe** ou **l'exception**. L'exécution du processus utilisateur est alors arrêtée ;
- la prise en compte d'une interruption par le matériel (IRQ). Le processus utilisateur est alors stoppé et l'exécution de la routine d'interruption associée à l'interruption survenue est exécutée en mode superviseur. L'interruption en question peut provenir de l'horloge du système. C'est par ce mécanisme que se gère le « temps de calcul » d'un processus.

Par ailleurs, il faut noter que le noyau Linux est un noyau **réentrant** : le traitement d'une interruption en mode noyau peut être interrompu au profit du traitement d'une autre interruption survenue entre-temps. De ce fait, les exécutions au sein du noyau peuvent être imbriquées les unes par rapport aux autres. Aussi, le noyau maintient-il une valeur de niveau d'imbrication, qui indique la profondeur d'imbrication courante dans les chemins de contrôle du noyau.

6.5.4 Principe de gestion des interruptions matérielles et logicielles

Dans le système Linux, chaque interruption, qu'elle soit matérielle ou logicielle, est identifiée par un entier de 8 bits appelé « **vecteur d'interruption** », dont la valeur varie de 0 à 255 :

- les valeurs de 0 à 31 correspondent aux interruptions « *non masquables* » (nous reviendrons sur ce terme) et aux exceptions. Correspond aux différentes interruptions gérées nativement par le CPU ;
- les valeurs de 32 à 47 sont affectées aux interruptions « *masquables* » levées par les périphériques (IRQ) ;
- les valeurs de 48 à 255 peuvent être utilisées pour identifier d'autres types de trappes que celles admises par le processeur (qui correspondent aux valeurs de 0 à 31). Comme nous l'avons déjà vu, l'entrée 128 (0x80 en hexadécimal) est réservée aux appels système.

Ce numéro permet d'adresser une table comportant 256 entrées, appelée « *table des vecteurs d'interruptions* » (`idt_table`), placée en mémoire centrale lors du démarrage de l'OS.

Ainsi, l'unité de contrôle du processeur, avant de commencer l'exécution nouvelle instruction machine, vérifie si une interruption ne lui a pas été délivrée. Si tel est le cas, les étapes suivantes sont mises en œuvre :

- l'interruption i a été délivrée au processeur. L'unité de contrôle accède à l'entrée n° *i* de la table des vecteurs d'interruptions, dont l'adresse est conservée dans un registre du processeur, et récupère l'adresse en mémoire centrale du gestionnaire de l'interruption levée ;
- l'unité de contrôle vérifie que l'interruption a été émise par une source autorisée ;
- l'unité de contrôle effectue un changement de niveau d'exécution (passage en mode superviseur) si cela est nécessaire et commute de pile d'exécution. En effet, comme nous l'avons déjà vu, les exécutions des gestionnaires d'interruptions pouvant être imbriquées, la prise en compte d'une interruption par l'unité de contrôle peut être faite alors que le mode d'exécution du processeur est déjà le mode superviseur (niveau d'imbrication > 1) ;
- l'unité de contrôle sauvegarde dans la pile noyau le contenu du registre d'état et du compteur ordinal;
- l'unité de contrôle charge le compteur ordinal avec l'adresse du **gestionnaire d'interruption**.

Le gestionnaire d'interruption s'exécute. Cette exécution achevée, l'unité de contrôle restaure le contexte sauvegardé au moment de la prise en compte de l'interruption, c'est-à-dire :

- l'unité de contrôle restaure les registres d'état et le compteur ordinal à partir des valeurs sauvegardées dans la pile noyau ;
- l'unité de contrôle commute de pile d'exécution pour revenir à la pile utilisateur si le niveau d'imbrication des exécutions du noyau est égal à 1.

Nous allons à présent examiner un peu plus en détail le fonctionnement des différents gestionnaires, pour les interruptions matérielles, pour les exceptions, et enfin, pour les appels systèmes.

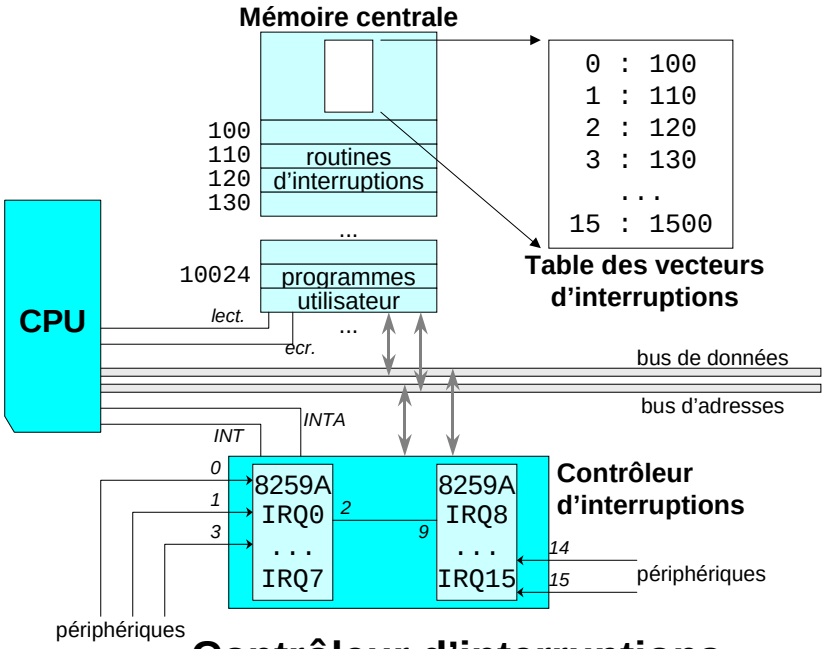
6.5.5 Prise en compte d'une interruption matérielle

Rappelons qu'une interruption matérielle est un signal permettant à un dispositif externe au CPU (un périphérique d'entrées/sorties par exemple) d'interrompre le travail courant du processeur afin qu'il aille réaliser un traitement particulier lié à la cause de l'interruption, appelé routine d'interruption (Cf. paragraphe sur les IRQ).

Comme il n'existe que 15 niveaux d'interruptions, il peut arriver que plusieurs périphériques différents soient attaches à la même interruption (partage de numéro d'IRQ). Dans ce cas, le contrôleur d'interruptions scrute chaque périphérique de la ligne d'interruption levée pour connaître le périphérique initiateur du signal.

Voici le déroulement des opérations lorsqu'une interruption matérielle est *levée* :

- le contrôleur d'interruption émet un signal sur la broche INT du CPU ;
- le processeur acquitte ce signal en envoyant un signal INTA ;
- le contrôleur place le numéro de l'interruption levée sur le bus de données ;
- le processeur (unité de contrôle) utilise ce numéro pour indexer la table des vecteurs d'interruptions et lancer l'exécution du gestionnaire de l'interruption matérielle selon la logique décrite précédemment.



Contrôleur d'interruptions

Une interruption survenant à n'importe quel moment, il peut être souhaitable d'interdire sa prise en compte par le processeur, notamment lorsque celui-ci train d'exécuter du code noyau critique. Pour cela, les interruptions peuvent être masquées grâce à une instruction machine spécifique (instruction « `cli` »). Dans ce cas, les interruptions sont ignorées par le processeur jusqu'à ce que celui-ci démasque les interruptions par le biais d'une seconde instruction machine (instruction « `sti` »).

Lorsqu'une IRQ i est levée au niveau matériel, c'est la fonction `IRQ`*`n`*`_interrupt()` qui est appelée, avec n le point d'entrée dans la table des vecteurs d'interruptions (n = i + 32, vu que les l'interruption matérielle sont numérotées à partir de 32 dans la table des vecteurs d'interruption).

Cette fonction commence par effectuer une sauvegarde complémentaire des registres généraux sur la pile noyau. A l'issue de cette sauvegarde, le gestionnaire de l'interruption invoque le traitement même de t'interruption (fonction `do_IRQ( )`). Comme l'interruption, si elle est partagée, peut avoir été émise par plusieurs périphériques différents, la routine d'interruption (appelée parfois « traitant d'interruption ») `do_IRQ()` contient plusieurs routines de services (appelées ISR : *interrupt service routines*), spécifiques à chacun des périphériques concernés. Ces routines sont exécutées les unes à la suite des autres de façon à déterminer quel est le périphérique initiateur de l'interruption. L'ISR concernée est alors totalement exécutée.

Les actions exécutées par une routine de service sont réparties selon 3 catégories:

- les actions critiques qui doivent être exécutées immédiatement, interruptions masquées;
- les actions non critiques qui peuvent être exécutées rapidement, interruptions démasquées;
- les actions pouvant être reportées. Ce sont des actions dont la durée d'exécution est longue et dont la réalisation n'est pas critique pour le fonctionnement du noyau. Pour cette raison, le système Linux choisit de reporter leur exécution en dehors de la routine de service, dans des fonctions appelées les « *parties basses* » (*bottom halves*) qui seront exécutées plus tard, juste avant de revenir en mode utilisateur. L'ISR se contente d'activer ces parties basses pour demander leur exécution ultérieure (fonction `mark_bh()`).

Une fois les routines de services exécutées, le contexte sauvegardé au moment de la prise en compte de l'interruption est restitué, d'une part par le gestionnaire de l'interruption matérielle (restitution des registres généraux sauvegardés lors de son appel), d'autre part par l'unité de contrôle du processeur (restitution du registre d'état et du compteur ordinal). Si le niveau d'imbrication du noyau est égal à 1, alors le mode d'exécution bascule au niveau utilisateur et le programme utilisateur reprend son exécution. Avant ce retour en mode utilisateur, les parties basses activées par les routines de services sont exécutées.

6.5.6 Gestion des exceptions (trappes)

L'occurrence d'une trappe est levée lorsque le CPU rencontre une situation anormale en cours d'exécution, comme, par exemple, une division par 0 (trappe 0 sous Linux), un débordement (trappe 4 sous Linux), l'utilisation d'une instruction interdite (trappe 6 sous Linux) ou encore, un accès mémoire interdit (trappe 10 sous Linux).

Le traitement d'une trappe est très comparable à celui des interruptions, évoqué au paragraphe précédent. La différence essentielle réside dans le fait que l'occurrence d'une trappe est synchrone avec l'exécution du programme.

Aussi, de la même manière que pour les interruptions, une trappe est caractérisée par un numéro et un gestionnaire de trappe lui est associé, dont l'adresse en mémoire est rangée dans la table des vecteurs d'interruptions.

Le traitement associé à la trappe consiste à envoyer un signal au processus fautif (fonction `do_handler_name`() où « *handler_name* » est le nom de l'exception). La prise en compte de ce signal provoque par défaut l'arrêt du programme en cours d'exécution, mais le processus a la possibilité de spécifier un traitement particulier qui remplace alors le traitement par défaut (nous verrons ce point ultérieurement lorsque nous parlerons des signaux). Dans le cas par défaut, une image mémoire de l'exécution arrêtée est créée afin de pouvoir être utilisée par le programmeur, pour comprendre le problème survenu (cette image mémoire du programme fautif est stocké dans un fichier `coredump`).

Une trappe particulière est celle liée à la gestion des défauts de pages en mémoire centrale (trappe 14 sous Linux). Son occurrence n'entraîne pas la terminaison du processus associé, mais permet le chargement à la demande des pages constituant l'espace d'adressage du processus (nous y reviendrons dans le chapitre traitant de la mémoire).

6.5.7 Exécution d'une fonction du système

L'appel à une fonction du système peut être réalisé soit depuis un programme utilisateur par le biais d'un appel de procédure qualifié dans ce cas **d'appel système**, soit par le biais d'une commande (lancée depuis le *prompt shell*). La commande est prise en compte par un programme particulier, l'interpréteur de commandes, qui à son tour invoque la fonction système correspondante.

L'invocation d'une fonction système se traduit par la levée d'une interruption logicielle particulière, entraînant un changement de mode d'exécution et le branchement à l'adresse de la fonction.

Plus précisément, l'ensemble des appels système disponibles est regroupé dans une ou plusieurs bibliothèques, avec lesquelles l'éditeur de liens fait la liaison. Chaque fonction de la bibliothèque encore appelée routine d'enveloppe comprend une instruction permettant le changement de mode d'exécution, ainsi que des instructions assurant le passage des paramètres depuis le mode utilisateur vers le mode superviseur. Souvent, des registres généraux prédéfinis du processeur servent à ce transfert de paramètres. Enfin, la fonction lève une trappe en passant au système un numéro spécifique à la routine appelée.

Comme nous l'avons déjà vu, chaque fonction système est identifiée par un numéro.

Le système cherche dans une table l'adresse de la routine identifiée par le numéro de la fonction, et se branche sur son code. A la fin de l'exécution de la routine, la fonction de la bibliothèque retourne les résultats de l'exécution via les registres réservés à cet effet, puis restaure le contexte du programme utilisateur.

Sous le système Linux, la trappe levée par la routine d'enveloppe pour l'exécution d'une fonction du système porte le numéro 128 (0x80 en hexadécimal). Les paramètres, dont le nombre ne doit pas être supérieur à 5, sont passés via les registres %CBX, %CCX, %CDX, %ESI, et %CDI du processeur x86. Le numéro de la routine système concernée est passé via le registre %EAX (l'accumulateur). Ce numéro sert d'index pour le gestionnaire des appels système dans une table des appels système, la table `sys_call_table`, comportant 256 entrées. Chaque entrée de la table contient l'adresse de la routine système correspondant à l'appel système demandé. Le gestionnaire d'appel système renvoie via le registre %EAX un code retour qui coïncide à une situation d'erreur si sa valeur est comprise entre -1 et -125. Dans ce cas, cette valeur est placée dans une variable globale `errno` et le registre %EAX est rempli avec la valeur -1, ce qui indique pour le processus utilisateur une situation d'échec dans la réalisation de l'appel système. La cause de l'erreur peut être alors consultée dans la variable errno à l'aide d'un ensemble de fonctions et variables prédéfinies (Cf. man) :

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
void perror(const char *s);
char *strerror(int errnum);
```

La fonction `perror()` affiche un message constitué de deux parties, d'une part, la chaîne spécifiée par le paramètre `s`, d'autre part, le message d'erreur correspondant à la valeur de la variable `errno`.

La fonction `strerror()` retourne le message d'erreur correspondant à la valeur de l'erreur spécifiée par l'argument `errnum`.

L'ensemble des valeurs admises pour la variable `errno` est spécifié dans le fichier « `errno.h` ».

6.5.8 Imbrication de la prise en compte des interruptions

Comme déjà évoqué, le noyau Linux est un noyau réentrant, c'est à dire qu'à un instant t donné, plusieurs exécutions en mode noyau peuvent exister, une seule étant active. En effet, le noyau autorise un gestionnaire d'interruption à interrompre l'exécution d'un autre gestionnaire d'interruption. L'exécution du gestionnaire est suspendue, son contexte sauvegardé sur la pile noyau, et l'exécution est reprise lorsque l'exécution du gestionnaire survenue entre-temps est achevée. De ce fait, les exécutions au sein du noyau peuvent être arbitrairement imbriquées.

Le noyau maintient une valeur de niveau d'imbrication, qui indique la profondeur d'imbrication courante dans les exécutions du noyau. Le retour en mode utilisateur n'est effectué que lorsque ce niveau d'imbrication est à 1. Il est précédé de l'exécution des parties basses activées par l'ensemble des routines de services exécutées. Plus précisément :

- un gestionnaire d'interruption peut interrompre un gestionnaire de trappe ou un autre gestionnaire d'interruption ;
- par contre, un gestionnaire de trappe ne peut jamais interrompre un gestionnaire d'interruption.

7 Le « multitâche » en pratique

7.1 Les processus

Nous avons survolé dans les précédents chapitres les mécanismes d'interruption et de commutation de contexte, qui sont les outils qui permettent la mise en place de « multitâche » (temps partagé, ou **pseudo-parallélisme**) sur un système mono-processeur.

La notion de « tâche » étant restée floue, nous allons essayer de la préciser. Historiquement, sous Unix (puis sous Linux), une tâche s'est appelée « **processus** ».

Définition : un **processus** est en quelque un programme (du code) qui s'exécute en mémoire. Il est protégé par un certains nombre de mécanismes système (que nous verrons dans ce cours), afin qu'aucun autre processus ne puisse venir directement le perturber (par exemple, un processus ne peut accéder aux données, au code, à la pile à l'exécution, etc. d'autres processus).

Voici, sous Linux, les éléments qui sont propre à chaque processus :

Gestion des tâches	Gestion de la mémoire	Gestion de fichiers
Etat des registres du CPU	Pointeur vers un segment texte (constantes, données d'initialisation des variables, etc.)	Répertoire racine (*root*)
Compteur ordinal	Pointeur vers un segment de données (où sont stockées les variables)	Répertoire de travail (*current directory*)
Etat du processus	Pointeur vers un segment de la pile	Descripteurs de fichiers ouverts
Pointeur de la pile		ID utilisateur
Etat du processus		ID groupe
Priorité		
Paramètres d'ordonnancement		
Identifiant du processus (PID)		
PID du processus parent		
Groupe du processus		
Etat des signaux		
Heure de lancement du processus		
Temps de CPU utilisé		
Temps de traitement du fils		
Heure de la prochaine alerte		

Les plus curieux (et les plus courageux, la lecture n'étant pas aisée de prime abord) pourront regarder le type « `struct task_struct` » défini sous Linux dans le fichier « `/usr/include/linux/sched.h` » pour stocker ces données.

Ce qu'il faut retenir de ce tableau : un processus n'est pas simplement du code et un lieu de stockage des données, mais il contient aussi un grand nombre d'informations utiles au système d'exploitation et à lui-même pour gérer la sécurité, la quantité de mémoire allouée, les entrées/sorties, etc.

Dans un environnement multitâche, un processus est caractérisé par son état. En effet, il peut-être :

- en cours d'éxécution (état « ELU »),
- en attende de temps CPU pour être exécuté à nouveau (état « PRET »),
- enfin, le processus peut être en attente d'une ressource - mémoire, fichier, etc. – (état « BLOQUE ».

Le diagramme des états d'un processus peut se matérialiser par le diagramme d'états suivant :

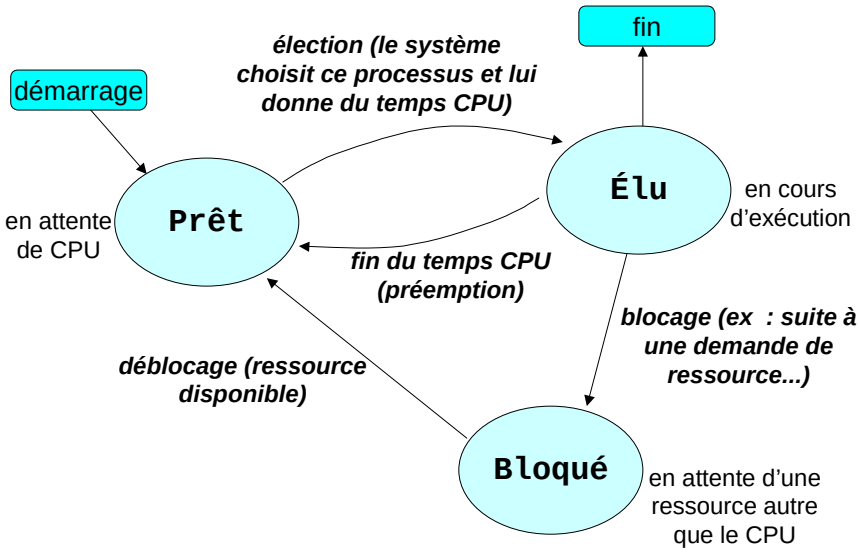


Diagramme d'état d'un processus Unix/Linux

Lors du démarrage, un mécanisme lit le contenu du fichier exécutable. Il regarde les deux premiers octets du fichier (qui forment un nombre appelé « *Magic Number* »). S'il détecte qu'il ne s'agit pas d'un script (script shell, perl, etc.) mais d'un fichier exécutable en binaire, il place le contenu du code et des données d'initialisation en mémoire, alloue de la mémoire pour les données, la pile, etc. Ce mécanisme s'appelle le « **chargeur** ».

Tout système d'exploitation multitâche définit une structure pour gérer les processus : il s'agit du **PCB** (*Process Control Block*, i.e. bloc de contrôle de processus). Le PCB contient un identificateur unique pour chaque processus (**PID** pour *Process ID*), l'état courant du processus (élu, prêt, bloqué), les informations liées à l'ordonnancement du processus, etc. Le PCB sous Linux correspond à la structure « `struct task_struct` » déjà évoquée ci-dessus. L'ensemble de tous les blocs de contrôle sont stockés dans une **table des processus**, nommée **PT** (*process table*) dans la littérature anglaise.

Dans le noyau Linux, cette table était définie dans le fichier « `kernel/sched.c` »), et s'appelle « `task[NRTASK]` ». Chaque élément de ce tableau est un pointeur qui pointe vers un BCB de type « `struct task_struct` ».

Chaque processus, identifié par son **PID**, appartient à un groupe de processus. Un groupe de processus est identifié par un identifiant : le **GID** (*Group ID*). En pratique, deux processus sont dans le même groupe de processus s'ils ont été lancés par le même processus parent. Le numéro du processus parent est noté **PPID** (*Parent PID*).

7.2 Création d'un processus sous Linux

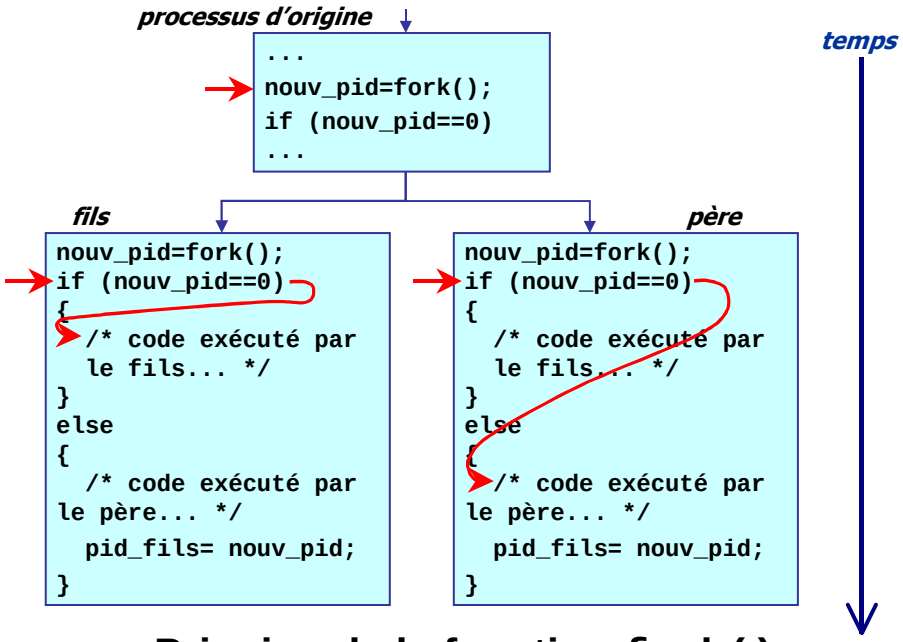
7.2.1 La fonction `fork()`

La création d'un processus sous Linux est faite avec la fonction « `fork()` » :

```
#include <unistd.h>
pid_t  fork(void);
```

L'appel de la fonction système « `fork()` » crée un nouveau processus, qui sera l'exacte copie du processus appelant. Le processus appelant est appelé le **processus parent**. Le processus nouvellement créé est appelé le **processus fils**. La fonction `fork()` retourne au père le numéro de PID du fils qui vient d'être créé, et retourne la valeur 0 au fils.

Ce mécanisme peut se schématiser le la façon suivante :



Principe de la fonction `fork()`

Lors de l'exécution de la fonction `fork()`, si les ressources noyaux sont disponibles, le système effectue les opérations suivantes :

- le système alloue un bloc de contrôle (**PCB**) et une pile noyau pour le nouveau processus, et trouve une entrée libre dans la table des processus pour ce nouveau bloc de contrôle ;
- le système copie les informations contenues dans le bloc de contrôle du père dans celui du fils sauf les informations concernant le PID, le PPID et les chaînages vers les fils et les frères ;
- le système alloue un **PID** unique au nouveau processus ;
- le système associe au processus fils le contexte mémoire du processus parent (c'est-à-dire le code, les données, et la pile) ;
- l'état du nouveau processus est mis à la valeur « `TASK_RUNNING` » ;
- le système retourne au processus père le PID du nouveau processus ainsi créé, et au nouveau processus (le fils), la valeur 0.

Ainsi, lors de cette création, le processus fils hérite de tous les attributs de son père sauf :

- son propre identificateur,
- et tout les compteurs de temps d'exécution (qui sont remis à 0).

S'il n'y a plus assez de ressource pour créer un nouveau processus, la fonction `fork()` retourne une valeur négative. La variable globale `errno` donne la raison de cette erreur (`errno` vaut `EAGAIN` s'il n'y a plus assez de place dans la table des processus, et `ENOMEM` s'il n'y a plus assez de mémoire).

7.2.2 Les fonctions de gestion des identifiants de processus

Le système propose un certain nombre de fonctions pour gérer les numéros de processus :

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

pid_t getpid(void); /* retourne le PID du processus courant */
pid_t getppid(void);/* retourne le PID du processus parent */
uid_t getuid(void); /* retourne l'UID du processus */
gid_t getgid(void); /* retourne le GID du processus */
```

A noter que l'UID, ainsi que le GID sont égaux pour le père et pour le fils juste après l'appel à la fonction `fork()`.

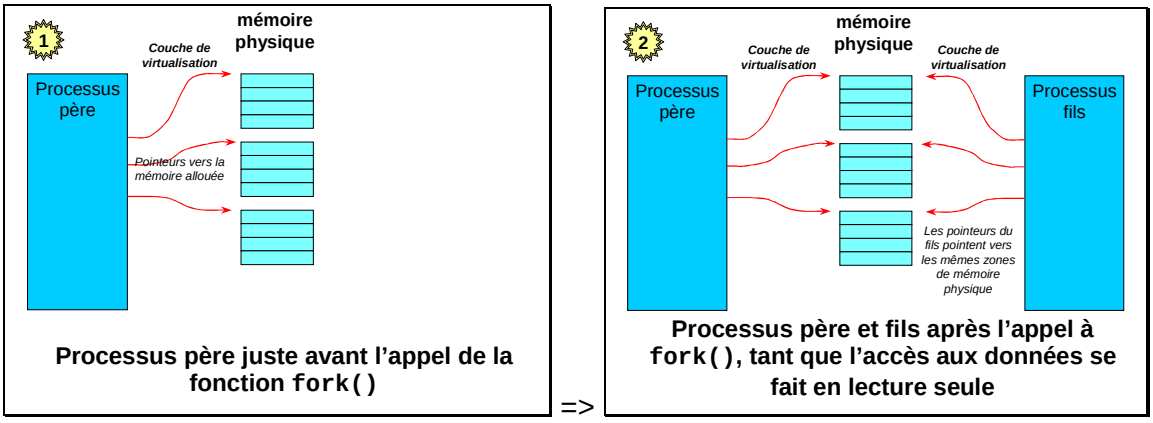
7.2.3 Optimisation de `fork()` : le « *copy on write* »

Comme nous venons de le voir, l'appel à la fonction `fork()` doit normalement engendrer :

- la duplication du code,
- la duplication de la pile à l'exécution,
- la duplication du « **tas** » (*heap*), c'est à dire des la zone mémoire composée du :
  - le segment texte (constantes, données d'initialisation des variables, etc.)
  - segment de données (où sont stockées les variables), etc.

S'il devait être exécuté entièrement, l'ensemble de ce travail serait très coûteux en terme de CPU (recopie de mémoire). Or, en pratique, il est assez fréquent que le processus père et le processus fils accèdent aux données en lecture seule, et que celles-ci ne sont pas modifiées avant le fin du processus fils. Dans ce cas, la recopie aura été totalement inutile : le père et le fils auraient pu partager les mêmes données.

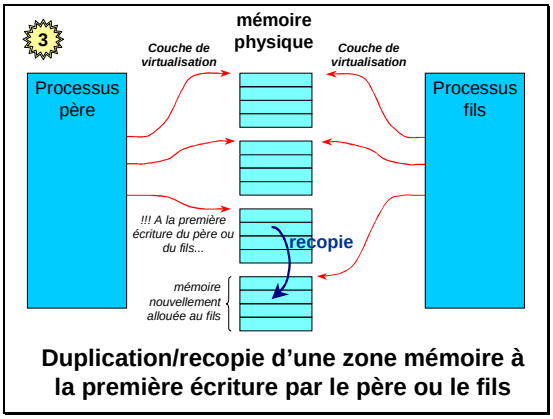
Aussi, en pratique, seule la pile à l'exécution est dupliquée lors d'un `fork()`. Le code et les autres segments de données ne sont pas dupliqués. Par un mécanisme de « *virtualisation* » de la mémoire physique (qui sera étudiée dans un prochain chapitre), la mémoire physique allouée au fils est exactement la même que celle allouée au père. Et c'est uniquement lors de la première modification (écriture) d'une valeur dans une zone mémoire que celle-ci est dupliquée (allocation mémoire puis recopie des données). Ce mécanisme s'appelle « ***copy on write*** » (recopie uniquement lors d'une écriture) :



Processus père juste avant l'appel de la fonction `fork()`

Processus père et fils après l'appel à `fork()`, tant que l'accès aux données se fait en lecture seule

=>



Duplication/recopie d'une zone mémoire à la première écriture par le père ou le fils

7.2.4 Les processus zombies (et comment les éviter)

Il existe une erreur classique dans le développement de programme Unix/Linux gérant plusieurs processus : un processus père qui crée des fils, et qui ne s'occupe pas ensuite d'acquérir leur code de fin (c'est à dire la valeur retournée par la fonction `exit()`, ou retournée par le `return()` de la fonction `main()`).

Ces processus fils restent dans un état « *fin d'exécution du programme* », jusqu'à ce que le père s'inquiète du code retour du processus fils. On parle alors de processus « ***zombie*** ».

A noter que les processus zombies ne peuvent pas être supprimés par les méthodes classiques (y compris pour les utilisateurs privilégiés). Les ressources (mémoire, fichiers, etc.) son libérées, mais le processus prend encore une place dans la table des processus. Comme cette dernière n'est pas extensible, le système se retrouve alors encombré de processus inactifs (ils ont comme état la valeur « `TASK_ZOMBIE` »).

Normalement, les processus zombies sont « purgés » à la mort du processus père. Si cette purge se passe mal, seul un redémarrage système sera efficace.

Plusieurs méthodes permettent de les éviter (on parle de **synchronisation père/fils**) :

- Utilisation des fonctions `wait()` et `waitpid()` :

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

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

extrait de « `man wait` » ou de « `man waitpid` » : <<

```
La fonction wait() suspend l'exécution du processus
courant jusqu'à ce qu'un enfant se termine, ou jusqu'à
ce qu'un signal à intercepter arrive. Si un processus
fils s'est déjà terminé au moment de l'appel (il est
devenu "zombie"), la fonction revient immédiatement.
Toutes les ressources utilisées par le fils sont
libérées.

La fonction waitpid() suspend l'exécution du processus
courant jusqu'à ce que le processus fils numéro pid se
termine, ou jusqu'à ce qu'un signal à intercepter
arrive. Si le fils mentionné par pid s'est déjà terminé
au moment de l'appel (il est devenu "zombie"), la
fonction revient immédiatement. Toutes les ressources
utilisées par le fils sont libérées.
```

La valeur de ***pid*** peut également être l'une des suivantes :

- **< -1** : attendre la fin de n'importe quel processus fils appartenant à un groupe de processus d'ID pid.
- **-1** : attendre la fin de n'importe quel fils. C'est le même comportement que wait.
- **0** : attendre la fin de n'importe quel processus fils du même groupe que l'appelant.
- **> 0** : attendre la fin du processus numéro ***pid***.

La valeur de l'argument ***option*** options est un OU binaire entre les constantes suivantes :

- **WNOHANG** : ne pas bloquer si aucun fils ne s'est terminé.
- **WUNTRACED** : recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue.

Si ***status*** est non NULL, wait() et waitpid() y stockent l'information sur la terminaison du fils.

Cette information peut être analysée avec les macros suivantes, qui réclament en argument le buffer status (un int, et non pas un pointeur sur ce buffer) :

- **WIFEXITED(status)** : non nul si le fils s'est terminé normalement
- **WEXITSTATUS(status)** : donne le code de retour tel qu'il a été mentionné dans l'appel exit() ou dans le return de la routine main. Cette macro ne peut être évaluée que si WIFEXITED est non nul.
- **WIFSIGNALED(status)** : indique que le fils s'est terminé à cause d'un signal non intercepté.
- **WTERMSIG(status)** : donne le nombre de signaux qui ont causé la fin du fils. Cette macro ne peut être évaluée que si WIFSIGNALED est non nul.
- **WIFSTOPPED(status)** : indique que le fils est actuellement arrêté. Cette macro n'a de sens que si l'on a effectué l'appel avec l'option WUNTRACED.
- **WSTOPSIG(status)** : donne le nombre de signaux qui ont causé l'arrêt du fils. Cette macro ne peut être évaluée que si WIFSTOPPED est non nul.

Valeur Renvoyée : en cas de réussite, le PID du fils qui s'est terminé est renvoyé, en cas d'echec -1 est renvoyé et ***errno*** contient le code d'erreur :

- **ECHILD** : Le processus indiqué par pid n'existe pas, ou n'est pas un fils du processus

```
                    appelant (Ceci peut arriver pour son
                    propre fils si l'action de SIGCHLD est
                    placé sur SIG_IGN, voir également le
                    passage de la section NOTES concernant
                    les threads).
       EINVAL       : L'argument options est invalide.
       ERESTARTSYS  : WNOHANG n'est pas indiqué, et un
                    signal à intercepter ou SIGCHLD a été
                    reçu. Cette erreur est renvoyée par
                    l'appel système. La routine de
                    bibliothèque d'interface n'est pas
                    autorisée à renvoyer ERESTARTSYS, mais
                    renverra EINTR.
 >>
```

- Le problème de la fonction `wait()`, et de la fonction `waitpid()` dans son fonctionnement par défaut est qu'elles sont bloquantes (suite à leur appel, le programme s'arrête jusqu'à ce qu'un fils se termine). Une première solution à utiliser `waitpid()` en mettant le paramètre « ***option*** » à « `WNOHANG` ». Mais comme nous ne savons pas à quel moment un fils se termine, il faut lancer régulièrement cette fonction `waitpid()`. On dit alors que nous faisons du « *polling* ». Ce mode de fonctionnement, qui consomme du CPU et des appels systèmes n'est pas optimum. Une solution bien plus esthétique consiste à n'appeler `waitpid()`que lorsque nous sommes sûr qu'un processus fils est terminé. Nous y reviendrons très largement dans le chapitre prévu à cet effet, mais vous pouvez noter dès à présent que le père reçoit un « *signal* » du système d'exploitation lorsqu'un de ses fils se termine. Ce signal peut être « *intercepté* » à l'aide de la fonction `signal()`. Voici un extrait de la documentation de cette fonction :

```
#include <signal.h>

void (*signal(int signum, void (*handler)(int)))(int);

L'appel-système signal() installe un nouveau
gestionnaire pour le signal numéro signum. Le
gestionnaire de signal est handler (NDLR : c'est à dire le pointeur
vers une fonction) qui peut être soit une fonction spécifique
de l'utilisateur, soit l'une des constantes SIG_IGN ou
SIG_DFL.

Lors de l'arrivée d'un signal correspondant au numéro
signum, les évènements suivants se produisent :
  - si le gestionnaire correspondant est configuré avec
    SIG_IGN, le signal est ignoré.
  - si le gestionnaire vaut SIG_DFL, l'action par défaut
    pour le signal est entreprise, comme décrit dans
    le manuel signal(7).
  - finalement, si le gestionnaire est dirigé vers une
    fonction handler(), alors tout d'abord, le
    gestionnaire est re-configuré à SIG_DFL, ou le
    signal est bloqué, puis handler() est appelé avec
    l'argument signum.

Utiliser une fonction comme gestionnaire de signal est
appelé "intercepter - ou capturer - le signal". Les
signaux SIGKILL et SIGSTOP ne peuvent ni être ignoré, ni
être interceptés.
```

```
Valeur Renvoyée : signal() renvoie la valeur précédente
du gestionnaire de signaux, ou SIG_ERR en cas d'erreur.
>>
```

En pratique, lorsqu'un fils se termine, le père reçoit un signal de type « SIGCHLD ». L'idée consiste à intercepter ce signal « SIGCHLD » lorsque le fils se termine, et seulement à ce moment là, lancer une des fonctions bloquante waitpid() en mode non bloquant. Voici un squelette d'un tel programme :

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

[...]
void fin_d_un_enfant(int num_signal)
{
    /* On ne l'utilise pas, mais notez qu'à ce */
    /* stade, le paramètre num_signal vaut */
    /* obligatoirement la valeur SIGCHLD (vu que */
    /* cette fonction ne sert qu'à intercepter */
    /* le signal SIGCHLD). */

    /* Nettoyage des processus fils */
    /* Comme plusieurs fils peuvent se terminer au */
    /* même instant, nous ne savons pas si la */
    /* fonction est appelée suite à la fin d'un ou de */
    /* plusieurs fils. Il faut lancer waitpid() */
    /* plusieurs fois (à l'aide d'une boucle while), */
    /* tant qu'il reste des zombies à faire */
    /* disparaître : */
    while(waitpid(-1, NULL, WNOHANG) > 0);
    /* on repositionne la présente fonction */
    /* comme vecteur d'interception (handler) */
    /* pour le prochain fils qui viendra à se */
    /* terminer */
    signal(SIGCHLD, fin_d_un_enfant);
}
[...]
int main(int argc, char *argv[])
{
    ...
    signal(SIGCHLD, fin_d_un_enfant);
    ...
    if (fork() == 0)
    {
        /* Code du fils... */
        /* Pour le fils, on n'a pas à intercepter */
        /* le signe SIGCHLD => on positionne le  */
        /* handler par défaut. */
        signal(SIGCHLD, SIG_DFL);
        [...] /* suite du code du fils */
    }
    else
    {
        /* Le code du père... */
        [...]
    }
}
```

7.2.5 Les fonctions de recouvrement

Lorsque nous appelons la fonction `fork()`, le code du processus fils est exactement le même que celui du processus parent. Or, nous pouvons être amenés à avoir à exécuter un autre programme dans le processus fils.

Un ensemble de fonctions, dites « *fonctions de recouvrement* » (ou *primitives de recouvrement*), permettent de charger un autre programme à la place de celui en cours. Voici la liste de ces fonctions sous Linux :

- `execl()`,
- `execlp()`,
- `execle()`,
- `execv()`,
- `execvp()`,
- `execve()`.

Le manuel Linux nous permet de comprendre le fonctionnement de ces fonctions :

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

      extern char **environ;

      int execl(const char *path, const char *arg, ...);
      int execlp(const char *file, const char *arg, ...);
      int execle(const char *path, const char *arg , ..., \
                char * const envp[]);
      int execv(const char *path, char *const argv[]);
      int execvp(const char *file, char *const argv[]);

      Description : la famille de fonctions exec remplace l'image
      mémoire du processus en cours par un nouveau processus. Les
      fonctions décrites dans cette page sont en réalité des
      frontaux pour l'appel système execve(2) . (Voir la page de
      execve pour des informations détaillées sur le remplacement du
      processus en cours).

      L'argument initial de toutes ces fonctions est le chemin
      d'accès du fichier à exécuter.

      Les arguments const char *arg ainsi que les points de
      suspension des fonctions execl, execlp, et execle peuvent être
      vues comme des arg0, arg1, ..., argn. Ensemble ils décrivent
      une liste d'un ou plusieurs pointeurs sur des chaînes de
      caractères terminées par des caractères nuls, qui constituent
      les arguments disponibles pour le programme à exécuter. Par
      convention, le premier argument doit pointer sur le nom du
      fichier associé au programme à exécuter. La liste des
      arguments doit se terminer par un pointeur NULL.

      Les fonctions execv et execvp utilisent un tableau de
      pointeurs sur des chaînes de caractères terminées par des
      caractères nuls, qui constituent les arguments disponibles
      pour le programme à exécuter. Par convention le premier
      argument doit pointer sur le nom du fichier associé au
```

programme à exécuter. Le tableau de pointeur doit se terminer par un pointeur NULL.

La fonction execle peut également indiquer l'environnement du processus à exécuter en faisant suivre le pointeur NULL qui termine la liste d'arguments, ou le pointeur NULL de la table par un paramètre supplémentaire. Ce paramètre est un tableau de pointeurs sur des chaînes de caractères terminées par des caractères nuls, qui doit se terminer par un pointeur NULL. Les autres fonctions fournissent au nouveau processus l'environnement constitué par la variable externe environ.

Certaines de ces fonctions ont une sémantique spécifique.

Les fonctions execlp et execvp agiront comme le shell dans la recherche du fichier exécutable si le nom fourni ne contient pas de slash (/). Le chemin de recherche est spécifié dans la variable d'environnement PATH. Si cette variable n'est pas définie, le chemin par défaut sera ``/bin:/usr/bin:''. De plus certaines erreurs sont traitées de manière spécifique.

Si la permission d'accès au fichier est refusée (execve renvoie EACCES), ces fonctions continueront à parcourir le reste du chemin de recherche. Si aucun fichier n'est trouvé, elle reviendront, et errno contiendra le code d'erreur EACCES.

Si l'en-tête d'un fichier n'est pas reconnu (execve renvoie **ENOEXEC**), ces fonctions exécuteront un shell avec le chemin d'accès au fichier en tant que premier argument. Si ceci échoue, aucune recherche supplémentaire n'est effectuée.

Valeur Renvoyée : si l'une quelconque des fonctions exec revient à l'appelant, c'est qu'une erreur a eu lieu. La valeur de retour est -1, et errno contient le code d'erreur.

Erreurs : toutes ces fonctions peuvent échouer et positionner ***errno*** sur n'importe quelle erreur décrite dans la fonction execve(2).

>>

Extrais du manuel de execve() :

<<

```
#include <unistd.h>

int execve(const char *fichier, char *constargv[], \
           char *const envp[]);
```

Description : execve() exécute le programme correspondant au ***fichier***. Celui-ci doit être un exécutable binaire ou bien un script commençant par une ligne du type "#! interpreteur [arg]". Dans ce dernier cas, l'interpréteur doit être indiqué par un nom complet, avec son chemin d'accès, et qui sera invoqué sous la forme interprèteur [arg] fichier.

argv est un tableau de chaînes d'arguments passées au nouveau programme. ***envp*** est un tableau de chaînes, ayant par

convention la forme *cle=valeur*, qui sont passées au nouveau programme comme environnement. ***argv*** ainsi que ***envp*** doivent se terminer par un pointeur NULL. Les arguments et l'environnement sont accessibles par le nouveau programme dans sa fonction principale, lorsqu'elle est définie comme :
  int main (int ***argc***, char * ***argv*** [], char * ***envp*** []).

En cas de réussite, execve() ne revient pas à l'appelant, et les segments de texte, de données, ainsi que la pile du processus appelant sont remplacés par ceux du programme chargé. Le programme invoqué hérite du PID du processus appelant, et de tous les descripteurs de fichiers qui ne possèdent pas le drapeau **Close-on-Exec**. Les signaux en attente destinés au processus parent sont effacés. Les signaux prêts à être intercepté par le processus appelant reprennent leur comportement par défaut.
[...]

Si l'éxécutable est au format ELF lié dynamiquement, l'interprèteur indiqué dans le segment PT_INTERP sera invoqué pour charger les bibliothèques partagées. Cet interprèteur est généralement /lib/ld-linux.so.1 pour les fichiers binaires liés avec la libc Linux version 5, ou /lib/ld-linux.so.2 pour ceux liés avec la GNU libc version 2.

Valeur Renvoyée : en cas de réussite, execve() ne revient pas, en cas d'échec il renvoie -1 et ***errno*** contient le code d'erreur.

Erreurs :

EACCES : Le fichier n'est pas un fichier régulier.
EACCES : L'autorisation d'exécution est refusée pour le fichier, ou un script, ou un interpréteur ELF.
EPERM : Le système de fichiers est monté avec l'attribut noexec.
EPERM : Le système de fichiers est monté avec l'attribut nosuid et le fichier a un bit Set-UID ou Set-GID positionné.
E2BIG : La liste d'arguments est trop grande.
ENOEXEC : Le fichier éxécutable n'est pas dans le bon format, ou est destiné à une autre architecture.
EFAULT : L'argument fichier pointe en dehors de l'espace d'adressage accessible.
ENAMETOOLONG : La chaîne de caractères fichier est trop longue.
ENOENT : Le fichier n'existe pas.
ENOMEM : Pas assez de mémoire pour le noyau.
ENOTDIR : Un élément du chemin d'accès n'est pas un répertoire.
ELOOP : Le chemin d'accès au fichier contient une référence circulaire (à travers un lien symbolique)
ETXTBSY : Le fichier exécutable a été ouvert en écriture par un ou plusieurs processus.
EIO : Une erreur d'entrée/sortie de bas-niveau s'est produite.

```
ENFILE      : Le nombre maximal de fichiers ouverts sur le
              système est atteint
EMFILE      : Le nombre maximal de fichiers ouverts par
              processus est atteint.
EINVAL      : Un exécutable ELF a plusieurs segments
              PT_INTERP (indique plusieurs interprèteurs).
EISDIR      : L'interprèteur ELF cité est un répertoire.
ELIBBAD     : L'interprèteur ELF mentionné n'est pas dans un
              format connu.
>>
```

7.3 Les *threads* (ou *processus légers*)

7.3.1 Principe des *threads*

Nous avons vu que la création d'un processus était assez gourmand en ressource. Le mécanisme de « *copy on write* » allège ce mécanisme dans les cas où les processus fils ne modifie pas leur données.

Or, ce lourd mécanisme de création de processus à travers la fonction `fork()` n'a d'intérêt que si le programmeur souhaite isoler les deux tâches en cours d'exécution. Or, dans de nombreux cas, cet isolement du père et du fils n'a pas d'intérêt. Pire, dans les cas où le père et le fils ont à dialoguer, cet isolement oblige chacun des processus à mettre en œuvre de lourds mécanisme de communication (Cf. chapitre sur la « *shared memory* » ou mémoire partagée).

Aussi, le mécanisme de *thread* (ou processus léger) a été mis en œuvre. Son principe : créer plusieurs tâches (plusieurs fils d'exécution) au sein d'un même processus, chaque fil d'exécution partageant les mêmes ressources et le même espace d'adressage (ce qui signifie par que si un *thread* modifie la valeur d'une variable globale, la nouvelle valeur sera visible de tous les autres *threads*). Le tableau suivant indique ce qui est propre à chaque *thread*, et ce qui est commun à tout le processus (partagé entre tous les *threads*) :

Eléments communs au processus	Eléments spécifiques à chaque *thread*
Espace d'adressage	Compteur ordinal
Variables globales	Registres
Fichiers ouverts	Pile
Processus enfants	+/- : Etat (Cf. *thread* noyau vs. utilisateur)
Alertes et attentes	
Signaux et interception de signaux	
Informations de décompte (utilisation CPU)	

Voici un bref récapitulatif des avantages/inconvénients des threads vs. les processus :

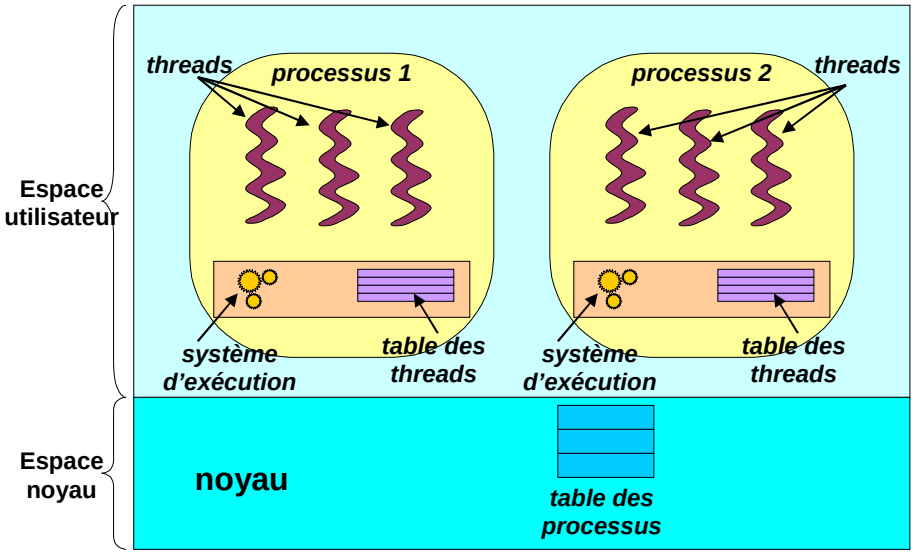
avantage *thread* vs. processus	inconvénient *thread* vs. processus
Avec les *thread*, allègement des opérations de commutations de contexte (il n'y à pas à basculer tout l'environnement du processus)	séparation des espaces d'adressage dans les processus ; le fils peut faire n'importe quoi (par exemple, changer de UID, ou modifier une variable globale) sans gêner le père.
allègement des opérations de création d'un nouveau fil dans le cas de création d'un *thread* (pas besoin de mettre en œuvre le **copy on write**)	Le partage de ressources (fichier par exemple) est aisé avec le processus, plus difficile à réaliser avec les *threads* (du fait que les fils d'exécution partagent les même espaces d'adressage).

Les thread peuvent être implémentés selon deux mécanismes :

- au niveau utilisateur,
- ou au niveau noyau.

7.3.2 Implémentation des *threads* au niveau utilisateur

Lorsque l'implémentation des processus légers est réalisée au niveau utilisateur, le noyau ignore ceux-ci et ordonnance des processus classiques composes d'un seul fil d'exécution. Une bibliothèque système gère l'interface avec le noyau en prenant en charge la gestion des *threads* et en cachant ceux-ci au noyau. Cet exécutif est donc responsable de commuter les *threads* au sein d'un même processus, lorsque ceux-ci en font explicitement la demande :



Gestion des *threads* au niveau utilisateur

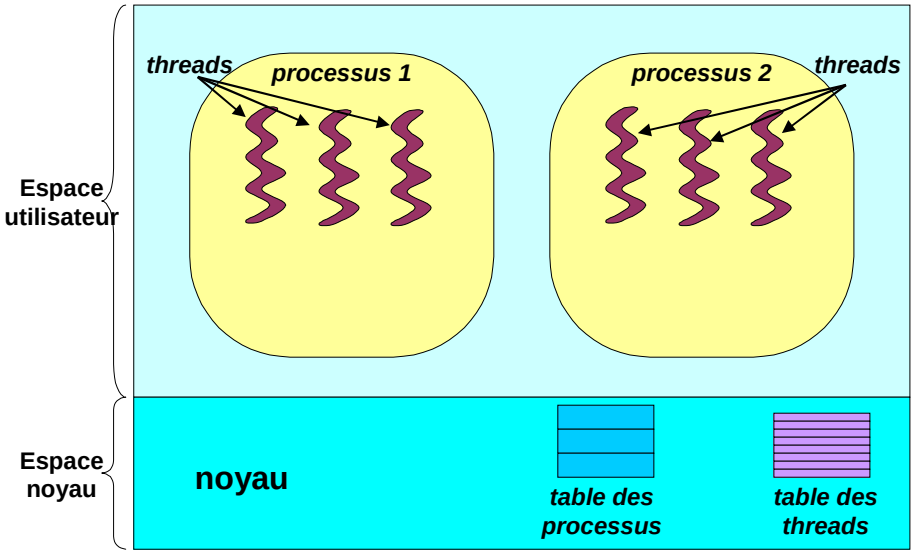
L'avantage de cette approche est qu'elle allége les commutations entre *threads* d'un même processus, puisque celles-ci s'effectuent au niveau utilisateur, sans passage par le mode noyau. On évite ainsi une commutation de contexte lourde.

Un inconvénient majeur est qu'au sein d'un même processus, un *thread* peut monopoliser le processeur pour lui seul. Par ailleurs, un *thread* bloqué au sein d'un processus suite à une demande de ressource ne pouvant être immédiatement satisfaite, bloque l'ensemble des *threads* du processus. Exemple : un processus est composé de trois *threads* utilisateurs. Un des trois *thread* demande l'ouverture d'un fichier. Comme le système d'exploitation ne sait pas que le processus est composé de plusieurs *threads*, il positionne

tout le processus dans l'état « bloqué ». Or, deux *thread*s, qui ne demandent pas l'ouverture du fichier mais du temps CPU, seront eux aussi bloqués.

7.3.3 Implémentation des *threads* au niveau noyau

Lorsque l'implémentation des *threads* est effectuée au niveau du noyau, ce dernier connaît l'existence des *threads* au sein d'un processus, et il attribue le processeur à chacun des *threads* de manière indépendante. Chaque descripteur de processus contient alors également une table des *threads* qui le composent, avec, pour chacun d'eux, la sauvegarde de son contexte processeur :



Gestion des *threads* au niveau noyau

Ainsi, il n'y a plus de blocage de tout le processus dès qu'un *thread* se retrouve bloqué en demandant une ressource. Par contre, chaque commutation de contexte nécessite un passage en mode noyau, ce qui est plus coûteux.

7.3.4 Fonctions système liées aux *threads*

Les primitives de création et de gestion des *threads* ressemblent aux primitives de création et de gestion des processus.

Comme chaque processus est implémenté par un numéro de processus (PID), chaque *thread* est identifié par un objet de type `pthread_t`. Les plus curieux peuvent regarder ce que cache ce type. Mais il est important de noter que le type `pthread_t` peut varier d'une implémentation à l'autre.

Pour utiliser la bibliothèque de gestion des *threads*, il faut penser à ajouter un :

```
#include <pthread.h>
```

au début des programmes. Ensuite, il faut ajouter un « `-lpthread` » à l'édition de lien (lors de la compilation).

Voici les routines à connaître :

- la primitive `pthread_create()` permet la création, au sein d'un processus, d'un nouveau fil d'exécution, identifié par l'identificateur `*thread` et attaché à l'exécution de la routine (`*start_routine`). Le fil d'exécution démarre son exécution au début de la fonction spécifiée et disparaît à la fin de l'exécution de celle-ci. Dans le prototype suivant, `*attr`, correspond aux attributs associés au *thread* si l'on

souhaite modifier les attributs standards affectés au moment de la création (cet argument est le plus souvent mis à NULL pour hériter des attributs standards), et `*arg` correspond à un argument passé au *thread* pour l'exécution de la fonction (`*start_routine`). La primitive renvoie la valeur 0 en cas de succès, et une valeur négative correspondant à l'erreur survenue sinon. Voici un extrait du manuel de cette fonction :

<<

```
#include <pthread.h>

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

Description : pthread_create() créé un nouveau thread
s'exécutant concurremment avec le thread appelant. Le
nouveau thread exécute la fonction start_routine() en
lui passant arg comme premier argument. Le nouveau
thread s'achève soit explicitement en appelant
pthread_exit(3) , ou implicitement lorsque la fonction
start_routine s'achève. Ce dernier cas est équivalent à
appeler pthread_exit(3) avec la valeur renvoyée par
start_routine comme code de sortie.

L'argument attr indique les attributs du nouveau thread.
Voir pthread_attr_init(3) pour une liste complète des
attributs. L'argument attr peut être NULL, auquel cas,
les attributs par défaut sont utilisés: le thread créé
est joignable (non détaché) et utilise la politique
d'ordonnancement usuelle (pas temps-réél).

Valeur Renvoyée : en cas de succès, l'identifiant du
nouveau thread est stocké à l'emplacement mémoire pointé
par l'argument thread, et 0 est renvoyé. En cas
d'erreur, un code d'erreur non nul est renvoyé.

Erreurs :

EAGAIN : pas assez de ressources système pour créer
         un processus pour le nouveau thread.
EAGAIN : il y a déjà plus de PTHREAD_THREADS_MAX
         threads actifs.
```

>>

- La primitive `pthread_exit(void *ret)` met fin au *thread* qui l'exécute. Elle retourne le paramètre `ret` qui peut être récupéré par un autre thread effectuant pour sa part un appel à la primitive « `pthread_join(pthread_t thread, void **retour)` » où `thread` correspond à l'identifiant du *thread* attendu, et `retour` correspond à la valeur `ret` retournée lors de la terminaison. L'exécution du processus qui effectue un appel à la fonction « `int pthread_join(pthread_t thread, void **ret)` » est suspendue jusqu'à ce que le *thread* attendu se soit achevé. Le paramètre `*ret` contient la valeur passée par le thread au moment de l'exécution de la primitive `pthread_exit()`. Voici les extraits des pages du manuel de `pthread_exit()` et de `pthread_join()` :

  <<

  ```
  #include <pthread.h>
  ```

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

Description : pthread_join suspend l'exécution du thread appelant jusqu'à ce que le thread identifié par ***th*** achève son exécution, soit en appelant pthread_exit(3), soit après avoir été annulé.

Si ***thread_return*** ne vaut pas NULL, la valeur renvoyée par ***th*** y sera enregistrée. Cette valeur sera soit l'argument passé à pthread_exit(3) , soit PTHREAD_CANCELED si le thread ***th*** a été annulé.

Le thread joint ***th*** doit être dans l'état joignable : il ne doit pas avoir été détaché par pthread_detach(3) ou par l'attribut PTHREAD_CREATE_DETACHED lors de sa création par pthread_create(3) .

Quand l'exécution d'un thread joignable s'achève, ses ressources mémoires (descripteur de thread et pile) ne sont pas désallouées jusqu'à ce qu'un autre thread le joigne en utilisant pthread_join. Aussi, pthread_join doit être appelée une fois pour chaque thread joignable pour éviter des "fuites" de mémoire.

Au plus, un seul thread peut attendre la mort d'un thread donné. Appeler pthread_join sur un thread ***th*** alors qu'un autre thread attend déjà la fin renvoie une erreur.

Annulation : pthread_join est un point d'annulation. Si un thread est annulé alors qu'il est suspendu dans pthread_join, l'exécution du thread reprend immédiatement et l'annulation est réalisée sans attendre la fin du thread ***th***. Si l'annulation intervient durant pthread_join, le thread ***th*** demeure non joint.

Valeur Renvoyée : en cas de succès, le code renvoyé par ***th*** est enregistré à l'emplacement pointé par thread_return, et 0 est renvoyé. En cas d'erreur, un code d'erreur non nul est renvoyé.

Erreurs :

ESRCH : Aucun thread correspondant à ***th*** n'a pu être trouvé.
EINVAL : Le thread ***th*** a été détaché.
EINVAL : Un autre thread attend déjà la mort de ***th***.
EDEADLK : L'argument ***th*** représente le thread appelant.

#include <pthread.h>

void pthread_exit(void ****retval***);

Description : pthread_exit termine l'exécution du thread appelant. Tous les gestionnaires de nettoyage enregistrés pour le thread courrant par pthread_cleanup_push(3) sont exécutés dans l'ordre

```
inverse (du plus récemment enregistré au plus ancien).
Les fonctions de fin pour les données spécifiques au
thread sont ensuite appelées pour toutes les clés qui
n'ont pas de valeur NULL associée à elles dans le thread
appelant (voir pthread_key_create(3) ). Enfin,
l'exécution du thread appelant est arrêtée.

L'argument retval est la valeur de retour du thread. Il
peut être consulté par un autre thread en utilisant
pthread_join(3) .

Valeur Renvoyée : la fonction pthread_exit ne rend
jamais la main.
```

>>

- Chaque thread est dote des attributs suivants:
  - l'adresse de départ et la taille de la pile qui lui est associée ;
  - la politique d'ordonnancement qui lui est associée (Cf. prochain chapitre) ;
  - la priorité qui lui est associée ;
  - son attachement ou son détachement. Un thread détaché se termine immédiatement sans pouvoir être pris en compte par un `pthread_join()`.
- Lors de la création d'un nouveau thread par un appel à la primitive `pthread_create()`, les attributs de ce thread sont fixés par l'intermédiaire de l'argument `pthread_attr_t *attr`. Si les attributs par défaut sont suffisants, ce paramètre est mis à NULL. Sinon, il faut au préalable initialiser une structure de type `pthread_attr_t`, en invoquant la primitive `pthread_attr_init()`, puis modifier chacun des attributs en utilisant les fonctions `pthread_attrgetXXX()` et `pthread_attr_setXXX()` qui permettent respectivement de consulter et modifier l'attribut XXX.

A noter que l'implémentation des *threads* sous Linux est réalisée au niveau du noyau, en s'appuyant sur la routine système « `clone()` ».

7.4 L'ordonnancement (le « *scheduler* »)

7.4.1 Rappels sur la commutation de contexte

Un système d'exploitation multitâche est capable de choisir un programme à exécuter parmi tous ceux éligibles (qui ont besoin de CPU). C'est le travail de l'**ordonnanceur** (*scheduler* en anglais). Lorsque cet ordonnanceur a choisi quel programme il va exécuter, il programme l'horloge afin que cette dernière lève une interruption matérielle au bout d'un temps donné. Puis, il lance l'exécution du programme de l'utilisateur (c'est la **réquisition** du CPU par le processus utilisateur).

Si, durant son exécution, le programme utilisateur n'a pas eu besoin de faire appel à une fonction système, l'interruption matérielle est levée par l'horloge. Le processus a épuisé son temps de CPU, et passe alors dans un état « prêt ». Regardons plus en détail ce qu'il se passe lorsque l'interruption est levée (on dit qu'il y a **préemption**) :

- toutes les interruptions matérielles commencent par sauvegarder les registres dans une entrée de la table des processus du processus en cours ;
- ensuite, l'information placée sur la pile par l'interruption est supprimée, et le pointeur de pile est défini de manière à pointer vers une pile temporaire utilisée par le

gestionnaire de processus. Des actions telles que la sauvegarde des registres et le positionnement du pointeur de pile ne peuvent pas être exprimées dans des langages de haut niveau comme le C. Elles sont donc accomplies par une petite routine en langage assembleur (propre à chaque CPU). Le plus souvent, la même routine sert à toutes les interruptions dans la mesure où la sauvegarde des registres est toujours une tâche identique, quelle que soit la cause de l'interruption ;

- lorsque la routine a terminé de s'exécuter, elle appelle une procédure C pour prendre en charge le reste des tâches à accomplir pour ce type spécifique d'interruption ;
- lorsque l'ordonnanceur a terminé son travail - au cours duquel il pourra très bien avoir fait passer certains processus en état prêt -, il est appelé pour exécuter le prochain processus. Ensuite, le contrôle retombe entre les mains du code assembleur, qui charge les registres et les « mappages » mémoire (Cf. prochains chapitres) pour le processus désormais actif et commence son exécution.

Voici le résumé des tâches que le système d'exploitation accomplit à son niveau le plus bas lorsqu'une interruption a lieu :

1. Le matériel place dans la pile le compteur ordinal, etc. ;
2. Le matériel charge un nouveau compteur ordinal à partir du vecteur d'interruptions ;
3. La procédure en langage assembleur sauvegarde les registres ;
4. La procédure en langage assembleur définit une nouvelle pile ;
5. Le service d'interruption en C s'exécute (généralement pour lire des entrées et les placer dans le tampon) ;
6. L'ordonnanceur décide du prochain processus à exécuter ;
7. La procédure C retourne au code assembleur ;
8. La procédure en langage assembleur démarre le nouveau processus actif.

Nous allons voir comment cet ordonnanceur fonctionne, c'est à dire quelles sont les différentes politiques possibles pour choisir un processus parmi tous ceux qui sont éligibles.

7.4.2 La politique du « *premier arrivé, premier servi* »

C'est l'ordonnanceur le plus simple à programmer. Il consiste à mettre en place une simple file d'attente, et à lancer les processus les uns à la suite des autres, dans leur ordre d'arrivée. Il n'y a pas de préemption : chaque processus s'exécute jusqu'à ce qu'il soit terminé, ou jusqu'à ce qu'il fasse appel à une primitive système.

Le grand inconvénient de cette solution est que les processus qui ont les plus petits temps d'exécution (beaucoup d'appel système par exemple) sont pénalisés par les processus qui consomment simplement du CPU (longs calculs par exemple).

7.4.3 La politique par priorité

Cette fois ci, chaque processus possède une priorité (priorité constante définie au départ). Avec cette politique, le programme qui va faire la réquisition du CPU est choisi comme étant le programme dans l'état « prêt » qui aura la plus grande priorité. Si plusieurs programmes éligibles on la même priorité, une file d'attente est gérée.

Il existe une variante où une préemption est faite à un processus sitôt qu'un processus plus prioritaire passe à l'état « *prêt* ».

Le gros inconvénient de cette méthode est qu'elle engendre des « *famines* » chez les processus ayant la priorité la plus faible : les processus ayant la priorité la plus élevée

consomment du CPU, au détriment des processus les moins prioritaires, qui peuvent se trouver dans une situation où ils ne sont plus jamais lancés.

Une des solutions à ce problème de famine (variantes) est de faire baisser dynamiquement la priorité des processus quand ils viennent d'avoir du CPU.

7.4.4 La politique du tourniquet (*round robin*)

C'est la solution la plus souvent mise en place dans les système en « temps partagé ». Ici, le temps est découpé tranches (on parle de « quantum de temps », de l'ordre de 10 à 100 ms. selon les systèmes).

Lorsqu'un processus est élu, il s'exécute durant un quantum de temps, avant d'être *préempté* (sauf s'il a lancé une fonction système entre temps).

Une file des processus permet de lancer les processus les uns à la suite des autres (comme dans la politique du « *premier arrivé, premier servi* »), mais cette fois-ci, pour un temps donné.

La recherche de la valeur du quantum de temps optimum est très importante : s'il est trop grand, l'impression de temps partagé disparaît. S'il est trop petit, il y aura beaucoup de commutation de contexte, ce qui réduit les performances.

7.4.5 Les politiques d'ordonnancement sous Linux

Trois politiques d'ordonnancement différentes sont mises en œuvre sous Linux :

- les deux premières, appelées « `SCHED_FIFO` » et « `SCHED_RR` » sont des algorithmes dit « *temps-réels* ». Les processus identifiés comme régis par ces deux politiques temps-réel sont toujours prioritaires ;
- la troisième, appelée « `SCHED_OTHER` » est utilisée par les processus « classiques ».

La politique d'ordonnancement est indiquée dans le champs « policy » du bloc de contrôle de processus (PCB, de type « `struct task_struct` » sous Linux comme déjà évoqué).

L'ordonnancement réalisé par le noyau est découpé en périodes. Au début de chaque période, le système calcule les quantums de temps attribués à chaque processus, c'est-à-dire, le nombre de « *ticks* » d'horloge durant lequel le processus peut s'exécuter. Une période prend fin lorsque l'ensemble des processus initialisés sur cette période a achevé son quantum. Le lecteur aura noté qu'avec cette méthode, tous les processus n'aura pas n'auront pas nécessairement le même temps de CPU. Les algorithmes mis en place peuvent se résumer ainsi :

- **Ordonnancement des processus temps-réel :** les processus temps réel sont qualifiés par une priorité fixe dont la valeur évolue entre 1 et 99 (paramètre `rt_priority` du bloc de contrôle du processus), et sont ordonnancés soit selon la politique « `SCHED_FIFO` », soit selon la politique « `SCHED_RR` » :
    - La politique « `SCHED_FIFO` » est une politique préemptive qui offre un service de type « *premier arrivé, premier servi* » entre processus de même priorité. A un instant *t*, le processus « `SCHED_FIFO` » de plus haute priorité le plus âgé est élu. Ce processus poursuit son exécution, soit jusqu'à ce qu'il se termine, soit jusqu'à ce qu'il soit préempté par un processus temps-réel plus prioritaire devenu prêt. Dans ce dernier cas, le processus préempté réintègre la tête de file correspondant à son niveau de priorité ;
    - La politique « `SCHED_RR` » est une politique du tourniquet à quantum de temps entre processus de même priorité. Dans ce cas, le processus élu est encore le processus « `SCHED_RR` » de plus forte priorité. Mais ce processus

ne peut poursuivre son exécution au-delà du quantum de temps qui lui a été attribué. Le quantum de temps (paramètre `counter` du bloc de contrôle du processus) correspond à un certain nombre de « *ticks* » horloge. Une fois ce quantum expiré, le processus élu est préempté et il réintègre la queue de la file correspondant à son niveau de priorité. Le processeur est alors alloué à un autre processus temps-reèl, qui est le processus temps réel le plus prioritaire ;

- **Ordonnancement des processus classiques :** Les processus classiques sont qualifiés par une priorité dynamique qui varie en fonction de l'utilisation faite par le processus des ressources de la machine et notamment du processeur. Cette priorité dynamique représente le quantum alloué au processus, c'est-à-dire le nombre de « *ticks* » horloge dont il peut disposer pour s'exécuter. Ainsi, un processus élu voit sa priorité initiale décroître en fonction du temps processeur dont il a disposé. Ainsi, il devient moins prioritaire que les processus qui ne se sont pas encore exécutés. Ce principe, appelé extinction de priorité, évite les problèmes de famine des processus de petite priorité.

  La politique « `SCHED_OTHER` » décrite ici correspond à la politique d'ordonnancement des systèmes Unix classiques. Plus précisément, la priorité dynamique d'un processus est égale à la somme entre un quantum de base (paramètre « `priority` » du bloc de contrôle du processus) et un nombre de « *ticks* » horloge restants au processus sur la période d'ordonnancement courante (paramètre « `counter` » du bloc de contrôle du processus). Un processus fils hérite à sa création du quantum de base de son père et de la moitié du nombre de *ticks* horloge restant de son père.

  Le processus possède ainsi une priorité égale à la somme entre les champs « `priority` » et « `counter` » de son bloc de contrôle, et s'exécute au plus durant un quantum de temps égal à la valeur de son champ « `counter` ».

La fonction d'ordonnancement Linux (`schedule()`) gère deux types de files :

- La file des processus actifs (*runqueue*), qui relie entre eux les blocs de contrôle des processus à l'état « *prêts* » (état « `TASK_RUNNING` ») ;
- Les files des processus en attente (*wait queues*), qui relient les PCB des processus bloqués (états « `TASK_INTERRUPTIBLE` » et « `TASK_UNINTERRUPTIBLE` ») dans l'attente d'un même événement. Il y a autant de « *wait queues* » (files d'attentes) qu'il y a d'événements sur lesquels un processus peut être en attente (ouverture d'un fichier, attente de lecture dans une pile de protocole réseau, etc.).

L'état d'un processus sous Linux est positionné dans le champ « **state** » du PCB, et peut prendre les valeurs suivantes (nous reviendrons sur certaines de ces valeurs dans les prochains chapitres, comme lorsque nous traiterons les signaux) :

```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_ZOMBIE             4
#define TASK_STOPPED            8
#define TASK_EXCLUSIVE          32
```

Les plus curieux qui s'intéressent au fonctionnement de cette fonction `schedule()` pourront regarder comment fonctionnent deux fonctions qu'elle appelle :

- la fonction `goodness()`, qui effectue le choix du prochain processus à exécuter,
- la fonction `switch_to()`, qui est la macro qui réalise la commutation de contexte. Cette fonction sauvegarde le contexte du processus courant, commute l'exécution du noyau sur la pile noyau du processus à élire, puis restaure le contexte du processus à élire.

Le bilan : le principal inconvénient de l'ordonnanceur Linux vient de la prise en compte des processus temps-réel. Un programme temps réel mal intentionné peut facilement monopoliser les ressources de la machine et provoquer une famine.

En revanche, si on écarte le cas des processus temps-réel, le risque de famine est inexistant : chaque processus se voit élu dans des délais acceptables grâce au principe de vieillissement. Et comme l'ordonnancement se fait aussi au changement d'état du processus courant, on tire partie des blocages d'un processus pour éviter les temps morts. Le processeur est utilisé de manière optimale.

8 Le démarrage d’un système Linux

8.1 Le chargeur de noyau

A l’allumage d’un ordinateur de type PC, le tout premier programme à être lancé est en réalité le BIOS (Basic Input Output System). Il s’agit de code contenant un certain nombre de routines, contenues dans une mémoire morte (ROM) ou de la mémoire « flash », située sur la carte mère. Ces routines servent à faire des opérations basiques liées aux entrées/sorties.

Celui-ci va chercher au début du 1er disque dur afin d'y trouver un chargeur (en réalité, aujourd’hui, le BIOS sait aussi aller chercher ce chargeur sur un CD-Rom, une clé USB, voire sur le réseau). Ce chargeur est un petit programme dont le but sera de lancer le système d'exploitation complet. Les chargeurs récents (GRUB, Lilo, etc.) sont capables de laisser à l'utilisateur le choix du système à lancer (on parlera de multiboot). Plusieurs OS peuvent alors cohabiter sur une même machine.

Ce chargeur va ensuite charger le MBR (master boot record), situé dans les premiers secteurs du disque considéré. Ce MBR contient la « table des partitions » (nous verrons l’utilité de cette table dans le chapitre sur la gestion de fichiers), ainsi qu’un second chargeur, dont le rôle sera de charger en mémoire et démarrer le noyau du système d’exploitation. A noter que des options peuvent être passées en paramètre au noyau, comme par exemple (liste non exhaustive, loin s’en faut) :

- **`vga=XXX`** : sert à changer la résolution d'écran utilisée pendant le démarrage. Utile si celle par défaut n'est pas reconnue par la carte graphique.
- **`no-scroll`** : permet de désactiver le défilement du texte sur l'écran. A utiliser notamment avec des terminaux braille pour lesquels le défilement peut poser problème.
- **`noapic`** : désactive le mécanisme de partage d’IRQ par plusieurs périphériques (qui pose parfois des problèmes avec certains d’entre eux) ;
- **`mem=XXX`** : permet d'indiquer la valeur de mémoire vive présente sur la machine pour le cas ou l'auto-detection échouerait. On peut utiliser des lettres pour cette taille. Par exemple 512M désignera 512 Méga-octets de mémoire ;
- **`init=XXX`** : permet d'indiquer de manière explicite quel programme doit être lancé après l'initialisation du noyau (Cf. détails ci-dessous).

8.2 Le processus « *init* »

Une fois que le noyau a fini tout son travail d'initialisation, il lance un programme qui va devenir le père de tous les autres processus, et portera le PID (identifiant de processus) 1.

Ce programme s'appelle généralement « *init* ». Le noyau va en fait essayer de lancer successivement (via la fonction `start_kernel()`) les programmes suivants :

- `/sbin/init`
- `/etc/init`
- `/bin/init`
- `/bin/sh`

Dès qu'un de ceux-là est lancé avec succès, les autres ne sont même pas testés. Ainsi, on constate que le programme `init` est cherché dans différents répertoires (`/sbin`, `/etc` et finalement `/bin`) . S'il ne se trouve dans aucun de ceux-là, le noyau tente de lancer un

shell (interpréteur de commandes), afin que l'utilisateur ait accès au système. Si cela échoue également, le noyau affiche un message d'erreur et s'arrête.

Le processus lance toujours quatre *threads* (`kupdate` et `bdflush` qui gèrent le cache disque, et `kswap` et `kpiod` qui gèrent la mémoire virtuelle). Puis, `init` va charger tous les autres processus. Le comportement d'`init` se configure à l'aide du fichier « `/etc/inittab` ». En voici un exemple d'extrait :

```
id:5:initdefault:
si::sysinit:/etc/rc.sysinit
l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6
ca::ctrlaltdel:/sbin/shutdown -t3 -r now
```

Chaque ligne est construite de la même manière avec les champs suivants :

```
Identifiant:Niveau d'exécution:Action:Programme
```

- **`Identifiant :`** une chaîne de caractères choisie par l'utilisateur (sauf dans certains cas particuliers) et permettant d'identifier la ligne.
- **`Niveaux d'exécution :`** les niveaux d'exécution (détaillés dans le chapitre suivant) pour lesquels cette ligne doit être prise en compte.
- **`Action :`** contient une des actions prédéfinies indiquant ce qui doit être fait. Le tableau suivant les liste.
- **`Programme :`** le programme qui doit être exécuté lorsque l'on rentre dans les niveaux indiqués.

Selon l'action choisie, le comportement sera différent et certains champs peuvent être ignorés. Voici la description des actions le plus souvent utilisées :

- **`initdefault :`** permet d'indiquer le niveau d'exécution à utiliser par défaut. Le champ « `Niveaux d'exécution` » contiendra alors une seule valeur qui sera ce niveau par défaut. Dans une telle ligne, le champ « `Programme` » est ignoré ;
- **`sysinit :`** le champs « `Programme` » contient le chemin vers un exécutable qui sera lancé en tout premier par « `init` » (donc juste après que le noyau ait terminé ses initialisations). Dans une telle ligne, le champ « `Niveaux d'exécution` » est ignoré ;
- **`wait :`** lorsque le système passera dans le niveau d'exécution spécifié, « `init` » exécutera la commande indiquée puis attendra qu'elle se termine ;
- **`respawn :`** semblable à « `wait` », si ce n'est qu'à chaque fois que le programme se termine, « `init` » le relancera ;
- **`ctrlaltdel :`** permet d'indiquer une commande devant être exécutée lorsque l'utilisateur presse la combinaison de touches Ctrl-Alt-Suppr. Dans une telle ligne, le champ « `Niveaux d'exécution` » est ignoré.

8.3 Les niveaux d'exécution

Sur les systèmes GNU/Linux, il existe plusieurs niveaux d'exécution possibles (appelés aussi modes d'exécution). Il s'agit en fait de modes de démarrage différents, qui diffèrent les uns des autres par les services qui y sont lancés.

La convention choisie est celle appelée « *System V init* » qui définit la manière dont doivent être gérés les différents niveaux. Dans le fichier « `/etc/inittab` » donné en exemple dans le chapitre précédent, on peut voir que c'est le programme « `/etc/rc.d/rc` » qui gère cela. Il est lancé avec en paramètre le numéro de niveau lorsque l'on a besoin de basculer dans un certain niveau d'exécution.

On trouve en général 7 niveaux d'exécution numérotés de 0 à 6. Leur utilisation est libre, mais traditionnellement, ils prennent la signification suivante :

- **0 (arrêt) :** passer dans ce niveau provoque un arrêt de la machine ;
- **1 (maintenance, ou sigle user) :** on a directement accès à un shell, mais quasiment aucun service n'est lancé. Utile pour le dépannage en cas de problème important ;
- **2 (multi-utilisateurs simple) :** plusieurs utilisateurs peuvent se connecter en mode texte. Mais les services sont limités (souvent pas de réseau par exemple) ;
- **3 (multi-utilisateurs complet) :** tous les services nécessaires sont démarrés et plusieurs utilisateurs peuvent se connecter en mode texte ;
- **4 (mode utilisateur) :** généralement non employé, il peut être librement utilisé ;
- **5 (mode graphique) :** identique au mode 3, mais les utilisateurs peuvent se connecter en mode graphique (X11) et disposer d'un gestionnaire de fenêtre (Gnome, KDE, etc.) ;
- **6 (redémarrage) :** passer dans ce niveau entraîne un redémarrage de la machine.

Après le démarrage, le système se trouve dans le mode indiqué par « `initdefault` » dans « `/etc/inittab` ». Pour pouvoir changer, il existe un outil appelé « `telinit` ». Il suffit de le lancer en lui passant en paramètre le numéro du niveau souhait. Par exemple, pour redémarrer la machine, il suffit de lancer la commande suivante en étant identifié comme étant l'utilisateur `root` :

```
# telinit 6
```

Bien qu'il y ait des conventions pour ce qui doit être fait dans chaque mode, cela peut être entièrement changé.

Pour comprendre comment influer sur ce comportement, il faut d'abord savoir ce que fait le programme « `/etc/rc.d/rc` » (ou un autre selon les systèmes et le contenu du fichier « `inittab` »).

Ce programme (souvent, un scipt shell, ou perl) gère les niveaux d'exécution en allant consulter le contenu du répertoire « `/etc/rc.d/rc`*X*`.d` » (ou « `/etc/rc`*X*`.d` » dans certain systèmes comme la distribution Debian), avec *X* correspondant au numéro du niveau devant être changé.

Ce script va rechercher d'abord tous les exécutables s'y trouvant et dont le nom commence par la lettre « K » (pour *Kill*) suivie par deux chiffres. Il lance ces programmes en leur passant en paramètre « `stop` ». Cela correspond aux services qui doivent être arrêté dans ce mode d'exécution là. Ils sont lancés dans l'ordre croissant du nombre indiqué après le K, ce qui permet de les ordonner (démarrer tel service avant tel autre).

C'est ensuite au tour des programmes dont le nom commence par la lettre « S » (pour *Start*) puis également un nombre sur deux chiffres. Ils sont lancés de la même manière que les précédents si ce n'est que c'est le paramètre « `start` » qui est passé.

Pour illustrer ceci, voici un contenu possible d'un de ces répertoires :

```
# ls /etc/rc.d/rc3.d
K15httpd
K20samba
K45named
S10network
S55sshd
S99local
```

Dans cet exemple (qui est totalement farfelu), lorsque le système passe dans le niveau 3 (soit au démarrage si c'est celui par défaut, soit ensuite, si c'est l'utilisateur qui le demande), on arrêtera les services *httpd* (serveur web), *samba* (serveur de fichiers suivant le protocole CIF, c'est-à-dire « *à la sauce Windows* ») et *named* (serveur de noms). Ensuite seront démarrés les services *network* (pour la prise en charge du réseau), *sshd* (serveur de connexion distante sécurisée par le protocole SSL) et *local* (qui contient traditionnellement par convention des programmes devant être exécuté en fin de démarrage).

Comme des programmes peuvent exister dans plusieurs niveaux différents, on ne trouvera en réalité dans les répertoires « `/etc[/rc.d]/rcX.d` » que des liens symboliques pointant vers des fichiers situés dans le répertoire « `/etc/rc.d/init.d` » (ou « `/etc/init.d` » sur certains systèmes). Ces fichiers sont en réalité pour la plupart des *scripts shell*, et le même sera lancé pour le démarrage ou l'arrêt. C'est donc de la responsabilité du script de voir s'il a été appelé avec le paramètre « `start` » ou le paramètre « `stop` », afin de savoir quelles actions entreprendre.

Sachant cela, ajouter ou supprimer des services dans un niveau donné revient uniquement à créer ou supprimer des liens symboliques. En reprenant l'exemple précédent, voici un exemple (tout aussi farfelu) de ce qui pourrait être fait :

```
# cd /etc/rc.d/rc3.d
# rm S55sshd
# ln -s /etc/rc.d/init.d/ftpd S40ftpd
```

Ces actions vont faire en sorte que dans le niveau 3, le serveur *ssh* ne sera plus exécuté. En revanche, un serveur *ftp* sera lancé. Tout cela suppose qu'un script « `ftpd` » soit présent dans le répertoire « `/etc/rc.d/init.d/` », et qu'il fournisse le service attendu (lancer un serveur FTP). Les distributions GNU/Linux incluent ce genre de script lorsqu'un service est installé. L'utilisateur n'a donc ensuite qu'à modifier les liens symboliques.

Le squelette d'un tel script situé dans le répertoire « `/etc/rc.d/init.d/` » ressemble souvent à :

```
# Exemple de squelette de fichier /etc/init.d/xxx sur Debian :
# L'instruction 'set -e' au début du script
# pour qu'il s'interrompe dès qu'une commande
# retourne une valeur de retour non nulle (erreur).
set –e

# On positionne les variables d'environnement en
# fonction du système d'exploitation :
PATH=/sbin:/bin:/usr/sbin:/usr/bin
DESC="description_de_ce_que_fait_le_demon"
NAME=nom_du_demon
DAEMON=/usr/sbin/$NAME
PIDFILE=/var/run/$NAME.pid
SCRIPTNAME=/etc/init.d/$NAME

# Fonction qui démarre le service :
d_start() {
    start-stop-daemon --start --quiet \
                  --pidfile $PIDFILE --exec $DAEMON
}
```

```
# Fonction qui arête le service :
d_stop() {
    start-stop-daemon --stop --quiet \
                --pidfile $PIDFILE --name $NAME
}

# Fonction qui force le service à relire sa configuration :
d_reload() {
    start-stop-daemon --stop --quiet \
                --pidfile $PIDFILE --name $NAME --signal 1
}

# Reste à regarder quelle est la chaîne passée en paramètre,
# et à appeler la bonne fonction :
case "$1" in
  start)
        echo -n "Starting $DESC: $NAME"
        d_start
        echo "."
        ;;
  stop)
        echo -n "Stopping $DESC: $NAME"
        d_stop
        echo "."
        ;;
  reload)
        echo -n "Reloading $DESC configuration..."
        d_reload
        echo "done."
        ;;
  restart|force-reload)
        echo -n "Restarting $DESC: $NAME"
        d_stop
        sleep 1
        d_start
        echo "."
        ;;
  *)
        echo "Usage: $SCRIPTNAME {start|stop |restart|force-
reload|reload}" >&2
        exit 1
        ;;
esac

exit 0
```

8.4 L’arborescence des processus

Vous l’aurez compris, le mécanisme de multitâche sous Linux repose sur le dédoublement des processus à l’aide de la fonction `fork()`, puis le recouvrement du processus fils à l’aide des fonctions de recouvrement `exec`*`XX`*`()`, et enfin, l’utilisation de *threads* gérés au niveau du noyau.

La commande `pstree` permet de rendre compte de l’arborescence des processus lancés. Exemple sur une station *Debian* :

```
# pstree -A -c –n
init-+-ksoftirqd/0
     |-events/0
     |-khelper
     |-kthread-+-kblockd/0
     |         |-pdflush
     |         |-pdflush
     |         |-aio/0
     |         |-kseriod
     |         `-khubd
     |-xbox_extsmi
     |-kswapd0
     |-kjournald
     |-portmap
     |-apache2-+-apache2
     |         |-apache2
     |         |-apache2
     |         `-apache2
     |-named
     |-freepopsd
     |-spamd-+-spamd
     |       `-spamd
     |-cron
     |-klogd
     |-lpd
     |-master-+-qmgr
     |        `-pickup
     |-nmbd
     |-smbd---smbd
     |-sshd---sshd---bash---pstree
     |-syslogd
     |-xinetd
     |-rpc.statd
     |-ntpd
     |-rpc.nfsd
     |-rpc.mountd
     |-miniserv.pl
     |-smartd
     |-winbindd---winbindd
     |-getty
     |-getty
     `-dhcpd3
```

De plus, la commande `ps` (Cf. `man ps`) permet d'obtenir un résultat moins visuel mais plus complet sur le chaînage de lancement des processus.

9 Les signaux

9.1 Présentation

Comme nous l'avons déjà vu précédemment, un signal est en quelque sorte un message envoyé par le noyau à un processus (suite à un événement, ou à la demande d'un autre processus). Un signal ne contient pas d'information en soit, excepté le nom du signal.

L'arrivée d'un signal oblige un processus à exécuter une fonction particulière liée au signal, appelée « ***handler de signal*** » ou « ***gestionnaire de signal*** ». On dit que le processus est « ***dérouté*** » ou « **détourné** ».

Les signaux sont parfois appelés « *interruptions logicielles* ». En effet, un parallèle peut-être fait entre le déroutement d'un processus suite à la réception d'un signal et le traitement des interruptions matérielles (même si en pratique, ces deux types d'interruptions sont différents).

Par contre, c'est grâce à cette notion de *signal* que les **trappes** (ou **exceptions**, c'est à dire gestion des évènements non désirés comme par exemple : « *division par 0* », « *accès par un processus d'une zone mémoire où il n'a pas le droit d'accéder* », etc.) sont traitées.

Nous verrons que le noyau Linux propose un mécanisme classique d'envoi et de gestion des signaux (semblable aux autres noyaux Unix), mais il fournit aussi un mécanisme de signaux « *temps réels* ».

9.2 Les signaux classiques

Depuis le noyau Linux version 2.2, ce système d'exploitation permet de gérer 64 signaux (le signal 0, qui ne porte pas de nom, 31 signaux classiques – qui ont une signification particulière et un nom particulier –, et 31 signaux temps réel).

Voici la liste des 31 signaux classiques sous Linux (listing extrait du fichier « `/usr/include/bits/signum.h` » d'une distribution Debian 3.1 :

```
#define SIGHUP     1  /* Hangup (POSIX).  */
#define SIGINT     2  /* Interrupt (ANSI).  */
#define SIGQUIT    3  /* Quit (POSIX).  */
#define SIGILL     4  /* Illegal instruction (ANSI).  */
#define SIGTRAP    5  /* Trace trap (POSIX).  */
#define SIGABRT    6  /* Abort (ANSI).  */
#define SIGIOT     6  /* IOT trap (4.2 BSD).  */
#define SIGBUS     7  /* BUS error (4.2 BSD).  */
#define SIGFPE     8  /* Floating-point exception (ANSI).  */
#define SIGKILL    9  /* Kill, unblockable (POSIX).  */
#define SIGUSR1   10  /* User-defined signal 1 (POSIX).  */
#define SIGSEGV   11  /* Segmentation violation (ANSI).  */
#define SIGUSR2   12  /* User-defined signal 2 (POSIX).  */
#define SIGPIPE   13  /* Broken pipe (POSIX).  */
#define SIGALRM   14  /* Alarm clock (POSIX).  */
#define SIGTERM   15  /* Termination (ANSI).  */
#define SIGSTKFLT 16  /* Stack fault.  */
#define SIGCLD    SIGCHLD /* Same as SIGCHLD (System V).  */
#define SIGCHLD   17  /* Child status has changed (POSIX).  */
#define SIGCONT   18  /* Continue (POSIX).  */
#define SIGSTOP   19  /* Stop, unblockable (POSIX).  */
#define SIGTSTP   20  /* Keyboard stop (POSIX).  */
#define SIGTTIN   21  /* Background read from tty (POSIX).  */
#define SIGTTOU   22  /* Background write to tty (POSIX).  */
#define SIGURG    23  /* Urgent condition on socket (4.2 BSD).  */
#define SIGXCPU   24  /* CPU limit exceeded (4.2 BSD).  */
```

```
#define SIGXFSZ   25  /* File size limit exceeded (4.2 BSD).  */
#define SIGVTALRM 26  /* Virtual alarm clock (4.2 BSD).  */
#define SIGPROF   27  /* Profiling alarm clock (4.2 BSD).  */
#define SIGWINCH  28  /* Window size change (4.3 BSD, Sun).  */
#define SIGPOLL   SIGIO /* Pollable event occurred (System V).  */
#define SIGIO     29  /* I/O now possible (4.2 BSD).  */
#define SIGPWR    30  /* Power failure restart (System V).  */
#define SIGSYS    31  /* Bad system call.  */
```

A noter que toutes les valeurs des signaux ne sont pas nécessairement les mêmes suivant les différents Unix, voire entre deux versions majeures de noyaux Linux. Aussi, il conviendra de toujours utiliser dans vos programmes le nom du signal (ex : `SIGURG` ou `SIGKILL`) plutôt que leur valeur numérique (23 ou 9).

Si la plupart des signaux on une signification imposée par le système, il existe deux signaux (`SIGUSR1` et `SIGUSR2`) qui peuvent être utilisés par le programmeur à sa guise (il peut leurs assigner par convention la sémantique qu'il désire).

Pour traiter les signaux, le descripteur de processus (PCB) contient les champs suivants :

- le champ **`signal`** est de type `sigset_t` qui stocke les signaux envoyés au processus. Cette structure est constituée de deux entiers non signés de 32 bits chacun (soit 64 bits au total), chaque bit représentant un signal. Une valeur à 0 indique que le signal correspondant n'a pas été reçu tandis qu'une valeur à 1 indique que le signal a été reçu ;
- le champ **`blocked`** est de type `sigset_t` (64 bits) et stocke les signaux bloqués (c'est-à-dire les signaux dont la prise en compte est retardée) ;
- le champ **`sigpending`** est un drapeau (*flag*) indiquant s'il existe au moins un signal non bloqué en attente ;
- le champ **`gsig`** est un pointeur vers une structure de type `signal_struct`, qui contient notamment, pour chaque signal, la définition de l'action qui lui est associée.

En pratique, la gestion des signaux se retrouve à plusieurs niveaux dans le système d'exploitation :

- envoi d'un signal par un processus (par le noyau ou par le processus d'un utilisateur, ou suite à une exception),
- prise en compte du signal et exécution du code qui lui est associé ;
- enfin, il existe une interaction entre certains appels systèmes et les signaux.

Nous allons détailler chacun de ces mécanismes.

9.2.1 L'envoi d'un signal

L'envoi d'un signal peut être effectué :

- par un processus à destination d'un autre processus, à l'aide de la fonction système « `kill()` » (Cf. prochains chapitres). Il existe une commande « `kill` » qui peut être lancée depuis un *shell*, et qui appelle la fonction système « `kill()` ». Exemple d'utilisation :
  ```
  # kill -SIGTERM num_de_pid
  ```
- ou alors, l'exécution du processus a levé une trappe, et le gestionnaire d'exception associé positionne un signal pour signaler l'erreur détectée. Par exemple, suite à une division par zéro, le gestionnaire d'exception `divide_error()` positionne le signal `SIGFPE`.

Lors de l'envoi d'un signal, le noyau exécute la routine du noyau `seng_sig_info()` qui positionne tout simplement à 1 le bit correspondant au signal reçu dans le champ `signal`

du PCB du processus destinataire. La fonction se termine immédiatement dans les deux cas suivant :

- le numéro du signal émis est 0. Ce numéro n'étant pas un numéro de signal valable, le noyau retourne immédiatement une valeur d'erreur ;
- ou lorsque le processus destinataire est dans l'état « *zombie* ».

Le signal ainsi délivré (mise à 1 du bit dans le champ `signal`) mais pas encore pris en compte par le processus destinataire (le *handler de signal* n'a pas encore été lancé) est qualifié de **signal pendant**.

A noter que ce mécanisme de mémorisation de la réception du signal permet effectivement de mémoriser qu'un signal a été reçu, mais ne permet pas de mémoriser combien de signaux d'un même type ont été reçus. Par exemple, si deux processus fils se terminent de façon rapprochée dans le temps, le premier enverra un signal `SIGCHLD` au père. Il est tout à fait possible que la mort du second fils engendre le même signal `SIGCHLD` au père, alors que le premier signal `SIGCHLD` était encore pendant (i.e. non pris en compte par le père). Ainsi, la réception d'un signal `SIGCHLD` par le père lui indique « *qu'au moins* » un fils s'est terminé, sans que le système ne puisse lui indiquer combien.

9.2.2 La prise en compte d'un signal

La prise en compte d'un signal par un processus s'effectue lorsque celui-ci s'apprête à quitter le mode noyau pour repasser en mode utilisateur (c'est à dire lors du retour d'un appel à une fonction système, ou lorsque le scheduler élit ce processus et souhaite le faire démarrer). On dit alors que le signal (qui était pendant) est **délivré** au processus.

Cette prise en compte est réalisée par la routine du noyau `do_signal()` qui traite chacun des signaux pendants du processus. Trois types d'actions peuvent être réalisés:

- ignorer le signal ;
- exécuter l'action par défaut ;
- exécuter une fonction spécifique installée consciemment par le programmeur.

Ces actions sont stockées pour chaque signal dans le champ `gsig.sa_handler` du bloc de contrôle du processus (PCB). Ce champ prend la valeur `SIG_IGN` dans le cas où le signal doit être ignoré, la valeur `SIG_DFL` s'il faut exécuter le *handler* de signal par défaut, et enfin contient l'adresse de la fonction spécifique à exécuter dans le troisième cas.

Lorsque le signal est ignoré, aucune action n'est exécutée. Il existe une exception à cette règle : le signal `SIGCHLD`. En effet, si le processus a programmé le signal `SIGCHLD` afin que ce dernier soit ignoré, le noyau force le processus à lire le code retour des zombies, afin que ces derniers libèrent leurs ressources (principalement des entrées dans les tables des PCB) et soient effacés définitivement.

Les actions par défaut qui peuvent être exécutées suites à un envoi de signal sont au nombre de cinq :

- fin du processus,
- fin du processus et création (si l'utilisateur l'a décidé) d'un fichier `core` dans le répertoire courant, qui est un fichier contenant l'image mémoire du processus, et qui peut être analysé par un débogueur,
- le signal est ignoré,
- le processus est stoppé (il passe dans l'état `TASK_STOPPED`),
- le processus reprend son exécution (l'état est positionné à `TASK_RUNNING`).

Voici un tableau qui indique les actions par défaut liées aux différents signaux :

Actions par défaut	Nom du signal
Fin du process	SIGHUP, SIGINT, SIGBUS, SIGKILL, SIGUSR1, SIGUSR2, SIGPIPE, SIGALRM, SIGTERM, SIGSTKFLT, SIGXCOU, SIGXFSZ, SIGVTALRM, SIGPROF, SIGIO, SIGPOLL, SIGPWR, SIGUNUSED
Fin du process et création core	SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGIOT, SIGFPE, SIGSEGV
Signal ignoré	SIGCHLD, SIGURG, SIGWINCH
Processus stoppé	SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU
Processus redémarré	SIGCONT

Tout processus peut installer un traitement spécifique pour chacun des signaux, **hormis pour le signal `SIGKILL`**. Ce traitement spécifique remplace alors le traitement par défaut défini par le noyau. Ce traitement spécifique peut être de deux natures différentes :

- demande à ignorer le signal (prend alors la valeur `SIG_IGN`),
- définit une action particulière programmée dans une fonction C attachée au code de l'utilisateur (prend alors la valeur du *handler* ou *gestionnaire du signal*).

A noter que lorsqu'un signal pendant est délivré au processus, si ce signal est détourné par une fonction programmée par l'utilisateur, le noyau :

- revient en mode « *utilisateur* »,
- lance l'exécution du handler,
- et à la fin de l'exécution de ce handler, le noyau s'arrange pour que le processus reprenne son exécution normale (c'est la fonction `restore_sigcontext()` qui effectue ce travail).

Voici en pratique ce qui se passe. Lorsque le noyau est prêt pour redonner la main à un processus en mode utilisateur (soit après un appel système, soit après que le scheduler ait élu ce processus), le noyau appelle tout d'abord la fonction `do_signal()`. Cette fonction vérifie qu'il n'existe pas de signal pendant. Si c'est le cas, le noyau appel le handler correspondant (avec le privilège utilisateur). Lorsque cette fonction se termine, si aucun autre signal n'est pendant, le noyau retourne à l'exécution du programme à l'aide de la fonction `restore_sigcontext()`.

9.2.3 Signaux et appels système

Lorsqu'un processus exécute un appel système qui se révèle bloquant (exemple : appel de la fonction `open()` pour ouvrir un fichier qui n'est pas encore ouvert par un autre processus ; il va falloir aller lire et interpréter des blocs sur le disque dur ; pendant ce temps, le processus est « *bloqué* »), le processus est placé par le noyau dans l'état `TASK_INTERRUPTIBLE` ou dans l'état `TASK_UNINTERRUPTIBLE` :

- s'il est dans l'état `TASK_INTERRUPTIBLE`, le processus peut être réveillé par le système lorsqu'il reçoit un signal ;
- inversement, le système place le processus dans l'état `TASK_UNINTERRUPTIBLE` si le processus ne doit pas être réveillé par un signal.

Lorsqu'un processus placé dans l'état « `TASK_INTERRUPTIBLE` » suite à un appel système blocant est réveillé par le noyau (lorsqu'il reçoit un signal), une fois le *handler* correspondant terminé, le système positionne ce processus dans l'état

« TASK_RUNNING », et il positionne la variable errno à EINTR. Nous verrons dans les prochains chapitres comment faire pour que le processus ne reprenne pas son exécution normale, et attende bien la fin de son appel système.

Remarque importante : un processus fils n'hérite pas des signaux pendants de son père. De plus, en cas de recouvrement du code hérité du père, les gestionnaires par défaut sont réinstallés pour tous les signaux du fils.

9.3 Les routines systèmes liées aux signaux

9.3.1 Envoyer un signal à un processus

Comme nous l'avons déjà vu, l'envoi d'un signal se fait à l'aide de la fonction kill(). Un « man 2 kill » nous donne :
<<

```
SYNOPSIS :
       #include <sys/types.h>
       #include <signal.h>
       int kill(pid_t pid, int sig);

DESCRIPTION
   kill() peut être utilisé pour envoyer n'importe quel signal
   à n'importe quel processus ou groupe de processus.

   • Si pid est positif, le signal sig est envoyé au
     processus pid.
   • Si pid vaut zéro, alors le signal sig est envoyé à tous
     les processus appartenant au même groupe que le
     processus appelant.
   • Si pid vaut -1, alors le signal sig est envoyé à tous
     les processus sauf le premier (init) dans l'ordre
     décroissant des numéros dans la table des processus (par
     ex: shutdown envoie le signal SIGTERM à tous les
     processus).
   • Si pid est inférieur à -1, alors le signal sig est
     envoyé à tous les processus du groupe -pid.

VALEUR RENVOYEE
     En cas de réussite 0 est renvoyé. En cas d'échec, -1 est
     renvoyé et errno contient le code d'erreur.

ERREURS
     EINVAL : Numéro de signal invalide.
     ESRCH  : Le processus ou le groupe de processus n'existe
     pas. Remarquez qu'un processus existant peut être un
     zombie, c'est à dire qu'il s'est déjà terminé mais que
     son père n'a pas encore lu sa valeur de retour avec
     wait().
     EPERM  : Le processus appelant n'a pas l'autorisation
     d'envoyer un signal à l'un des processus concernés. Pour
     qu'un processus ait le droit d'envoyer un signal à un
     autre processus pid il doit avoir des privilèges de
     Super-Utilisateur, ou avoir un UID réel ou effectif égal
     à l'ID réel ou sauvegardé du processus récepteur.
```

```
      REMARQUE
           On ne peut pas envoyer de signal au processus numéro 1
           (init), qui ne dispose pas de routine de gestion de
           signaux. Ceci évite que le système soit arrêté
           accidentellement.
>>
```

9.3.2 Bloquer les signaux

Pour bloquer les signaux, nous devons manipuler des objets de type « `sigset_t` », qui représente un « *ensemble de signaux* ».

Pour manipuler les variables de type `sigset_t`, nous avons à notre disposition les fonctions suivantes (Cf. « `man 3 sigsetops` ») :

```
<<
      SYNOPSIS
             #include <sys/types.h>
             #include <signal.h>

             int sigemptyset (sigset_t *set);
             int sigfillset (sigset_t *set);
             int sigaddset (sigset_t *set, int signum);
             int sigdelset (sigset_t *set, int signum);
             int sigismember (const sigset_t *set, int signum);

      DESCRIPTION
         Les  fonctions sigsetops(3) permettent la manipulation des
         ensembles de signaux POSIX.

          • sigemptyset() vide l'ensemble de  signaux set.
          • sigfillset() remplit totalement l'ensemble de signaux
            set en incluant tous les signaux.
          • sigaddset() (respectivement sigdelset()) ajoute
            (respectivement supprime) le signal signum de l'ensemble
            set.
          • sigismember()  teste  si  le  signal  signum  est
            membre de l'ensemble set.

      VALEUR RENVOYÉE
           sigemptyset(), sigfullset(), sigaddset() et sigdelset()
           renvoient 0 s'ils réussissent, et -1 s'ils échouent.

           sigismember()  renvoie  1  si  le  signal  signum  est
           dans l'ensemble set, 0 si signum n'y est  pas,  et  -1
           en  cas d'erreur.

      ERREURS
           EINVAL : sig n'est pas un signal valide.
>>
```

La fonction `sigprocmask()` permet à un processus de bloquer (ou de débloquer) un ensemble de signaux, à l'exception des signaux SIGKILL et SIGCONT. En pratique, cette fonction manipule le champ « `blocked` » du PCB. Extrait du manuel de cette fonction :

```
<<
      SYNOPSIS
           #include <signal.h>
           int sigprocmask(int how, const sigset_t *set, \
                           sigset_t *oldset);
```

DESCRIPTION

sigprocmask() est utilisé pour modifier le masque de signaux, l'ensemble des signaux actuellement bloqués. Le comportement de cet appel est dépendant de la valeur de how, avec les conventions suivantes :

- **SIG_BLOCK :** L'ensemble des signaux bloqués est l'union de l'ensemble actuel et de l'argument **set**.
- **SIG_UNBLOCK :** Les signaux dans l'ensemble **set** sont supprimés de la liste des signaux bloqués. Il est possible de débloquer un signal non bloqué.
- **SIG_SETMASK :** L'ensemble des signaux bloqués est égal à l'argument **set**.

Si **oldset** est non nul, la valeur précédente du masque de signaux est stockée dans **oldset**.

Si **set** est NULL, le masque des signaux n'est pas modifié (c'est-à-dire que **how** est ignoré), mais la valeur courante du masque de signaux est néanmoins renvoyée dans **oldset** (s'il n'est pas NULL).

L'utilisation de sigprocmask() dans un processus multithread n'est pas spécifiée ; voir pthread_sigmask(3).

VALEUR RENVOYÉE

sigprocmask() renvoie 0 s'il réussit et -1 s'il échoue.

ERREURS

EINVAL : La valeur spécifiée dans **how** n'est pas valide.

NOTES

Il n'est pas possible de bloquer **SIGKILL** ou **SIGSTOP**. Les tentatives pour le faire sont silencieusement ignorées.

Si **SIGBUS**, **SIGFPE**, **SIGILL** ou **SIGSEGV** sont engendrés pendant qu'ils sont bloqués, le résultat n'est pas défini à moins que le signal ait été généré par kill(2), sigqueue(2) ou raise(3).

Voir sigsetops(3) pour les détails sur la manipulation des jeux de signaux.

>>

La primitive « sigsuspend(const sigset_t *ens); » permet de façon atomique de modifier le masque des signaux et de se bloquer en attente. Une fois un signal non bloqué délivré, la primitive se termine en restaurant le masque antérieur :

<<

SYNOPSIS

```
#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

DESCRIPTION

L'appel sigsuspend() remplace temporairement le masque de signaux bloqués par celui fourni dans **mask**, puis endort le processus jusqu'à arrivée d'un

signal qui déclenche un gestionnaire de signal ou termine le processus.

Si le signal termine le processus, sigsuspend() ne retourne pas à l'appelant. Si le signal est intercepté, sigsuspend() retourne après l'exécution du gestionnaire, et le masque de signaux bloqués est restauré à sa valeur précédant l'appel à sigsuspend().

Il n'est possible de bloquer ni SIGKILL ni SIGSTOP. Ceci est imposé silencieusement par le système.

VALEUR RENVOYÉE
sigsuspend() renvoie toujours -1, et **errno** est normalement positionné à **EINTR**.

ERREURS
EFAULT : **mask** pointe en-dehors de l'espace d'adressage accessible.
EINTR : L'appel a été interrompu par un signal.

NOTES
En général, sigsuspend() est utilisé conjointement avec sigprocmask() pour empêcher l'arrivée d'un signal pendant l'exécution d'une section de code critique. L'appelant commence par bloquer les signaux avec sigprocmask(). Après la fin de la section critique, l'appelant attend les signaux avec sigsuspend() utilisé avec le masque renvoyé par sigprocmask() (dans l'argument **oldset**).

>>

Enfin, la primitive « int sigpending(sigset_t *ens); » permet de connaître l'ensemble des signaux pendants d'un processus.

<<

SYNOPSIS
```
#include <signal.h>
int sigpending(sigset_t *set);
```

DESCRIPTION
sigpending() renvoie l'ensemble des signaux en attente de livraison au thread appelant (c'est-à-dire les signaux qui se sont déclenchés en étant bloqués). Le masque des signaux en attente en renvoyé dans **set**.

VALEUR RENVOYÉE
sigpending() renvoie 0 s'il réussit et -1 s'il échoue.

ERREURS
EFAULT : **set** pointe sur de la mémoire qui n'est pas une partie valide dans l'espace adressable du processus.

>>

Exemple classique de l'utilisation de ces fonctions : empêcher le CTRL+C (break) pendant l'exécution d'une phase sensible de code :

<<
```
/* Exemple de blocage du CTRL+C */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

int main()
{
     sigset_t   ens, ens1;
     /* bla bal [...] autres déclarations */

     printf("Blocage du CTRL+C\n");
     sigemptyset(&ens);
     sigaddset(&ens, SIGINT);
     sigprocmask(SIG_SETMASK, &ens, 0);
     printf("Le CTRL+C est bloqué\n");

     printf("Début du travail...\n");
     /* bla bla bla gros calcul, etc. */
     /* Par exemple : */
     sleep(10); /* On attend 10 secondes */
     printf("Fin du travail\n");

     sigpending(&ens1);
     if (sigismember(&ens1, SIGINT))
          printf("CTRL+C a été tapé au moins 1 fois.\n");

     printf("Déblocage du CTRL+C\n");
     sigemptyset(&ens);
     sigprocmask(SIG_SETMASK, &ens, 0);
     printf("Le CTRL+C est débloqué\n");
     return(0) ;
}
```
>>

9.3.3 Attacher un *handler* à un signal

Deux primitives permettent d'attacher un gestionnaire de traitement d'un signal à un signal. Ils s'agit des fonctions `signal()` et `sigaction()`. La première est plus simple d'utilisation, mais risque de poser des soucis de compatibilité avec d'autres Unix. La seconde est plus complexe d'appréhension, mais plus universelle. Voici leurs documentations :

<<
```
SYNOPSIS
     #include <signal.h>
     void (*signal(int signum, void (*handler)(int)))(int);

DESCRIPTION
     L'appel-système signal() installe un nouveau
     gestionnaire pour le signal numéro signum. Le
     gestionnaire de signal est handler qui peut être soit
     une fonction spécifique de l'utilisateur, soit l'une des
     constantes SIG_IGN ou SIG_DFL.

     Lors de l'arrivée d'un signal correspondant au numéro
     signum, les évènements suivants se produisent :
```

- Si le gestionnaire correspondant est configuré avec **SIG_IGN**, le signal est ignoré.
- Si le gestionnaire vaut **SIG_DFL**, l'action par défaut pour le signal est entreprise, comme décrit dans signal(7) .
- Finalement, si le gestionnaire est dirigé vers une fonction **handler**(), alors tout d'abord, le gestionnaire est re-configuré à **SIG_DFL**, ou le signal est bloqué, puis **handler**() est appelé avec l'argument **signum**.

Utiliser d'une fonction comme gestionnaire de signal est appelé "intercepter - ou capturer - le signal". Les signaux **SIGKILL** et **SIGSTOP** ne peuvent ni être ignoré, ni être interceptés.

VALEUR RENVOYEE

signal() renvoie la valeur précédente du gestionnaire de signaux, ou **SIG_ERR** en cas d'erreur.

PORTABILITE

La fonction signal() originale d'Unix réinitialisait le gestionnaire à **SIG_DFL**, comme c'est le cas sous Système V. Linux agissait ainsi avec les bibliothèques libc4 et libc5. Au contraire, BSD ne réinitialise pas le gestionnaire, mais bloque les éventuelles nouvelles occurrences du signal durant l'appel de la fonction. La bibliothèque GlibC 2 suit ce comportement.

Néanmoins, si l'on inclut sur un système sous libc5 <bsd/signal.h> à la place de <signal.h>, alors, signal est redéfini en tant que __bsd_signal et disposera alors de la sémantique BSD. C'est peu recommandé.

Sur un système fonctionnant avec la GlibC 2, si on définit la constante _XOPEN_SOURCE ou si on utilise la fonction sysv_signal(), on obtient le comportement habituel. C'est peu recommandé.

La modification de la sémantique de l'appel en utilisant une constante symbolique ou un fichier d'en-tête spécial n'est pas une bonne idée. Il vaut mieux éviter d'utiliser signal() complètement, et utiliser plutôt sigaction(2).

NOTES

Comme spécifié par la norme POSIX, le comportement d'un processus est indéfini après la réception d'un signal **SIGFPE**, **SIGILL**, ou **SIGSEGV** qui n'a pas été engendré par une fonction kill() ou raise().

La division entière par zéro a un résultat indéfini. Sur certaines architectures, elle déclenche un signal **SIGFPE**. Ignorer ce signal peut conduire à des boucles infinies. De même diviser l'entier le plus négatif par -1 peut déclencher **SIGFPE**.

La norme POSIX (B.3.3.1.3) désapprouve le positionnement de **SIGCHLD** à **SIG_IGN**. Les comportements BSD et SYSV diffèrent, faisant échouer sous Linux les logiciels BSD qui positionne l'action de **SIGCHK** à **SIG_IGN**.

SYNOPSIS

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, \
              struct sigaction *oldact);
```

DESCRIPTION

L'appel système sigaction() sert à modifier l'action effectuée par un processus à la réception d'un signal spécifique.

signum indique le signal concerné, à l'exception de **SIGKILL** et **SIGSTOP**.

Si **act** est non nul, la nouvelle action pour le signal **signum** est définie par **act**. Si **oldact** est non nul, l'ancienne action est sauvegardée dans **oldact**.

La structure sigaction est définie par quelque chose comme :

```
struct sigaction {
  void     (* sa_handler)   (int);
  void     (* sa_sigaction) (int, siginfo_t *, void *);
  sigset_t    sa_mask;
  int         sa_flags;
  void     (* sa_restorer)  (void);
}
```

Sur certaines architectures on emploie une *union* ; il ne faut donc pas utiliser ou remplir simultanément **sa_handler** et **sa_sigaction**.

L'élément **sa_restorer** est obsolète et ne doit pas être utilisé (POSIX ne mentionne pas de membre **sa_restorer**).

sa_handler indique l'action affectée au signal **signum**, et peut être **SIG_DFL** pour l'action par défaut, **SIG_IGN** pour ignorer le signal, ou un pointeur sur une fonction de gestion de signaux.

sa_mask fournit un masque de signaux à bloquer pendant l'exécution du gestionnaire. De plus, le signal ayant appelé le gestionnaire est bloqué à moins que les attributs **SA_NODEFER** ou **SA_NOMASK** soient précisés.

sa_flags spécifie un ensemble d'attributs qui modifient le comportement du gestionnaire de signaux. Il est formé par un OU binaire (|) entre les options suivantes :

- **SA_NOCLDSTOP :** Si signum vaut **SIGCHLD**, ne pas recevoir les signaux de notification d'arrêt d'un processus fils (quand le fils reçoit un signal **SIGSTOP**, **SIGTSTP**, **SIGTTIN** ou **SIGTTOU**).

- **SA_ONESHOT** ou **SA_RESETHAND** : Rétablir l'action à son comportement par défaut une fois que le gestionnaire a été appelé (C'est le comportement par défaut avec la fonction signal()).
- **SA_RESTART :** Fournir un comportement compatible avec la sémantique BSD en redémarrant automatiquement les appels systèmes lents interrompus par l'arrivée du signal.
- **SA_NOMASK** ou **SA_NODEFER** : Ne pas empêcher un signal d'être reçu depuis l'intérieur de son propre gestionnaire.
- **SA_SIGINFO :** Le gestionnaire de signal recevra trois arguments, et non plus un seul. Dans ce cas, il faut utiliser le membre **sa_sigaction** et non pas **sa_handler** (le champ **sa_sigaction** est apparu dans Linux 2.1.86.).

Le paramètre **siginfo_t** de la routine **sa_sigaction** est une structure contenant les éléments suivants :

```
siginfo_t {
  int     si_signo;       /* Numéro de signal         */
  int     si_errno;       /* Numéro d'erreur          */
  int     si_code;        /* Code du signal           */
  pid_t   si_pid;         /* PID de l'émetteur        */
  uid_t   si_uid;         /* UID réel de l'émetteur   */
  int     si_status;      /* Valeur de sortie         */
  clock_t si_utime;       /* Temps utilisateur écoulé */
  clock_t si_stime;       /* Temps système écoulé     */
  sigval_t si_value;      /* Valeur de signal         */
  int     si_int;         /* Signal Posix.1b          */
  void *  si_ptr;         /* Signal Posix.1b          */
  void *  si_addr;        /* Emplacement d'erreur     */
  int     si_band;        /* Band event               */
  int     si_fd;          /* Descripteur de fichier   */
}
```

Les champs **si_signo**, **si_errno** et **si_code** sont définis pour tous les signaux. Le reste de la structure peut être une *union*, et il ne faut donc tenir compte que des champs qui sont significatifs pour le signal reçu. L'appel-système kill(2) , les signaux Posix.1b et **SIGCHLD** remplissent les champs **si_pid** et **si_uid**. **SIGCHLD** remplit aussi **si_status**, **si_utime** et **si_stime**. **si_int** et **si_ptr** sont fournis par l'émetteur d'un signal Posix.1b. **SIGILL**, **SIGFPE**, **SIGSEGV** et **SIGBUS** remplissent **si_addr** avec l'adresse de l'erreur. **SIGPOLL** remplit **si_band** et **si_fd**.

si_code indique la raison pour laquelle le signal a été émis. Il s'agit d'une valeur, pas d'un masque de bits.

VALEUR RENVOYEE

sigaction, sigprocmask, sigpending et sigsuspend renvoient 0 s'ils réussissent, ou -1 s'ils échouent, auquel cas errno contient le code d'erreur.

```
      ERREURS
            EINVAL : Un signal invalide est indique. Ceci se produit
            également si l'on tente de modifier l'action associée à
            SIGKILL ou SIGSTOP.
            EFAULT : act, oldact, set ou oldset pointent en dehors
            de l'espace d'adressage accessible.
            EINTR  : L'appel système a été interrompu.
>>
```

9.3.4 Traiter les appels systèmes interrompus

Lorsqu'un processus placé dans l'état « TASK_INTERRUPTIBLE » suite à un appel système blocant est réveillé par le noyau lorsqu'il reçoit un signal, le handler correspondant s'exécute. Celui-ci terminé, le système positionne ce processus dans l'état « TASK_RUNNING », et il positionne la variable errno à EINTR. Or, il ne faut pas reprendre l'exécution d'un processus qui est bloqué après un appel système. Deux solutions permettent de résoudre ce problème :

- relancer « *manuellement* » l'exécution de l'appel système lorsque celui-ci échoue avec un code d'erreur errno valant EINTR. Par exemple :

```
do
{
      nb_lus=read(desc_fic, buffer, nb_a_lire);
} while ((nb_lus == -1)&&(errno == EINTR));
```

- autre solution, plus hestétique : utiliser la fonction « int siginterrupt(int signum, int interrupt); ». Cette fonction est appelée après l'installation du gestionnaire de signal associé au signal signum. Si le paramètre « interrupt » est nul, l'appel système interrompu par le signal « signum » sera relancé automatiquement. Sinon, si « interrupt » est non nul, le système reprend son fonctionnement par défaut (l'appel système retourne une erreur, et errno prend la valeur EINTR).

9.3.5 Attendre un signal

L'appel à la fonction « pause() » bloque un processus dans l'attente de la délivrance d'un signal :

```
<<
      SYNOPSIS
            #include <unistd.h>
            int pause(void);

      DESCRIPTION
            Un appel à pause endort le processus appelant jusqu'à ce
            qu'il reçoive un signal.

      VALEUR RENVOYEE
            pause() renvoie toujours -1, et errno est positionné à
            la valeur EINTR.
>>
```

9.3.6 Armer une temporisation

La primitive « alarm() » permet d'armer une temporisation. A la fin de cette temporisation, le signal SIGALRM est délivré au processus.

```
<<
```

```
SYNOPSIS
     #include <unistd.h>
     unsigned int alarm(unsigned int nb_sec);

DESCRIPTION
     alarm() programme une temporisation pour qu'elle envoie
     un signal SIGALRM au processus en cours dans nb_sec
     secondes. Si le signal n'est pas masqué ou intercepté,
     sa réception terminera le processus.

     Les programmations successives d'alarmes ne sont pas
     empilées, chaque appel de alarm annule l'éventuelle
     programmation précédente.

     Si nb_sec vaut zéro, aucune alarme n'est planifiée.

VALEUR RENVOYEE
     alarm renvoie le nombre de secondes qu'il restait de la
     programmation précédente (qui se retrouve annulée), ou
     zéro si aucune alarme n'avait été planifiée auparavant.

NOTES
     Les fonctions alarm() et setitimer() partagent la même
     temporisation. Aussi l'appel de l'une interfère avec
     l'utilisation de l'autre.

     sleep() peut être implémenté en utilisant SIGALRM, aussi
     il est déconseillé de mélanger les appels alarm() et
     sleep().

     Les délais dûs au multitâche peuvent, comme toujours,
     retarder le déclenchement d'une alarme d'une durée
     arbitraire.
>>
```

9.4 Les signaux temps réel

9.4.1 Présentation

Comme nous l'avons vu, en plus des signaux Unix classiques (qui répondent à la norme POSIX), Linux propose aussi un mécanisme de signaux « *temps réels* », qui respectent la norme POSIX 1b. Il est possible de programmer 31 signaux, de 33 à 63. Afin de rédiger du code le plus portable possible, plutôt que d'utiliser ces valeurs 33 à 63, mieux vaut utiliser les constantes `SIGRTMIN` et `SIGRTMAX`.

De façon générale, les propriétés des signaux temps-réel sont :

- empilement des occurrences de chaque signal. Comme nous l'avons vu précédemment, la réception d'un signal classique par un processus est gérée à l'aide d'un seul bit par signal (bit qui est mis à 1 si le signal a été reçu et à 0 sinon). Cette manière de faire ne permet pas de garder la mémoire le fait que plusieurs occurrences successives d'un même signal on eut lieu. Aussi, un signal reçu 2 fois ne sera délivré qu'une seule fois au processus. Linux associe à chaque signal temps réel une file d'attente qui lui permet au contraire de mémoriser l'ensemble de toutes les occurrences. Chaque occurrence présente dans la file d'attente donne lieu à une délivrance spécifique. Le système permet d'empiler jusqu'à 1 024 signaux ;

- lorsque plusieurs signaux temps réel doivent être délivrés à un processus, le noyau délivre toujours les signaux temps réel de plus petit numéro avant les signaux temps réel de plus grand numéro. Ceci permet d'attribuer un ordre de priorité entre les signaux temps réel ;
- le signal temps réel, contrairement au signal classique, peut transporter avec lui une petite quantité d'informations, délivrées au handler qui lui est attaché.

9.4.2 Envoyer un signal temps réel

L'envoi d'un signal temps réel peut être réalisé avec la fonction `kill()`. Cette façon de faire est déconseillée : elle ne permet pas de délivrer d'informations complémentaires au numéro du signal reçu.

Pour pouvoir délivrer des informations complémentaires au signal, il faut utiliser la primitive `sigqueue()`, dont le prototype est :

<<

```
SYNOPSIS
     #include <signal.h>
     int sigqueue(pid_t pid, int sig, \
                  const union sigval valeur);

DESCRIPTION
     sigqueue() envoie le signal sig au processus indiqué par
     pid. Les permissions requises pour l'émission du signal
     sont les mêmes que pour kill(2). Comme avec kill(2), le
     signal nul (0) peut servir à vérifier si un processus
     existe avec un PID donné.

     L'argument valeur sert à indiquer une donnée (soit un
     entier, soit un pointeur) qui accompagnera le signal et
     se présente avec le type suivant :
              union sigval {
                  int   sival_int;
                  void *sival_ptr;
              };
     Si le processus récepteur a installé un gestionnaire
     pour le  signal avec l'option  SA_SIGINFO lors du
     sigaction(2), il pourra obtenir la donnée dans le champ
     si_value de la structure siginfo_t passée en  second
     argument au gestionnaire. De plus, le champ si_code de
     la structure contiendra SI_QUEUE.

VALEUR RENVOYÉE
     L'appel sigqueue() renvoie 0 s'il réussit, indiquant que
     le  signal  a bien  été  mis dans la file du processus
     récepteur. Sinon il renvoie -1 et errno contient le code
     de l'erreur.

ERREURS
     EAGAIN : La limite du nombre de signaux en file  a  été
     atteinte (Voir signal(7) pour plus d'informations).
     EINVAL : sig est invalide.
     EPERM : Le  processus  n'a pas la permission d'envoyer
     un signal au processus visé. Pour les permissions
     nécessaires, voir kill(2).
     ESRCH : Il n'y a pas de processus correspondant au pid.
```

```
      NOTES
           Si l'appel consiste à envoyer un signal au processus qui
           l'a  invoqué, et  si  le  signal n'est pas bloqué par le
           thread appelant, et si aucun autre thread ne peut
           traiter le signal (soit en l'ayant débloqué, ou en
           faisant  une  attente avec sigwait(3)), alors au moins
           quelques signaux seront délivrés au thread appelant
           avant que la fonction ne revienne.
>>
```

9.4.3 Attacher un handler à un signal temps réel

Du fait de cette transmission d'informations complémentaires associées au signal temps réel, l'attachement et la définition d'un gestionnaire de signal s'effectuent différemment que dans le cadre d'un signal classique. Aussi, le gestionnaire doit avoir le type suivant :

```
void mon_gestionnaire_tr(int num_sig, \
                         struct siginfo *info, void *rien);
```

La structure de type `struct siginfo` contient notamment les champs suivants, conformes à la norme POSIX :

- `int si_signo` : le numéro du signal ;
- `int si_code` : information dépendant du signal. Pour les signaux temps réel, cette information indique l'origine du signal ;
- `int si_value.sival_int` correspond au champ `sival_int` dans l'appel système `sigqueue()` ;
- `void *si_value.sival_ptr` correspond au champ `sival_ptr` dans l'appel système `sigqueue()`.

Le premier paramètre (`num_sig`) correspond au numéro du signal reçu. Le troisième paramètre (`rien`) n'est pas normalisé par le standard POSIX et n'est pas utilisé.

L'attachement de ce gestionnaire de signal à un signal s'effectue en utilisant la primitive `sigaction()` décrite dans les précédents chapitres. Mais cette fois-ci, le gestionnaire n'est pas renseigné dans le champ `sa_handler` de la structure mais dans le champ `sa_sigaction` de celle-ci. En effet, comme le montre le détail de la structure qui suit, le premier champ de celle-ci est en fait construit comme une union du champ `sa_handler` et du champ `sa_sigaction` :

```
<<
      #include <signal.h>
      struct  sigaction
      {
        union
        {
          void    (*__sa_handler)(int);
          void    (*__sa_sigaction)(int, struct __siginfo *, \
                           void *);
        }        __sigaction_u; /* signal handler */
        int      sa_flags;      /* see signal options below */
        sigset_t sa_mask;       /* signal mask to apply */
      };
      #define sa_handler      __sigaction_u.__sa_handler
      #define sa_sigaction    __sigaction_u.__sa_sigaction
      int sigaction(int sig, const struct sigaction * restrict act,\
                    struct sigaction * restrict oact);
>>
```

Par ailleurs, le champ `sa_flags` doit prendre la valeur `SA_SIGINFO`.

9.4.4 Exécution du gestionnaire de signal

L'exécution d'un gestionnaire de signal telle qu'elle est décrite dans les précédents chapitres nécessite plusieurs commutations de contexte : une pour aller exécuter le gestionnaire dans le code utilisateur, puis une seconde pour revenir dans le noyau et achever la routine système prenant en compte les signaux.

Dans le cadre du temps réel, ces commutations de contexte peuvent être pénalisantes en terme de temps. Aussi, le système Linux offre deux primitives permettant à processus de simplement attendre un signal temps réel, sans que le noyau active le gestionnaire du signal. Ces mécanismes sont mis à disposition via les deux primitives `sigwaitinfo()` et `sigtimedwait()`.

Lorsqu'un processus effectue un appel à la primitive `sigwaitinfo()`, il reste bloqué jusqu'à ce qu'au moins un signal de l'ensemble fourni en paramètre lui ait été délivré. La primitive se termine simplement en retournant le numéro du signal reçu, et éventuellement les informations associées. Le processus peut alors invoquer le gestionnaire de signal associé au signal reçu comme un simple appel de procédure, ce qui est beaucoup moins coûteux en terme de commutations de contexte.

La fonction `sigwaitinfo()` constitue tout simplement une version temporisée de la fonction `sigtimedwait()` :

```
<<
      SYNOPSIS
            #include <signal.h>
            int sigwaitinfo(const sigset_t *set, siginfo_t *info);
            int sigtimedwait(const sigset_t *set, siginfo_t *info,\
                             const struct timespec timeout);

      DESCRIPTION
            sigwaitinfo() suspend l'exécution du processus appelant
            jusqu'à ce que l'un des signaux de l'ensemble set soit
            reçu (si l'un des signaux de l'ensemble set est déjà en
            attente pour le processus appelant, sigwaitinfo()
            revient immédiatement avec les informations concernant
            le signal).

            sigwaitinfo() retire le signal de la liste des signaux
            en attente pour le processus, et renvoie le numéro du
            signal en retour de fonction. Si l'argument info n'est
            pas NULL, alors il sera rempli avec une structure du
            type siginfo_t (voir sigaction(2)) contenant les
            informations concernant le signal.

            Les signaux reçus via sigwaitinfo() sont délivrés dans
            l'ordre habituel, voir signal(7) pour plus de détails.

            sigtimedwait() opère exactement commee sigwaitinfo()
            mais elle a un argument supplémentaire, timeout, qui
            permet de fixer une limite maximale au temps d'attente.
            Cet argument est du type :
              struct timespec {
                long    tv_sec;     /* secondes */
                long    tv_nsec;    /* nanosecondes */
              };
```

Si les deux champs de cette structure sont nuls, sigtimedwait() revient tout de suite, soit avec des informations sur un signal déjà en attente, soit avec une erreur si aucun signaux de l'ensemble set n'étaient disponibles.

VALEUR RENVOYÉE
S'ils réussissent sigwaitinfo() et sigtimedwait() renvoient un numéro de signal (une valeur supérieure à zéro). S'ils échouent, ils renvoient -1, et **errno** contient le code d'erreur.

ERREURS
EAGAIN : Aucun signal de l'ensemble set n'a été reçu avant expiration du délai **timeout** indiqué pour sigtimedwait().
EINVAL : **timeout** etait invalide.
EINTR : L'appel a été interrompu par un gestionaire de signal (un signal autre que ceux présents dans l'ensemble set).

NOTES
En utilisation habituelle, l'appelant bloque les signaux de l'ensemble **set** au préalable avec un appel sigprocmask() (afin que la délivrance des signaux ne se produise pas en dehors de l'appel sigwaitinfo() ou sigtimedwait()) et n'installe pas de gestionnaire pour ces signaux.

POSIX ne spécifie as le comportement si une valeur NULL est indiquée pour l'argument **timeout** de sigtimedwait() permettant par exemple d'avoir le même comportement que celui de sigwaitinfo(), ce qui est le cas sous Linux.
>>

9.4.5 Complément sur les signaux temps-réels

Pour compléter ce qui a été écrit précédemment concernant les signaux temps réels, voici un extrait de la documentation ad-hoc de Linux. Il est conseillé de lire attentivement les phrases **en gras** si vous souhaitez utiliser les signaux temps-réel :

<<

Linux supporte les signaux temps-réel tels qu'ils ont été définis à l'origine dans les extentions temps-réel Posix.4 (et inclus à présent dans Posix 1003.1-2001). Linux supporte 32 signaux temps-réel numéroté de 32 (SIGRTMIN) à 63 (SIGRTMAX). (Les applications **doivent toujours se référer aux signaux temps-réel en utilisant la notation SIGRTMIN+n**, car la plage des numéros des signaux varie suivant les Unix).

Contrairement aux signaux standards, les signaux temps-réel n'ont pas de signification prédéfinie : l'ensemble complet de ces signaux peut être utilisée à des fins spécifiques à l'application. Notez quand même que l'implémentation *LinuxThreads* utilise les trois premiers signaux temps-réel.

L'action par défaut pour un signal temps-réel non capturé est de terminer le processus récepteur.

Les signaux temps-réel se distinguent de leurs homologues classiques ainsi :

1. Plusieurs instances d'un signal temps-réel peuvent être empilées. Au contraire, si plusieurs instances d'un signal standard arrivent alors qu'il est bloqué, une seule instance sera mémorisée ;

2. Si le signal est envoyé en utilisant `sigqueue()`, il peut être accompagné d'une valeur (un entier ou un pointeur). Si le processus récepteur positionne un gestionnaire en utilisant l'attribut `SA_SIGACTION` de l'appel `sigaction()`, alors, il peut accéder à la valeur transmise dans le champ `si_value` de la structure `siginfo_t` passée en second argument au gestionnaire. De plus, les champs `si_pid` et `si_uid` de cette structure fournissent le PID et l'UID réel du processus émetteur.

3. Les signaux temps-réel **sont délivrés dans un ordre précis**. Les divers signaux temps-réel du même type sont délivrés dans l'ordre où ils ont été émis. Si différents signaux temps-réel sont envoyés au processus, ils sont délivrés en commençant par le signal de numéro le moins élevé (le signal de plus fort numéro est celui de priorité la plus faible).

Si des signaux standards et des signaux temps-réel sont simultanément en attente pour un processus, Posix ne précise pas d'ordre de délivrance. Linux, comme beaucoup d'autres implémentations, donne priorité aux signaux temps-réel dans ce cas.

D'après Posix, une implémentation doit permettre l'empilement d'au moins `_POSIX_SIGQUEUE_MAX` signaux pour un processus (*NDLR : à l'heure où ses lignes sont écrites, `_POSIX_SIGQUEUE_MAX` vaut 32*). Néanmoins, plutôt que de fixer une limite par processus, Linux impose une limite pour l'ensemble des signaux empilés sur le système pour tous les processus. Cette limite peut être consultée, et modifiée (avec les privilèges adéquats) grâce au fichier « `/proc/sys/kernel/rtsig-max` ». Un fichier associé, « `/proc/sys/kernel/rtsig-max` », indique combien de signaux temps-réel sont actuellement empilés.

>>

10 Communication centralisée inter-processus

Nous avons vu les mécanismes sous-jacents aux systèmes multiprogrammés (processus, threads), et la technique d'envoi de signal d'une tâche à une autre, pouvant avoir comme conséquence de dérouter le code du processus qui a reçu le signal vers une fonction particulière. Nous avons insisté sur le fait que les processus étaient protégés les uns des autres (ils ne partagent pas de variables, il n'ont pas le même espace mémoire alloué, etc.). Pourtant, les processus peuvent être amenés à communiquer. Nous allons voir maintenant comment ces processus peuvent communiquer entre eux (au sein d'une même machine) via :

- les ***tubes*** (ou *pipes* en anglais),
- les ***files de messages*** (***MSQ*** ou *messages queues* en anglais),
- et la ***mémoire partagée*** (ou *shared memory*).

Dans la littérature, certains de ces mécanismes introduits dans « Unix Système V » (*MSQ* et *mémoire partagée*), auquel il faut ajouter les *sémaphores* (Cf prochains chapitres) sont classés sous le terme générique d'***IPC* : Inter Processus Communication** (*communication inter processus*), ou *IPC système V*.

10.1 Les tubes (pipes)

Le système d'exploitation propose deux types de tubes :

- les tubes anonymes,
- et les tubes nommés (*named pipe*).

Une analogie peut être faite entre un tube entre deux processus et les systèmes de pneumatiques (transfert de flux de données). Un tube est mono directionnel : une fois qu'un émetteur envoie des information dans un tube, il ne peut plus y lire des données.

Dans les systèmes d'exploitation Unix/Linux, les tubes sont gérés au niveau du gestionnaire de fichier. L'ouverture d'un tube retourne des descripteurs de fichiers. La lecture ou l'écriture dans un tube ressemble alors à la lecture ou l'écriture dans un fichier classique. La routine d'ouverture d'un tube retourne un tableau de deux descripteurs de fichiers : un d'entre eux sera utilisé par le processus écrivain pour émettre les données dans le tube, et l'autre sera utilisé par le processus lecteur pour y lire les données.

Le concept de tube est indépendant du type de données qui sont transmises. Le tube fonctionne comme une file (FIFO) : les premières données écrites par un processus A seront les premières données lues par un processus B.

10.1.1 Les tubes anonymes

Le tube anonyme est géré par le système de gestion de fichiers, s'utilise comme un fichier, mais ne possède pas de nom. De ce fait, les tubes anonymes ne peuvent être manipulés que par les processus qui les créent, et sont invisibles des autres processus.

La création d'un tube anonyme se fait à l'aide de la fonction `pipe()` :

```
<<
SYNOPSIS
       #include <unistd.h>
       int pipe(int filedes[2]);
```

```
DESCRIPTION
       pipe()  crée  une  paire  de  descripteurs de fichiers,
       pointant sur un inoeud de tube, et les place dans un
       tableau  filedes.  filedes[0]  est utilisé pour la lecture,
       et filedes[1] pour l'écriture.

VALEUR RENVOYÉE
       En  cas  de  réussite,  zéro est renvoyé, sinon -1 est
       renvoyé et errno contient le code d'erreur.

ERREURS
       EFAULT : filedes est invalide.
       EMFILE : Trop de descripteurs de fichiers sont utilisés par
                le processus.
       ENFILE : La  limite  du nombre total de fichiers ouverts sur
                le système a été atteinte.
```

>>

La fermeture d'un tube se fait comme la fermeture d'un fichier, à l'aide de la fonction `close()` :

<<

```
SYNOPSIS
       #include <unistd.h>
       int close(int fd);

DESCRIPTION
       close() ferme le descripteur fd, de manière à ce qu'il ne
       référence plus aucun fichier, et puisse être réutilisé.

       Si fd est la dernière copie d'un descripteur de fichier
       donné, les ressources qui lui sont associées sont libérées.
       Par exemple tous les verrouillages sont supprimés et si le
       descripteur etait la dernière référence sur un fichier
       supprimé avec unlink(2), le fichier est effectivement
       effacé.

VALEUR RENVOYEE
       close renvoie 0 s'il réussit, ou -1 en cas d'échec, auquel
       cas errno contient le code d'erreur.

ERREURS
       EBADF : Le descripteur de fichier fd est invalide
```

>>

La lecture et l'écrite dans un tube se fait à l'aide des fonctions `read()` et `write()` :

<<

```
SYNOPSIS
       #include <unistd.h>
       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read()  lit  jusqu'à  count  octets depuis le descripteur de
       fichier fd dans le tampon pointé par buf.
       Si count vaut zéro, read() renvoie zéro et n'a pas d'autres
       effets.  Si count est supérieur à SSIZE_MAX, le résultat est
       indéfini.
```

VALEUR RENVOYÉE

read() renvoie -1 s'il échoue, auquel cas **errno** contient le code d'erreur, et la position de la tête de lecture est indéfinie. Sinon, read() renvoie le nombre d'octets lus (0 en fin de fichier), et avance la tête de lecture de ce nombre. Le fait que le nombre renvoyé soit plus petit que le nombre demandé n'est pas une erreur. Ceci se produit à la fin du fichier, ou si on lit depuis un tube ou un terminal, ou encore si read() a été interrompu par un signal.

ERREURS

EAGAIN : On utilise une lecture non bloquante (attribut O_NONBLOCK du descripteur de fichier) et aucune donnée n'était disponible.

EBADF : **fd** n'est pas un descripteur de fichier valide ou n'est pas ouvert en lecture.

EFAULT : **buf** pointe en dehors de l'espace d'adressage accessible.

EINTR : read() a été interrompu par un signal avant d'avoir eu le temps de lire quoi que ce soit.

EINVAL : Le descripteur **fd** correspond à un objet sur lequel il est impossible de lire. Ou bien le fichier a été ouvert avec le drapeau O_DIRECT, et l'adresse de **buf**, la valeur de count ou la position actuelle de la tête de lecture ne sont pas alignées correctement.

EIO : Erreur d'entrée-sortie. Ceci arrive si un processus est dans un groupe en arrière-plan et tente de lire depuis le terminal. Il reçoit un signal SIGTTIN mais il l'ignore ou le bloque. Ceci se produit également si une erreur d'entrée-sortie bas-niveau s'est produite pendant la lecture d'un disque ou d'une bande.

EISDIR : fd est un répertoire.

D'autres erreurs peuvent se produire, suivant le type d'objet associé à **fd**. POSIX permet à un read() interrompu par un signal de renvoyer soit le nombre d'octets lus à ce point, soit -1, et de placer errno à EINTR.

SYNOPSIS

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

DESCRIPTION

write() écrit jusqu'à **count** octets dans le fichier associé au descripteur **fd** depuis le tampon pointé par **buf**. La norme POSIX réclame qu'une lecture avec read() effectuée après le retour d'une écriture avec write(), renvoie les nouvelles données. Notez que tous les systèmes de fichiers ne sont pas compatibles avec POSIX.

VALEUR RENVOYÉE

write() renvoie le nombre d'octets écrits (0 signifiant aucune écriture), ou -1 s'il échoue, auquel cas **errno** contient le code d'erreur. Si **count** vaut zéro, et si le

descripteur est associé à un fichier normal, 0 sera renvoyé sans effets de bord. Pour un fichier spécial, les résultats ne sont pas portables.

ERREURS

EAGAIN : L'écriture est non-bloquante (attribut O_NONBLOCK du descripteur), et l'opération devrait bloquer.

EBADF : **fd** n'est pas un descripteur de fichier valide, ou n'est pas ouvert en écriture.

EFAULT : **buf** pointe en dehors de l'espace d'adressage accessible.

EFBIG : Tentative d'écrire sur un fichier dont la taille dépasse un maximum dépendant de l'implémentation ou du processus, ou d'écrire à une position qui dépasse le maximum autorisé.

EINTR : L'appel système a été interrompu par un signal avant d'avoir pu écrire quoi que ce soit.

EINVAL : **fd** correspond à un objet sur lequel il est impossible d'écrire. Ou bien le fichier a été ouvert avec l'attribut O_DIRECT, et soit l'adresse de buf, soit la valeur de count, soit la position actuelle dans le fichier ne sont pas alignées correctement.

EIO : Une erreur d'entrée-sortie bas niveau s'est produite durant la modification de l'inoeud.

ENOSPC : Le périphérique correspondant à **fd** n'a plus de place disponible.

EPIPE : **fd** est connecté à un tube (pipe) ou une socket dont l'autre extrémité est fermée. Quand ceci se produit, le processus écrivain reçoit un signal SIGPIPE. Ainsi la valeur de retour de write n'est vue que si le programme intercepte, bloque ou ignore ce signal.

D'autres erreurs peuvent se produire suivant le type d'objet associé à fd.

>>

Exemple d'utilisation d'un tube anonyme :

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

int main(int argc, char *argv[])
{
    int   pfd[2];
    pid_t cpid;
    char  buf;
    if (argc != 2)
    {
        fprintf(stderr, "Vous devez fournir un paramètre\n");
        exit(-1) ;
    }
```

```
    if (pipe(pfd) == -1)
    {
        fprintf(stderr, "Erreur avec pipe()\n");
        exit(-1);
    }
    cpid = fork();
    if (cpid == -1)
    {
        fprintf(stderr, "Erreur avec fork()\n");
        exit(-1);
    }
    if (cpid == 0)
    {   /* Le fils lit dans le tube */
        /* Fermeture du descripteur en écriture inutilisé : */
        close(pfd[1]);
        while (read(pfd[0], &buf, 1) > 0)
        write(STDOUT_FILENO, &buf, 1);
        write(STDOUT_FILENO, "\n", 1);
        close(pfd[0]);
        exit(0);
    }
    else
    {   /* Le père écrit argv[1] dans le tube */
        /* Fermeture du descripteur en lecture inutilisé */
        close(pfd[0]);
        write(pfd[1], argv[1], strlen(argv[1]));
        close(pfd[1]);          /* Le lecteur verra EOF */
        wait(NULL);             /* Attente du fils */
        exit(EXIT_SUCCESS);
    }
    return(0);
}
```

>>

Les pipes anonymes sont utilisé au niveau de l'interpréteur de commandes (le shell) avec la syntaxe « | » (barre verticale, appelée « *pipe* », faite avec AltGr+6 sur un clavier AZERTY).

Pour comprendre le mécanisme du pipe du shell Unix/Linux, il faut savoir que chaque programme lancé depuis le shell possède trois descripteurs ouverts par défaut :

- ***stdin :*** c'est le descripteur de fichier d'entrée standard. Sauf action explicite, il s'agit du clavier. Dans ce cas, une fin de fichier peut être envoyée à l'aide de la combinaison de touches CTRL+Z (code ASCII 26) ;
- ***stdout :*** c'est le descripteur de fichier de sortie standard. Sauf action explicite, écrire sur ce descripteur envoie le texte à l'écran. Par exemple, c'est sur ce descripteur que sont envoyées les chaînes de caractères passées à la fonction `printf()` ;
- ***stderr :*** c'est le descripteur de fichier où sont envoyés les messages d'erreur (par exemple, ceux affichés par `perror()`).

Par convention, *stdin* est le premier fichier ouvert. Il a donc comme numéro de descripteur 0. Ensuite, *stdout* vaut 1, et *stderr* vaut 2. A noter que pour éviter tout problème si un jour cette convention venait à changer, et pour écrire du code portable, mieux vaut plutôt utiliser les constantes `STDIN_FILENO`, `STDOUT_FILENO`, et `STDERR_FILENO`.

Sous *shell*, rediriger *stdin* se fait avec le symbole « < ». Rediriger *stdout* se fait avec le symbole « > ». Rediriger *stderr* se fait avec la suite de caractères « 2>&1 » (sans espace, sinon, le symbole « & » sera interprété autrement par le shell).

Exemple : la commande shell « wc –l » (sans autre paramètre) compte le nombre de lignes dans *stdin*. La commande « ls » liste les fichiers présents dans le répertoire courant. Ainsi, la commande :

```
ls | wc -l
```

va rediriger *stdout* de ls pour et fournir ce flux en entrée à *stdin* de wc. Le résultat indiquera le nombre de fichiers contenus dans le répertoire courant.

Si nous voulions simuler ce même mécanisme dans un de nos programmes, il est nécessaire d'utiliser la fonction dup(), qui duplique un descripteur :

```
<<
SYNOPSIS
       #include <unistd.h>
       int dup(int oldfd);

DESCRIPTION
       dup() crée une copie du descripteur de fichier oldfd.
       Après un appel réussi à dup(), l'ancien et le nouveau
       descripteurs peuvent être utilisés de manière
       interchangeable. Ils référencent la même description
       de fichier ouvert (voir open(2)) et ainsi partagent les
       pointeurs de position et les drapeaux. Par exemple,
       si le pointeur de position est modifié en utilisant
       lseek(2) sur l'un des descripteurs, la position est
       également changée pour l'autre.

       Les deux descripteurs ne partagent toutefois pas
       l'attribut close_on_exec. L'attribut close_on_exec
       (FD_CLOEXEC ; voir fcntl(2)) de la copie est désactivé,
       ce qui signifie qu'il ne sera pas fermé lors d'un exec().

       dup() utilise le plus petit numéro inutilisé pour le
       nouveau descripteur.

VALEUR RENVOYÉE
       dup()renvoit le nouveau descripteur, ou -1 s'il échoue,
       auquel cas errno contient le code d'erreur.

ERREURS
       EBADF  : oldfd n'est pas un descripteur valide.
       EMFILE : Le processus dispose déjà du nombre maximum de
                descripteurs de fichiers autorisés simultanément,
                et tente d'en ouvrir un nouveau.

>>
```

Ce qui est important, c'est que dup() utilise le plus petit numéro disponible. Ainsi, si nous fermons le fichier d'entrée standard *stdin* (avec un « close(STDIN_FILENO) »), puis un « dup(descripteur_de_pipe_en_lecture) », le numéro de descripteur retrourné par dup() a toutes les chances d'être 0 (le numéro de descripteur de *stdin*).

Un bon exercice : selon ce principe, écrire un programme C qui ouvre un pipe anonyme, puis lance un fils avec fork(). Le père ferme *stdout*, puis lui ré-attribue grâce à la fonction dump() le descripteur en écriture du pipe, et lance (avec la fonction execlp()) une commande « ls ». De son coté, le fils ferme *stdin*, puis attribue (à l'aide de dump()) le descripteur en écriture du pipe à *stdin*, avant de lancer la commande « wc –l » avec la fonction execlp().

10.1.2 Les tubes nommés

Tout comme les tubes anonymes, les tubes nommés sont mono directionnels, et également gérés par le système de gestion de fichiers. Mais cette fois ci, il y a correspondance entre le tube et un vrai fichier (identifié avec son nom). De ce fait, ils sont accessibles par n'importe quel processus connaissant ce nom et disposant des droits d'accès ad hoc. Le tube nommé permet donc à des processus sans liens de parenté de communiquer selon un mode d'échange par flux.

Les fichiers de type « *tubes nommés* » ne sont pas des fichiers classiques de type « *conteneur*s » où seraient stockés des octets. En faisant un « `ls -l` » sous *shell*, les tubes nommés sont caractérisés par le type « p ». Ce sont des fichiers constitués d'un seul « *i-noeud* » (*i-node* en anglais : il s'agit de la structure physique qui permet de construire des fichiers ; nous verrons tout ça dans un prochain chapitre sur les fichiers), auquel n'est associé aucun bloc de données. Tout comme pour les tubes anonymes, les données contenues dans les tubes sont placées dans un tampon, constitué par une seule case de la mémoire centrale.

La création d'un tube nommé se fait à l'aide de la commande *shell* :

```
mkfifo nom_pipe
```

Cette commande fait appel à la fonction système (que vous pouvez vous aussi utiliser dans vos programmes) : `mkfifo()` :

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

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

DESCRIPTION
       La  fonction  mkfifo()  crée  un  fichier  spécial FIFO
       (tube nommé) à l'emplacement pathname. mode indique les
       permissions d'accès. Ces  permissions  sont  modifiées par
       la valeur d'umask du processus : les permissions d'accès
       effectivement adoptées sont (mode & ~umask).

       Une  fois  qu'un  fichier  FIFO est créé, n'importe quel
       processus peut l'ouvrir en lecture ou écriture, comme tout
       fichier ordinaire. En fait, il  faut  ouvrir  les  deux
       extrémités  simultanément avant de pouvoir effectuer une
       opération d'écriture ou de lecture. L'ouverture d'un FIFO
       en lecture est généralement bloquante, jusqu'à ce qu'un
       autre processus ouvre le même FIFO en écriture, et
       inversement. Voir  fifo(7)  pour  la gestion non bloquante
       d'une FIFO.

VALEUR RENVOYÉE
       La  valeur  renvoyée  par mkfifo() est 0 si elle réussit, ou
       -1 si elle échoue, auquel cas errno contient le code erreur.

ERREURS
       EACCES       : L'un des répertoires dans pathname ne permet
                      pas  la  recherche (exécution).
       EEXIST       : pathname existe déjà.
       ENAMETOOLONG : Soit  la  longueur totale de pathname est
                      supérieure à PATH_MAX, soit un élément de
                      pathname  a  une  longueur  plus  grande  que
```

NAME_MAX. Sur les systèmes GNU il n'y a pas de limite absolue à la longueur du nom d'un fichier, mais certains autres systèmes en ont une.

ENOENT : L'un des répertoire de **pathname** n'existe pas, ou est un lien symbolique pointant nulle part.

ENOSPC : Le répertoire, ou le système de fichiers, n'a pas assez de place pour un nouveau fichier.

ENOTDIR : Un élément de **pathname** n'est pas un répertoire.

EROFS : **pathname** est sur un système de fichiers en lecture seule.

>>

Ensuite, l'utilisation d'un tube nommé ressemble à l'utilisation d'un fichier. L'ouverture se fait à l'aide de la primitive « open( ) » :

<<

SYNOPSIS

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

DESCRIPTION

Étant donné le chemin **pathname** d'un fichier, open() renvoie un descripteur de fichier (petit entier positif ou nul) qui pourra ensuite être utilisé dans d'autres appels système (read(2), write(2), lseek(2), fcntl(2), etc.). Le descripteur de fichier renvoyé par un appel réussi sera le plus petit descripteur de fichier non actuellement ouvert par le processus.

Le nouveau descripteur de fichier est configuré pour rester ouvert après un appel à execve(2) (son attribut **FD_CLOEXEC** décrit dans fcntl(2) est initialement désactivé). La position dans le fichier est fixée au début du fichier (voir lseek(2)).

Un appel à open() crée une nouvelle description de fichier ouvert, une entrée dans la table de fichiers ouverts du système. Cette entrée enregistre la position dans le fichier et les attributs d'état du fichier (modifiables par l'opération **F_SETFL** de fcntl()). Un descripteur de fichier est une référence à l'une de ces entrées ; cette référence n'est pas modifiée si **pathname** est ensuite supprimé ou modifié pour correspondre à un autre fichier. La nouvelle description de fichier ouvert n'est initialement partagée avec aucun autre processus, mais ce partage peut apparaître après un fork(2).

Le paramètre **flags** est l'un des éléments **O_RDONLY**, **O_WRONLY** ou **O_RDWR** qui réclament respectivement l'ouverture du

fichier en lecture seule, écriture seule, ou lecture-écriture.

De plus, zéro ou plus d'attributs de création de fichier et d'attributs d'état de fichier peuvent être spécifiés dans **flags** avec un OU binaire. Les attributs de création de fichier sont **O_CREAT**, **O_EXCL**, **O_NOCTTY** et **O_TRUNC**. Les attributs d'état de fichier sont tous les autres attributs listés ci-dessous. La distinction entre ces deux groupes est que les attributs d'état de fichier peuvent être lus et (dans certains cas) modifiés avec fcntl(2). La liste complète des attributs de création et d'état de fichier est la suivante :

O_APPEND :
Le fichier est ouvert en mode « ajout ». Initialement, et avant chaque write(), la tête de lecture/écriture est placée à la fin du fichier comme avec lseek(). Il y a un risque d'endommager le fichier lorsque **O_APPEND** est utilisé, sur un système de fichiers NFS, si plusieurs processus tentent d'ajouter des données simultanément au même fichier. Ceci est dû au fait que NFS ne supporte pas l'opération d'ajout de données dans un fichier, aussi le noyau du client est obligé de la simuler, avec un risque de concurrence des tâches.

O_ASYNC :
Déclencher un signal (SIGIO par défaut, mais peut être changé via fcntl(2)) lorsque la lecture ou l'écriture deviennent possibles sur ce descripteur. Ceci n'est possible que pour les terminaux, pseudo-terminaux, sockets et (depuis Linux 2.6) tubes et FIFO. Voir fcntl(2) pour plus de détails.

O_CREAT :
Créer le fichier s'il n'existe pas. Le possesseur (UID) du fichier est renseigné avec l'UID effectif du processus. Le groupe propriétaire (GID) du fichier est le GID effectif du processus ou le GID du répertoire parent (ceci dépend du système de fichiers, des options de montage, du mode du répertoire parent, etc.) Voir par exemple les options de montage bsdgroups et sysv groups du système de fichiers ext2, décrites dans la page mount(8)).

O_DIRECT :
Essayer de minimiser les effets du cache d'entrée-sortie sur ce fichier. Ceci dégradera en général les performances, mais est utilisé dans des situations spéciales, lorsque les applications ont leur propres caches. Les entrées-sorties dans le fichier se font directement depuis l'espace utilisateur, elles sont synchrones (à la fin de read(2) ou write(2), les données ont obligatoirement été transférées). Sous Linux 2.4, la taille des

transferts, l'alignement du tampon et la position dans le fichier doivent être des multiples de la taille de blocs logiques du système de fichiers. Sous Linux 2.6, un alignement sur des multiples de 512 octets est suffisant.

Une interface à la sémantique similaire (mais obsolète) pour les périphériques de type bloc est décrite à la page raw(8).

O_DIRECTORY :

Si **pathname** n'est pas un répertoire, l'ouverture échoue. Cet attribut est spécifique à Linux et fut ajouté dans la version 2.1.126 du noyau, pour éviter des problèmes de dysfonctionnement si opendir(3) est invoqué sur une FIFO ou un périphérique de bande. Cet attribut ne devrait jamais être utilisé ailleurs que dans l'implémentation de opendir.

O_EXCL :

En conjonction avec **O_CREAT**, déclenchera une erreur si le fichier existe, et open() échouera. On considère qu'un lien symbolique existe, quel que soit l'endroit où il pointe. **O_EXCL** ne fonctionne pas sur les systèmes de fichiers NFS. Les programmes qui ont besoin de cette fonctionnalité pour verrouiller des tâches risquent de rencontrer une concurrence critique (race condition). La solution consiste à créer un fichier unique sur le même système de fichiers (par exemple avec le PID et le nom de l'hôte), utiliser link(2) pour créer un lien sur un fichier de verrouillage. Si link() renvoie 0, le verrouillage est réussi. Sinon, utiliser stat(2) sur ce fichier unique pour vérifier si le nombre de liens a augmenté jusqu'à 2, auquel cas le verrouillage est également réussi.

O_LARGEFILE :

(LFS) (Ndt : Large Files System) Autoriser l'ouverture des fichiers dont la taille ne peut pas être représentée avec un off_t (mais qui peut être représentée avec un off64_t).

O_NOATIME :

(Depuis Linux 2.6.8) Ne pas mettre à jour l'heure de dernier accès au fichier (champ st_atime de l'i-noeud) quand le fichier est lu avec read(2). Ce attribut est seulement conçu pour les programmes d'indexation et d'archivage, pour lesquels il peut réduire significativement l'activité du disque. L'attribut peut ne pas être effectif sur tous les systèmes de fichiers. Par exemple, avec NFS, l'heure d'accès est mise à jour par le serveur.

O_NOCTTY :

Si **pathname** correspond à un périphérique de terminal - voir tty(4) -, il ne deviendra pas le terminal contrôlant le processus même si celui-ci n'est attaché à aucun autre terminal.

O_NOFOLLOW :
Si **pathname** est un lien symbolique, l'ouverture échoue. Ceci est une extension FreeBSD, qui fut ajoutée à Linux dans la version 2.1.126. Les liens symboliques se trouvant dans le chemin d'accès proprement dit seront suivis normalement.

O_NONBLOCK ou **O_NDELAY :**
Le fichier est ouvert en mode « non-bloquant ». Ni la fonction open() ni aucune autre opération ultérieure sur ce fichier ne laissera le processus appelant en attente. Pour la manipulation des FIFO (tubes nommés), voir également fifo(7). Pour une explication de l'effet de **O_NONBLOCK** en conjonction avec les verrouillages impératifs et les bauxs de fichiers, voir fcntl(2).

O_SYNC :
Le fichier est ouvert en écriture synchronisée. Chaque appel à write() sur le fichier bloquera le processus appelant jusqu'à ce que les données aient été écrites physiquement sur le support matériel (voir la section RESTRICTIONS plus bas).

O_TRUNC :
Si le fichier existe, est un fichier régulier, et est ouvert en écriture (**O_RDWR** ou **O_WRONLY**), il sera tronqué à une longueur nulle. Si le fichier est une FIFO ou un périphérique terminal, l'attribut **O_TRUNC** est ignoré. Sinon, le comportement de **O_TRUNC** n'est pas précisé. Sur de nombreuses versions de Linux, il sera ignoré ; sur d'autres versions il déclenchera une erreur).

Certains de ces attributs optionnels peuvent être modifiés par la suite avec la fonction fcntl().

L'argument **mode** indique les permissions à utiliser si un nouveau fichier est créé. Cette valeur est modifiée par le umask du processus : la véritable valeur utilisée est (**mode** & ~umask). Notez que ce mode ne s'applique qu'aux accès ultérieurs au fichier nouvellement créé. L'appel open() qui crée un fichier dont le mode est en lecture seule fournira quand même un descripteur de fichier en lecture et écriture.

Les constantes symboliques suivantes sont disponibles pour **mode** :

S_IRWXU : 00700 L'utilisateur (propriétaire du fichier) a les autorisations de lecture, écriture, exécution.
S_IRUSR : 00400 L'utilisateur a l'autorisation de lecture.

S_IWUSR : 00200 L'utilisateur a l'autorisation d'écriture.
S_IXUSR : 00100 L'utilisateur a l'autorisation d'exécution.
S_IRWXG : 00070 Le groupe a les autorisations de lecture, écriture, exécution.
S_IRGRP : 00040 Le groupe a l'autorisation de lecture.
S_IWGRP : 00020 Le groupe a l'autorisation d'écriture.
S_IXGRP : 00010 Le groupe a l'autorisation d'exécution.
S_IRWXO : 00007 Tout le monde a les autorisations de lecture, écriture, exécution.
S_IROTH : 00004 Tout le monde a l'autorisation de lecture.
S_IWOTH : 00002 Tout le monde a l'autorisation d'écriture.
S_IXOTH : 00001 Tout le monde a l'autorisation d'exécution.

Le mode devrait toujours être indiqué quand **O_CREAT** est dans les attributs **flags**, (il est ignoré dans les autres cas).

creat() est équivalent à open() avec l'attribut **flags** égal à **O_CREAT** | **O_WRONLY** | **O_TRUNC**.

VALEUR RENVOYÉE

open() et creat() renvoient le nouveau descripteur de fichier s'ils réussissent, ou -1 s'ils échouent, auquel cas **errno** contient le code d'erreur.

NOTES

Notez que open() peut ouvrir des fichiers spéciaux mais creat() ne peut pas en créer, il faut utiliser mknod(2) à la place.

Sur les systèmes de fichiers NFS, où la correspondance d'UID est activée, open() peut renvoyer un descripteur de fichier alors qu'une requête read(2) par exemple sera refusée avec le code d'erreur EACCES.
En effet, c'est parce que le client a effectué open() en vérifiant les autorisations d'accès, mais la correspondance d'UID est calculée par le serveur au moment des requêtes de lecture ou d'écriture.

Si un fichier est créé, ses horodatages st_atime, st_ctime, st_mtime (respectivement heure de dernier accès, de dernière modification d'état, et de dernière modification ; voir stat(2)) sont fixés à l'heure actuelle, ainsi que les champs st_ctime et st_mtime du répertoire parent. Sinon, si le fichier est modifié à cause de l'attribut **O_TRUNC**, ses champs st_ctime et st_mtime sont remplis avec l'heure actuelle.

ERREURS

EACCES : L'accès demandé au fichier est interdit, ou la permission de parcours pour l'un des répertoires du chemin **pathname** est refusée, ou le fichier n'existe pas encore et le répertoire parent ne permet pas l'écriture. (Voir aussi path_resolution(2).)
EEXIST : **pathname** existe déjà et **O_CREAT** et **O_EXCL** ont été indiqués.

EFAULT : **pathname** pointe en dehors de l'espace d'adressage accessible

EISDIR : On a demandé une écriture alors que **pathname** correspond à un répertoire (en fait, **O_WRONLY** ou **O_RDWR** ont été demandés).

ELOOP : **pathname** contient une référence circulaire (à travers un lien symbolique), ou l'attribut **O_NOFOLLOW** est indiqué et **pathname** est un lien symbolique.

EMFILE : Le processus a déjà ouvert le nombre maximal de fichiers.

ENAMETOOLONG : **pathname** est trop long.

ENFILE : La limite du nombre total de fichiers ouverts sur le système a été atteinte.

ENODEV : **pathname** correspond à un fichier spécial et il n'y a pas de périphérique correspondant. (Ceci est un bogue du noyau Linux ; dans cette situation, ENXIO devrait être renvoyé.)

ENOENT : **O_CREAT** est absent et le fichier n'existe pas. Ou un répertoire du chemin d'accès **pathname** n'existe pas, ou est un lien symbolique pointant nulle part.

ENOMEM : Pas assez de mémoire pour le noyau.

ENOSPC : **pathname** devrait être créé mais le périphérique concerné n'a plus assez de place pour un nouveau fichier.

ENOTDIR: Un élément du chemin d'accès **pathname** n'est pas un répertoire, ou l'attribut **O_DIRECTORY** est utilisé et **pathname** n'est pas un répertoire.

ENXIO : **O_NONBLOCK** | **O_WRONLY** est indiqué, le fichier est une FIFO et le processus n'a pas de fichier ouvert en lecture. Ou le fichier est un noeud spécial et il n'y a pas de périphérique correspondant.

EOVERFLOW : **pathname** fait référence à un fichier régulier, trop grand pour pouvoir être ouvert – voir **O_LARGEFILE** plus haut.

EPERM : L'attribut **O_NOATIME** est indiqué, mais l'UID effectif de l'appelant n'est pas le propriétaire du fichier, et l'appelant n'est pas privilégié (CAP_FOWNER).

EROFS : Un accès en écriture est demandé alors que **pathname** réside sur un système de fichiers en lecture seule.

ETXTBSY : On a demandé une écriture alors que **pathname** correspond à un fichier exécutable actuellement utilisé.

EWOULDBLOCK : L'attribut **O_NONBLOCK** est indiqué, et un bail incompatible est détenu sur le fichier (voir fcntl(2)).

RESTRICTIONS

Plusieurs problèmes se posent avec le protocole NFS, concernant entre autres **O_SYNC**, et **O_NDELAY**.
POSIX fournit trois variantes différentes des entrées-sorties synchronisées correspondant aux attributs **O_SYNC**, **O_DSYNC** et **O_RSYNC**. Actuellement (2.1.130) elles sont toutes équivalentes sous Linux.

>>

La lecture, l'écriture, et la fermeture se font avec les même fonction que celles utilisées pour les tubes anonymes (et les fichiers), c'est à dire avec `read()`, `write()`, et `close()`.

10.2 Caractéristiques communes aux IPCs

Comme nous l'avons déjà vu, les IPC Système V (*Inter Process Communication*) forment un groupe de trois outils de communication :

- les ***files de messages*** ou ***MSQ*** (*messages queues*) ;
- les ***regions de mémoire partagée*** (*shared memory*) ;
- les ***sémaphores***.

Ces trois outils sont gérés dans des tables du système, une par outils. Un outil IPC est identifié de manière unique par un identifiant externe appelé la ***clé*** (qui a le même rôle que le chemin d'accès d'un fichier) et par un identifiant interne (qui joue le rôle de descripteur, comme pour les descripteurs de fichiers). Un outil IPC est accessible à tout processus connaissant l'identifiant interne de cet outil. La connaissance de cet identifiant s'obtient par héritage, ou par une demande explicite au système au cours de laquelle le processus fournit l'identifiant externe de l'outil IPC.

La clé est une valeur numérique de type `key_t`. Les processus désirant utiliser un même outil IPC pour communiquer doivent se mettre d'accord sur la valeur de la clé référençant l'outil. Ceci peut être fait de deux manières:

- la valeur de la clé est figée dans le code de chacun des processus,
- ou la valeur de la clé est calculée par le système à partir d'une référence commune à tous les processus. Cette référence est composée de deux parties : un nom de fichier et un entier.

Le calcul de la clé est effectué par la fonction système « `ftok()` » :

```
<<
SYNOPSIS
       #include <sys/types.h>
       #include <sys/ipc.h>
       key_t ftok(const char *pathname, int proj_id);

DESCRIPTION
       La fonction ftok() utilise l'identité du fichier indiqué
       par  pathname (qui doit exister et être accessible), et les
       huit bits de poids faible de proj_id (qui doit être non nul)
       pour créer une clé IPC Système V  de type key_t, utilisable
       avec msgget(2), semget(2) ou shmget(2).

       La  valeur résultante est la même pour tous les chemins
       d'accès identifiant le même fichier, en utilisant une valeur
       identique pour  proj_id.
       La  valeur  devrait  être  différente  lorsque  des fichiers
       différents (existants simultanément), ou des identificateurs
       de projet différents sont employés.

VALEUR RENVOYÉE
       Si  elle  réussit, la fonction ftok() renvoie la clé de type
       key_t créée. Sinon elle renvoie -1, et errno indique
       l'erreur de la même façon que l'appel système stat(2).

>>
```

A noter que les IPC sont créés à l'aide de primitives ayant la forme `xxxget()`. Le dernier argument des ces fonctions (le flag) permet de spécifier les droits (en lecture/écriture, pour l'utilisateur qui l'a créé, et pour les autres). Pour ce, il suffit d'utiliser les constantes définies dans « #include <sys/stat.h> » (Cf. `man 2 stat`). Par exemple, `S_IRUSR` et `S_IWUSR` spécifient des permissions de lecture et écriture pour le propriétaire de l'IPC, tout comme `S_IROTH` et `S_IWOTH` spécifient des permissions de lecture et écriture pour les autres utilisateurs. En l'absence de ces *flags*, seul l'utilisateur `root` pourrait utiliser les IPC. Autrement, si vous programmez à l'aide des IPCs, et si votre code ne fonctionne que sous root (et pas en étant connecté comme utilisateur quelconque), c'est que vous avez oublié ces options.

Sous shell, la commande ipcs permet de lister tous les IPC existants à un moment donné dans le système. Pour chaque IPC, cette commande fournit :

- le type (q pour message queue, s pour sémaphore, et m pour mémoire partagée) ;
- l'identification interne,
- la valeur de la clé,
- les droits d'accès,
- le propriétaire et le groupe propriétaire.

Exemple :

```
# ipcs
IPC status from /dev/mem as of sam 14 avr 19:53:17 DFT 2007
T         ID     KEY        MODE       OWNER    GROUP
Message Queues:
q    1310720 0x4107001c -Rrw-rw----     root   printq
q    3276801   00000000 --rw-rw-rw-     root   system
q    3276802   00000000 -Rrw-rw-rw-     root   system
Shared Memory:
m          0 0x58052224 --rw-rw-rw-     root   system
m          1 0x47041246 --rw-r--r--   imnadm   imnadm
m          2 0x58041246 --rw-r--r--   imnadm   imnadm
m    3145739   00000000 --rw-rw-rw-     root   system
m    3276812 0x610b0002 --rw-rw-rw-     root   system
m         13 0x0d052cf4 --rw-rw-rw-     root   system
Semaphores:
s     262144 0x58052224 --ra-ra-ra-     root   system
s          1 0x44052224 --ra-ra-ra-     root   system
s     131074   00000000 --ra-ra-ra-   imnadm   imnadm
```

10.3 Les files de messages (*messages queues*/*MSQ*)

Le principe repose sur le concept de communication (qui peut être bi-directionnel) par « boîte aux lettres ». Le noyau Linux permet de créer/gérer `MSGMNI` (128 par défaut) *boîtes aux lettres* de messages. La création ou l'accès à une file de messages existant se font à l'aide de la fonction « `msgget()` » :

```
<<
SYNOPSIS

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

       int msgget(key_t key, int msgflg);
```

DESCRIPTION

L'appel système msgget() renvoie l'identifiant de la file de messages associée à la clé **key**. Une nouvelle file de messages est créée si **key** a la valeur IPC_PRIVATE ou bien si **key** n'est pas IPC_PRIVATE, aucune file de messages n'est associée à **key**, et IPC_CREAT a été introduit dans **msgflg**.

Si **msgflg** indique à la fois IPC_CREAT et IPC_EXCL et une file de messages est déjà associée à **key**, msgget() échoue en positionnant errno à EEXIST. Ceci est similaire au comportement de open(2) avec la combinaison **O_CREAT** | **O_EXCL**.

Lors de la création, les bits de poids faibles de l'argument **msgflg** définissent les permissions d'accès à la file de message (pour le propriétaire, le groupe, et les autres) avec le même format et la même signification que les permissions d'accès (mode) dans les appels open(2). (Les permissions d'exécution ne sont pas utilisées).

Pendant la création, l'appel système initialise la structure système msqid_ds (voir msgctl(2)) de la file de messages comme suit :

msg_perm.cuid et msg_perm.uid sont remplis avec l'UID effectif du processus appelant.

msg_perm.cgid et msg_perm.gid sont remplis avec le GID effectif du processus appelant.
Les 9 bits de poids faibles de **msgflg** sont copiés dans les 9 bits de poids faibles de msg_perm.mode.

msg_qnum, msg_lspid, msg_lrpid, msg_stime et msg_rtime sont fixés à 0.

msg_ctime est rempli avec l'heure actuelle.

msg_qbytes est rempli avec la limite système MSGMNB.

Si la file de message existe déjà, les permissions d'accès sont contrôlées, et une vérification est faite pour voir si la file est prête à être détruite.

VALEUR RENVOYÉE

msgget() renvoie l'identifiant de la file de messages (un entier positif), s'il réussit. En cas d'échec -1 est renvoyé et errno contient le code d'erreur.

ERREURS

En cas d'échec, errno prend l'une des valeurs suivantes :

EACCES : Une file de messages existe associée à la clé **key**, mais le processus appelant n'a pas de permissions pour accéder à cette file et n'a pas la capacité CAP_IPC_OWNER.

EEXIST : Une file de messages existe associée à la clé **key** et **msgflg** réclame à la fois IPC_CREAT et IPC_EXCL.

ENOENT : Aucune file de messages n'existe associée à la clé **key** et **msgflg** ne contient pas IPC_CREAT.
ENOMEM : Le système doit créer une file de messages, mais n'a pas assez de mémoire pour les nouvelles structures de données.
ENOSPC : Le nombre maximum de files de messages sur le système (MSGMNI) est atteint.

NOTES

IPC_PRIVATE n'est pas destiné au champ msgflg mais est du type key_t.
Si cette valeur spéciale est fournie à la place de key, l'appel système ignorera tout sauf les 9 bits de poids faibles de msgflg et créera une nouvelle file de messages.

La limite système concernant les files de messages et Affectant msgget() est MSGMNI. Nombre maximum de files de messages sur le système (sous Linux, cette limite peut être consultée et modifiée grâce au fichier /proc/sys/kernel/msgmni).

>>

La création d'une file se fait en positionnant `msgflg` à `IPC_CREAT` et `IPC_EXCL`. La clé est calculée au préalable avec `ftok()`. L'accès à une file existante se fait en mettant le paramètre `msgflg` à 0.

Si clé vaut « `IPC_PRIVATE` », la file ne sera accessible que du processus lui-même, et à ses descendants.

Ensuite, `msgsnd()` et `msgrcv()` permettent d'envoyer et de recevoir des messages :

<<
SYNOPSIS

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

int msgsnd(int msqid, const void *msgp, size_t msgsz, \
                int msgflg);

ssize_t  msgrcv(int  msqid,  void *msgp, size_t msgsz, \
                long msgtyp, int msgflg);
```

DESCRIPTION

Les appels système msgsnd() et msgrcv() servent respectivement à envoyer et à recevoir des messages d'une file de messages. Le processus appelant doit avoir une permission d'écriture sur la file pour envoyer un message, et une permission de lecture pour en recevoir un.

L'argument **msgp** est un pointeur vers une structure définie par l'appelant de forme générale suivante :

```
struct msgbuf {
     long mtype;     /* type de message ( > 0 ) */
     char mtext[1];  /* contenu du message */
};
```

Le champ mtext est un tableau ou autre structure de taille **msgsz**, valeur entière positive ou nulle. Les message de taille nulle (sans champ mtext) sont autorisés. Le membre mtype doit avoir une valeur strictement positive qui puisse être utilisée par le processus lecteur pour la sélection de messages (voir la description de msgrcv ci-dessous).

L'appel système msgsnd() insère une copie du message pointé par l'argument **msgp** dans la file dont l'identifiant est indiqué par la valeur de l'argument **msqid**.

S'il y a assez de place dans la file, msgsnd() réussit immédiatement (la capacité de la file est définie par le champ msg_bytes de la structure associée à la file de messages). Durant la création de la file, ce champ est initialisé à MSGMNB octets, mais cette limite peut être modifiée avec msgctl().) S'il n'y a pas assez de place, alors le comportement par défaut de msgsnd() est de bloquer jusqu'à obtenir suffisamment d'espace. En indiquant IPC_NOWAIT dans l'argument **msgflg**, le message ne sera pas envoyé et l'appel système échouera en retournant EAGAIN dans **errno**.

Un appel à msgsnd() bloqué peut aussi échouer si la file est supprimée (auquel cas l'appel système échouera avec errno valant EIDRM), ou si un signal a été intercepté (auquel cas l'appel système échouera avec errno valant EINTR). (msgsnd et msgrcv ne sont jamais relancés automatiquement après interruption par un gestionnaire de signal, quelle que soit la configuration de SA_RESTART lors de l'installation du gestionnaire.)

Si l'appel système réussit, la structure décrivant la file de messages est mise à jour comme suit :

msg_lspid contient le PID du processus appelant.
msg_qnum est incrémenté de 1.
msg_stime est rempli avec l'heure actuelle.

L'appel système msgrcv() supprime un message depuis la file indiquée par **msqid** et le place dans le tampon pointé par **msgp**.

L'argument **msgsz** indique la taille maximale en octets du membre mtext de la structure pointée par l'argument **msgp**. Si le contenu du message est plus long que **msgsz** octets, le comportement dépend de la présence ou non de MSG_NOERROR dans **msgflg**. Si MSG_NOERROR est spécifié, alors le message sera tronqué (et la partie tronquée sera perdue) ; si MSG_NOERROR n'est pas spécifié, le message ne sera pas extrait de la file, et l'appel système échouera en renvoyant -1 et en indiquant E2BIG dans errno.

L'argument **msgtyp** indique le type de message désiré :

- Si **msgtyp** vaut 0, le premier message est lu.
- Si **msgtyp** est supérieur à 0, alors le premier
- message de type **msgtyp** est extrait de la file.
- Si **msgflg** contient MSG_EXCEPT, l'inverse est

effectué, le premier message de type différent de **msgtyp** est extrait de la file.
- Si **msgtyp** est inférieur à 0, le premier message de la file avec un type inférieur ou égal à la valeur absolue de **msgtyp** est extrait.

L'argument **msgflg** est composé d'un OU binaire « | » avec les options suivantes :

IPC_NOWAIT :
Retourne immédiatement si aucun message du type désiré n'est présent dans la file. L'appel système échoue et errno est fixé à ENOMSG.

MSG_EXCEPT :
Utilisé avec **msgtyp** supérieur à 0 pour lire les messages de type différent de **msgtyp**.

MSG_NOERROR :
Tronque silencieusement les messages trop longs.

Si aucun message du type requis n'est disponible et si on n'a pas demandé IPC_NOWAIT dans **msgflg**, le processus appelant est bloqué jusqu'à l'occurrence d'un des événements suivants :

- Un message du type désiré arrive dans la file.
- La file de messages est supprimée. L'appel système échoue et errno contient EIDRM.
- Le processus appelant reçoit un signal à intercepter. L'appel système échoue et errno contient EINTR.

Si l'appel système réussit, la structure décrivant la file de messages est mise à jour comme suit :
- msg_lrpid est rempli avec le PID du processus appelant.
- msg_qnum est décrémenté de 1.
- msg_rtime est rempli avec l'heure actuelle.

VALEUR RENVOYÉE

En cas d'échec les deux appels système renvoient -1 et **errno** contient le code d'erreur. Sinon msgsnd() renvoie 0 et msgrcv() renvoie le nombre d'octets copiés dans la table mtext.

ERREURS

En cas d'échec de msgsnd(), errno aura l'une des valeurs suivantes :

EACCES : Le processus appelant n'a pas de permissions d'écriture dans la file et n'a pas la capacité CAP_IPC_OWNER.

EAGAIN : Le message n'a pas pu être envoyé à cause de la limite msg_qbytes pour la file et de la requête IPC_NOWAIT dans **msgflg**.

EFAULT : **msgp** pointe en dehors de l'espace d'adressage accessible.

EIDRM : La file de messages a été supprimée.

EINTR : Un signal est arrivé avant d'avoir pu écrire

quoi que ce soit.

EINVAL : **msqid** est invalide, ou bien mtype n'est pas positif, ou bien **msgsz** est invalide (négatif ou supérieur à la valeur MSGMAX du système).

ENOMEM : Le système n'a pas assez de mémoire pour copier le message pointé par **msgp**.

En cas d'échec de msgrcv(), errno prend l'une des valeurs suivantes :

E2BIG : Le message est plus long que **msgsz**, et MSG_NOERROR n'a pas été indiqué dans **msgflg**.

EACCES : Le processus appelant n'a pas de permission de lecture dans la file et n'a pas la capacité CAP_IPC_OWNER.

EAGAIN : Aucun message n'est disponible dans la file, et IPC_NOWAIT est spécifié dans **msgflg**.

EFAULT : **msgp** pointe en dehors de l'espace d'adressage accessible.

EIDRM : La file de messages a été supprimée alors que le processus attendait un message.

EINTR : Un signal est arrivé avant d'avoir pu lire quoi que ce soit.

EINVAL : msgqid ou **msgsz** invalides.

ENOMSG : IPC_NOWAIT a été requis et aucun message du type réclamé n'existe dans la file.

NOTES

L'argument **msgp** est déclaré comme un struct msgbuf * avec les bibliothèques libc4, libc5, glibc 2.0, glibc 2.1. Il est déclaré comme un void * avec la bibliothèque glibc 2.2, suivant ainsi les spécifications SUSv2 et SUSv3.

Les limites système concernant les files de messages et affectant msgsnd() sont :

MSGMAX : Taille maximale d'un message : 8192 octets (sous Linux, cette limite peut être lue et modifiée grâce au fichier /proc/sys/kernel/msgmax).

MSGMNB : Taille maximale d'une file de messages : 16384 octets (sous Linux, elle peut être lue et modifiée grâce au fichier /proc/sys/kernel/msgmnb). Le superutilisateur peut augmenter la taille d'une file de messages au-delà de MSGMNB en utilisant l'appel système msgctl(2).

L'implémentation des files de messages sous Linux n'a pas de limite intrinsèque pour le nombre maximal d'entêtes de messages (MSGTQL) et la taille maximale de l'ensemble de tous les messages sur le système (MSGPOOL).

>>

A noter qu'un appel à `msgrcv()` est bloquant, sauf si `msgflg` a possède la valeur « `IPC_NOWAIT` ».

Une file de message est détruite avec un appel « `msgctl(int msqid, IPC_RMID, NULL)` » (Cf. « `man 2 msgctl` » pour plus d'informations sur cette fonction).

Sous shell, la suppression d'une file de message peut se faire à l'aide de la commande « ipcrm » :

```
           # ipcrm –q identifiant
     ou    # ipcrm –Q cle
```

Exemple d'utilisation des files de messages : un processus A crée une MSQ de clé 12300, puis écrit le message « ceci est un message » à destination du processus B. Le processus B lit le message et l'affiche, puis détruit la MSQ. Correction (simplissime) :

```
<<
/* Processus A */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>

#define cle 12300

int msqid;
struct msgbuf_EXD
{
   long mtype;
   char mtext[128];
} msgp;

int main()
{
   /* Création du "message queue" */
   msqid = msgget(cle, \
              IPC_CREAT|IPC_EXCL|S_IRUSR|S_IWUSR|S_IROTH|S_IWOTH);

   /* Remplit la structure */
   msgp.mtype = 12;
   strcpy(msgp.mtext, "ceci est un message");

   /* Envoie le message */
   msgsnd(msqid, &msgp, strlen(msgp.mtext), 0);

   return(0);
}

  /*********************************************************/

/* Processus B */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>

#define cle 12300

int msqid;
struct msgbuf_EXD
{
   long mtype;
   char mtext[20];
} msgp;
```

```
int main()
{
   /* Réouvre la file de message */
   msqid = msgget(cle, 0);

   /* Lit le message */
   msgrcv(msqid, &msgp, 19, 12, 0);

   /* Détruit la MSQ */
   msgctl(msqid, IPC_RMID, NULL);

   /* Affiche le message */
   printf("Message : \"%s\"\n", msgp.mtext);

   return(0);
}
```

>>

10.4 Les régions de mémoire partagée

Nous savons que la mémoire allouée à chaque processus est protégée, afin qu'un autre processus utilisateur ne puisse venir y lire ou écrire des données. Néanmoins, il peut arriver que des processus aient besoin de partager une zone mémoire. C'est le rôle des ***régions de mémoire partagées***, ou *shared memory*.

Evidemment, il ne faut pas que les processus qui partagent une zone mémoire l'utilisent de façon anarchique. Il faut que les processus mettent une sorte de verrou sur la zone quand ils l'utilisent, puis déverrouillent la zone une fois le travail terminé, laissant ainsi à un autre processus la possibilité de l'utiliser à son tour. C'est le principe des *sémaphores* que nous verrons dans un prochain chapitre.

A noter que comme les files de messages, les régions de mémoire partagée est un mécanisme d'IPC. Aussi, il est chacun de ces objets est identifié par une clé, et il faut utiliser `ftok()` pour créer/gérer ces clés (ou le programmeur choisi une valeur de clé de façon conventionnelle).

La création d'une région de mémoire partagée se fait à l'aide de la fonction `shmget()`, dont le principe s'apparente à celui de `msgget()` :

```
<<
SYNOPSIS

       #include <sys/ipc.h>
       #include <sys/shm.h>
       int shmget(key_t key, size_t size, int shmflg);

DESCRIPTION
       shmget() renvoie l'identifiant du segment de mémoire
       partagée associé à la valeur de l'argument key. Un nouveau
       segment mémoire, de taille size arrondie au multiple
       supérieur de PAGE_SIZE, est créé si key a la valeur
       IPC_PRIVATE ou si aucun segment de mémoire partagée
       n'est associé à key, et IPC_CREAT est présent dans shmflg.

       Si shmflg contient à la fois les attributs IPC_CREAT et
       IPC_EXCL, et si un segment de mémoire partagée est déjà
       associé à key, shmget() échoue avec le code d'erreur
```

EEXIST. Ceci est similaire au comportement de open(2) avec la combinaison O_CREAT | O_EXCL.

shmflg est composé de :
IPC_CREAT : Créer un nouveau segment. Sinon shmget() recherche le segment associé à **key** et vérifie que l'appelant a la permission d'y accéder.
IPC_EXCL : Utilisé avec IPC_CREAT pour garantir l'échec si le segment existe déjà.

mode d'accès :
(les 9 bits de poids faibles) Permissions pour le propriétaire, le groupe et les autres. Ces bits ont le même format et la même signification que l'argument mode de open(2). Actuellement la permission d'exécution n'est pas utilisée par le système.

SHM_HUGETLB : (depuis Linux 2.6)
Allouer le segment en utilisant des pages immenses. Voir le fichier documentation/vm/hugetlbpage.txt dans les sources du noyau pour plus d'informations.

SHM_NORESERVE : (depuis Linux 2.6.15)
Cet attribut a le même objet que l'attribut MAP_NORESERVE de mmap(2). Ne pas réserver d'espace de swap pour ce segment. Lorsque de l'espace en swap est réservé, le système garantit qu'il sera possible de modifier le segment. Lorsque l'espace en swap n'est pas réservé, on peut recevoir SIGSEGV lors d'une écriture si la mémoire physique est pleine. Voir aussi la discussion du fichier /proc/sys/vm/overcommit_memory dans proc(5).

Si un nouveau segment de mémoire partagée est créé, le système initialise son contenu à zéro, et la structure shmid_ds (voir shmctl(2)) associée au segment comme suit :
shm_perm.cuid et shm_perm.uid contiennent l'UID effectif de l'appelant.

shm_perm.cgid et shm_perm.gid contiennent le GID effectif de l'appelant.

Les 9 bits de poids faibles de shm_perm.mode contiennent les 9 bits de poids faibles de **shmflg**.

shm_segsz prend la valeur **size**.

shm_lpid, shm_nattch, shm_atime et shm_dtime sont mis à 0.

shm_ctime contient l'heure actuelle.

Si le segment de mémoire existe déjà, les permissions d'accès sont vérifiées, et un contrôle a lieu pour voir

s'il est marqué pour destruction.

AUTRES APPELS SYSTÈMES

- fork() Après un fork() le fils hérite des segments de mémoire partagée.
- exec() Après un exec() tous les segments de mémoire partagée sont détachés (pas détruits).
- exit() Lors d'un exit() tous les segments de mémoire partagée sont détachés (pas détruits).

VALEUR RENVOYÉE

Un identifiant de segment shmid valide est renvoyé en cas de réussite, sinon -1 est renvoyé et errno contient le code d'erreur.

ERREURS

En cas d'erreur, errno prend l'une des valeurs suivantes :

EACCES : L'appelant n'a pas les autorisations d'accès au segment, et n'a pas la capacité CAP_IPC_OWNER.

EEXIST : **shmflg** contient IPC_CREAT | IPC_EXCL mais le segment existe déjà.

EINVAL : Un nouveau segment devait être créé et **size** < SHMMIN ou **size** > SHMMAX, ou bien un segment associé à **key** existe, mais sa taille est inférieure à **size**.

ENFILE : La limite du nombre total de fichiers ouverts sur le système a été atteinte.

ENOENT : Aucun segment n'est associé à **key**, et IPC_CREAT n'était pas indiqué.

ENOMEM : Pas assez de mémoire pour allouer le segment.

ENOSPC : Tous les identifiants de mémoire partagée sont utilisés (SHMMNI), ou l'allocation d'un segment partagé de taille **size** dépasserait les limites de mémoire partagée du système (SHMALL).

EPERM : L'attribut SHM_HUGETLB est indiqué, mais l'appelant n'est pas privilégié (ne possède pas la capacité CAP_IPC_LOCK).

NOTES

IPC_PRIVATE n'est pas une option mais une valeur de type key_t. Si cette valeur spéciale est utilisée comme clé, l'appel système ignore tout sauf les 9 bits de poids faibles de **shmflg** et tente de créer un nouveau segment.

Les limites suivantes influent sur l'appel système shmget :

SHMALL : Nombre maximal de pages de mémoire partagée sur le système (sous Linux, cette limite peut être lue et modifiée grâce au fichier /proc/sys/kernel/shmall).

SHMMAX : Taille maximale d'un segment partagé (sous Linux, cette limite peut être lue et modifiée grâce au fichier /proc/sys/kernel/shmmax).

SHMMIN : Taille minimale d'un segment partagé : dépend de l'implémentation (actuellement 1 octet, bien que PAGE_SIZE soit la valeur effectivement utilisée).

SHMMNI : Nombre maximal de segments de mémoire partagée sur le système (actuellement 4096, mais 128 avant Linux 2.3.99 ; sous Linux, cette limite peut être lue et modifiée grâce au fichier /proc/sys/kernel/shmmni).

Il n'y a pas de limite pour le nombre de segments partagés par processus (SHMSEG).

>>

Une fois la mémoire partagée allouée, il faut pouvoir récupérer un pointeur qui pointe vers la zone allouée par la fonction shmget(). On dit que nous allons attacher un pointeur à de la mémoire partagée. Cette opération s'effectue à l'aide de la fonction shmat(). Inversement, la routine shmdt() détache un pointeur à une région de mémoire partagée :

<<
SYNOPSIS

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

void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

DESCRIPTION

La fonction shmat() attache le segment de mémoire partagée identifié par **shmid** au segment de données du processus appelant. L'adresse d'attachement est indiquée par **shmaddr** avec les critères suivants :

Si **shmaddr** vaut NULL, le système essaye de trouver une adresse libre pour attacher le segment.

Si **shmaddr** n'est pas NULL et si SHM_RND est indiqué dans **shmflg**, l'attachement a lieu à l'adresse **shmaddr** arrondie au multiple inférieur de SHMLBA. Sinon **shmaddr** doit être alignée sur une frontière de page, où l'attachement a lieu.

Si SHM_RDONLY est indiqué dans **shmflg**, le segment est attaché en lecture seulement, et le processus doit disposer de la permission de lecture dessus. Sinon le segment est attaché en lecture et écriture et le processus doit disposer des deux permissions d'accès. Il n'y a pas de notion d'écriture seule pour les segments de mémoire partagée.

L'option (spécifique à Linux) SHM_REMAP peut être ajoutée dans **shmflg** pour indiquer que la projection du segment doit remplacer une projection précédente dans l'intervalle commençant en **shmaddr** et s'étendant sur la taille du segment. (Normalement une erreur EINVAL se produirait si une projection existait déjà dans l'intervalle indiqué.) Dans ce cas, **shmaddr** ne doit pas être NULL.

La valeur brk(2) du processus appelant n'est pas altérée par l'attachement. Le segment est automatiquement détaché quand le processus se termine. Le même segment peut être attaché à

la fois en lecture seule et en lecture/écriture. Il peut également être attaché en plusieurs endroits de l'espace d'adressage du processus.

Quand shmat() réussit, les membres de la structure shmid_ds associée au segment de mémoire partagée (voir shmctl(2)) sont mis à jour ainsi :

- shm_atime correspond à l'heure actuelle.
- shm_lpid contient le PID de l'appelant.
- shm_nattch est incrémenté de 1.

La fonction shmdt() détache le segment de mémoire partagée situé à l'adresse indiquée par **shmaddr**. Le segment doit être effectivement attaché, et l'adresse **shmaddr** doit être celle renvoyée précédemment par shmat().

Quand shmdt() réussit, les membres de la structure shmid_ds associée au segment de mémoire partagée sont mis à jour ainsi :

- shm_dtime correspond à l'heure actuelle.
- shm_lpid contient le PID de l'appelant.
- shm_nattch est décrémenté de 1. S'il devient nul, et si le segment est marqué pour destruction, il est effectivement détruit.

AUTRES APPELS SYSTÈMES

- fork() Après un fork() le fils hérite des segments de mémoire partagée.
- exec() Après un exec() tous les segments de mémoire partagée sont détachés (pas détruits).
- exit() Lors d'un exit() tous les segments de mémoire partagée sont détachés (pas détruits).

VALEUR RENVOYÉE

Si elles réussit, shmat() renvoie l'adresse d'attachement du segment de mémoire partagée. En cas d'échec (void *) -1 est renvoyé, et errno contient le code d'erreur.

shmdt() renvoie zéro s'il réussit et -1 s'il échoue et écrit la cause de l'erreur dans errno.

ERREURS

Quand shmat() échoue, errno prend l'une des valeurs suivantes :

EACCES : L'appelant n'a pas les permissions d'accès nécessaires pour l'attachement, et n'a pas la capacité CAP_IPC_OWNER.

EINVAL : **shmid** est invalide, **shmaddr** est mal alignée (c'est-à-dire pas alignée sur une page et SHM_RND n'a pas été précisé) ou invalide, échec lor de brk(), ou SHM_REMAP a été réclamé et **shmaddr** est NULL.

ENOMEM : Pas assez de mémoire pour le système.

Quand shmdt() échoue, errno prend l'une des valeurs suivantes :

EINVAL : Aucun segment de mémoire partagée n'est attaché à l'adresse **shmaddr**, ou bien **shmaddr** n'est pas un multiple de la taille de page.

NOTES

Utiliser shmat() avec **shmaddr** égale à NULL est la manière conseillée, portable, d'attacher un segment de mémoire partagée. Soyez conscients que le segment attaché de cette manière peut l'être à des adresses différentes dans les différents processus. Ainsi, tout pointeur contenu dans la mémoire partagée doit être relatif (typiquement par rapport au début du segment) et pas absolu.

Sous Linux, il est possible d'attacher un segment de mémoire partagée qui est déjà marqué pour effacement. Cependant, ce comportement n'est pas décrit par POSIX.1-2001, et beaucoup d'autres implémentations ne le permettent pas.

Les paramètres système suivants influent sur shmat() :

SHMLBA : Multiple pour l'adresse de début de segment. Doit être aligné sur une frontière de page. Pour l'implémentation actuelle, SHMLBA à la même valeur que PAGE_SIZE.

L'implémentation n'a pas de limite intrinsèque pour le nombre maximal de segment de mémoire partagée par processus (SHMSEG).

>>

Une région de mémoire partagée peut être détruite à l'aide d'un appel :

```
shmctl(int shmid, IPC_RMID, NULL);
```

Cf. « man shmctl » pour plus de détails. Autrement, sous shell, la suppression d'une région de mémoire partagée peut se faire à l'aide de « ipcrm » (même principe que pour détruire une file de messages) :

```
# ipcrm –q identifiant
```
ou
```
# ipcrm –Q cle
```

Exemple d'utilisation de shared memory : le programme A, crée une mémoire partagée, va aller inscrire une structure dans cette mémoire. Un des champs de cette structure est mis à jour en permanence. Le programme B attache la mémoire partagée créée par A, son propre processus, puis va lire les données contenues dans la structure crée par A. Correction (vraiment pas belle) :

<<

```
/* Code source du processus A */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <math.h>

/* on définit une clef au hasard */
#define CLEF 12345

/* On définit une structure quelconque qui comporte un entier  */
/* et un double. */
```

```
typedef struct
{
  int a;
  double b;
} structure_partagee;

int main ()
{
  structure_partagee Data;
  int i = 0;
  /* identificateur du segment de mém. partagée associé à CLEF: */
  int mem_ID;
  /* Ptr sur l'adresse d'attachement du segment de mém partagée: */
  void *ptr_mem_partagee;

  /* On crée un nouveau segment mémoire de taille */
  /* "taille de ma structure data" octets, avec des */
  /* droits d'écriture et de lecture, et on s'assure */
  /* que l'espace mémoire a été correctement créé */
  if ((mem_ID = shmget (CLEF, sizeof (Data), \
                IPC_CREAT|S_IRUSR|S_IWUSR|S_IROTH|S_IWOTH)) < 0)
  {
      perror ("shmget");
      exit (1);
  }
  /* On attache le segment de mémoire partagée identifié */
  /* par mem_ID au segment de données du processus A dans */
  /* une zone libre déterminée par le Système d'exploitation */
  /* et on s'assure que le segment de mémoire a été */
  /* correctement attaché à mon processus */
  if ((ptr_mem_partagee = shmat (mem_ID, NULL, 0)) == (void *) -1)
  {
      perror ("shmat");
      exit (1);
  }

  /* On alloue des valeurs aux variables de la structure */
  Data.a = 2;
  Data.b = 2.6544;

  /* On met à jour ces valeurs en mémoire partagée. */
  /* ptr_mem_partagee est un pointeur de void. On */
  /* caste pour qu'il devienne un pointeur de */
  /* "structure_partagee". Et on va écrire dans */
  /* la structure Data à l'adresse pointée par ce pointeur. */

  *((structure_partagee *) ptr_mem_partagee) = Data;

  /* On va modifier en permanence le champ a de la structure */
  /* et le mettre à jour. Le processus B lira la structure Data. */
  while (1)
  {
      Data.a = i;
      *((structure_partagee *) ptr_mem_partagee) = Data;
      i++;
      if (i == 100000000)  /* on remets à 0 de temps en temps... */
        i = 0;
  }
```

```
  /* Une fois sortie de la boucle (bon OK là elle est infine), */
  /* on détache mon segment mémoire du processus, et quand tous */
  /* les processus en auront fait autant, ce segment mémoire */
  /* sera détruit. */
  shmdt (ptr_mem_partagee);

  return(0); /* bye bye... */
}

/************************************************************/

/* code source du processus B */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
/* !! même clef que celle du processus A!! */
#define CLEF 12345

/* Même structure que dans le programme A. */
/* Les noms des champs n'ont rien à voir */
/* avec le programme A, seule la structure */
/* est importante. */
typedef struct
{
  int c;
  double d;
} structure_partagee_B;

int main ()
{
  /* On déclare des variables aptes à recevoir les variables */
  /* de la structure "structure_partagee" définie dans le */
  /* processus A */
  int var1;
  double var2;
  /* identificateur du segment de mémoire partagée associé à */
  /* CLEF (là encore, le nom de cette variable n'a rien à voir */
  /* avec celle du programme A mais son contenu sera évidemment */
  /* identique) */
  int mem_ID_B;

  /* adresse d'attachement du segment de mémoire partagée */
  /* (même remarque) */
  void *ptr_mem_partagee_B;

  /* On  cherche le segment mémoire associé à CLEF et */
  /* on récupère l'identificateur de ce segment mémoire... */
  /* Rq: droits de lecture uniquement */
  if ((mem_ID_B = shmget (CLEF, sizeof (structure_partagee_B), \
                                                    0444)) < 0)
  {
      perror ("shmget");
      exit (1);
  }
  /* On attache le segment de mémoire partagée identifié par */
```

```
  /* mem_ID_B au segment de données du processus B dans une */
  /* zone libre déterminée par le Système d'exploitation */
  if ((ptr_mem_partagee_B = shmat (mem_ID_B,NULL,0))==(void *) -1)
  {
      perror ("shmat");
      exit (1);
  }

  /* On affiche le contenu des variables inscrites par A dans */
  /* la mémoire partagée */
  while (1)
  {
      var1 = ((structure_partagee_B *) ptr_mem_partagee_B)->c;
      var2 = ((structure_partagee_B *) ptr_mem_partagee_B)->d;
      printf("data.a : %d\n", var1);
      printf("data.b : %lf\n", var2);
  }

  /* On  détruit le segment (le segment n'est pas détruit tant */
  /* qu'au moins un processus est lié au segment) */
  shmdt (ptr_mem_partagee_B);

  return(0);
}
```

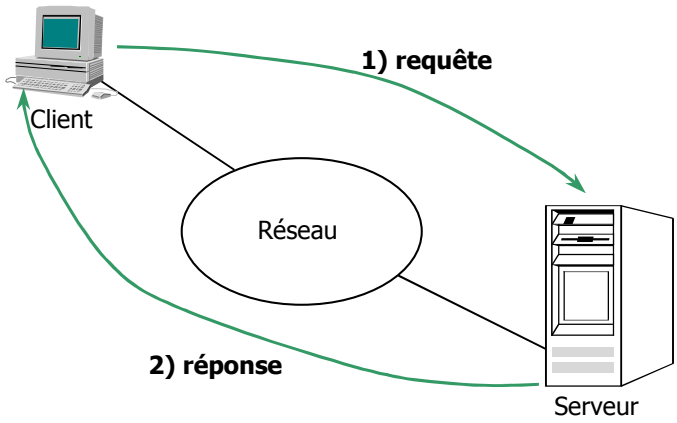
>>

11 La communication répartie

11.1 Le modèle client-serveur

Tous les mécanismes que nous avons vu jusqu'à présent concernent des opérations au sein d'une même machine. Or, il peut arriver que des processus distants, situés sur des machines différentes, aient besoin de communiquer via le réseau.

Une façon classique de traiter ce problème est d'utiliser le modèle client-serveur. Un ordinateur (le client) établit le dialogue en envoyant une requête via le réseau à une autre machine (le serveur), qui lui répond :



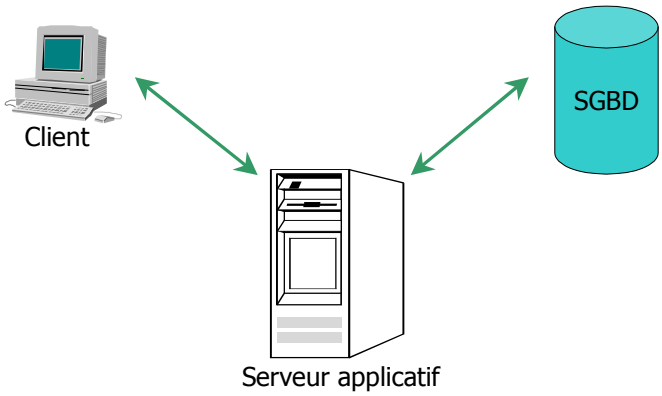
Modèle client-serveur

Nous pouvons caractériser les modèles clients-serveurs suivant 3 critères :

- la manière dont le serveur traite les requête en provenance du client. Il existe les serveurs :
  - *itératifs* : les requêtes sont traitées les unes à la suite des autres (une seule à la fois),
  - *parallèles* : le processus qui reçoit les requête fait un `fork()`, et la requête est traitée par le fils. Plus souple, ayant des meilleurs temps de réponse, ce modèle parallèle peut charger une machine, si de nombreuses machines envoient des requêtes en même temps. Une limite du nombre de fils évite de voir le serveur s'écrouler en cas de surcharge ;
- le niveau de fiabilité des serveurs. On distingue les serveurs :
  - *sans état* : en cas de crash d'un processus serveur qui traîte une requête, il est impossible de savoir ce qui a déjà été traité, et ce qu'il restait à faire,
  - *à état* : chaque étape intermétdiaire du traitement de la requête est historisée par le serveur sur le disque dur. Ainsi, en cas de panne, il est possible d'annuler ce qui avait déjà été fait et qui restait innachevé, ou de terminer le traitement ;
- enfin, la communication client/serveur peut se faire :
  - par envoi de courts messages (on parle de communication par *paquets* ou par *datagrammes*),
  - ou en établissant une *connexion* (les informations sont envoyées sous forme de « *flux de données* » ou « *flot de données* »).

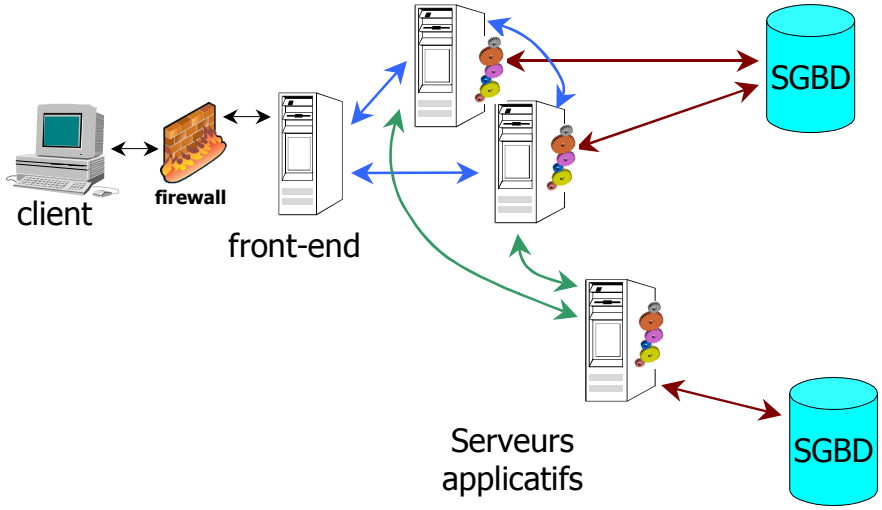
Il est tout à fait possible de cascader des le modèle de client/serveur. Par exemple, un client qui envoie une requête à un serveur d'application, qui envoie lui-même une requête à

un gestionnaire de base de données (SGBD). Dans ce dernier cas, on parle d'architecture 3-tiers :



Architecture 3-tiers

Le modèle 3-tiers peut ête étendu en répartissant la couche applicative sur plusieurs machines. On parle d'architecture N-tiers (ou multi-tiers) :



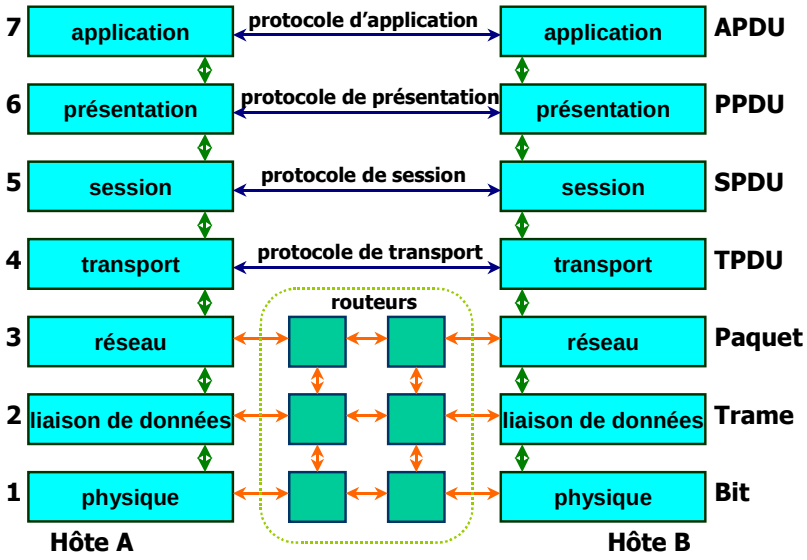
Architecture N-tiers

L'utilisation de plusieurs serveurs applicatifs peut avoir deux raisons :

- effectuer des calculs en parallèles (utilisation d'algorithmes de calculs répartis),
- et/ou répartir la charge sur plusieurs serveurs. Dans ce cas, une machine reçoit les requêtes des clients (le « *frontend* »), et les distribuent tour à tour à plusieurs serveurs, tous capables de traiter la requête.

11.2 Quelques rappels réseau

Les théoriciens décomposent un protocole de communication en « *couches* ». Le modèle OSI (de l'ISO) contient 7 couches :



Le modèle OSI

Classiquement, la communication Unix/Linux se fait avec le protocole TCP/IP. Nous nous intéresserons principalement aux protocoles de niveau 4/transport « TCP » (communication par *connexion*) et « UDP » (communication par *paquets*). Ces deux protocoles s'appuient sur le protocole de communication « IP » (niveau 3/réseau), qui s'appuie lui-même sur des protocoles de niveau 2 de type Ethernet, ou Wifi... (802.x).

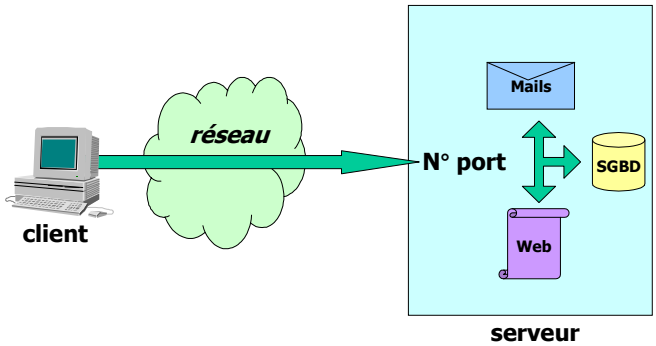
A noter que toute machine d'un réseau IP est identifiée au niveau 3 par une adresse IP. Dans le protocole IP version 4, une adresse IP est constituée de 4 octets.

De plus, les machines possèdent un nom de dommaine sous la forme :

```
nom_noeud. nom_ss-domaine[...].nom_domaine_racine
```

Le service réseau qui assure la traduction « *nom* » <-> « *adresse IP* » est le **DNS** (*Domain Name Service*).

Enfin, chaque service (mail, DNS, pop, http, etc) est identifié par un numéro, appelé « *numéro de port* » :



Identification du service par un numéro de port

Par convention, les numéros de ports sont assignés dans les plages suivantes :

- **Port n° 0 :** non utilisable pour une application. C'est un « jocker » qui indique au système que c'est à lui de compléter le numéro (Cf. N°49152 à 65535) ;

- **Ports 1 à 1023 :** ports réservés au superutilisateur (root/Administrateur). On y trouve les serveurs « classiques » (DNS, FTP, SMTP, Telnet, SSH...). Il existait anciennement un découpage 1-255, 256-511, et 512-1023 qui n'est plus utilisé ;
- **Ports 1024 à 49151 :** services enregistrés par l'IANA (Internet Assigned Numbers Authority), accessibles aux utilisateurs ordinaires ;
- **Ports 49152 à 65535 :** zone d'attribution automatique des ports, pour la partie cliente.

Sous Unix/Linux, l'API qui permet d'établir une communication entre un client et un serveur s'appuie sur la notion de « *socket* » (une *socket* s'apparente à une fiche). Il existe deux types de sockets :

- les sockets en *mode datagrammes*, qui s'appuient sur le protocole UDP,
- et les sockets en *mode connecté*, qui s'appuient sur le protocole TCP.

11.3 Les fonctions et structures propres à TCP/IP

11.3.1 La normalisation des entiers (endianness)

Certains microprocesseur (motorola 68000, sparc...) stockent les mots de plusieurs octets en mettant par convention l'octer de poid fort en premier (Exemple : le mot de 32 bits 0xa0b1c2d3 verra l'octet 0xa0 stocké en premier, 0xb1 en second, 0xc2 en 3ème, puis 0xd3 en dernier. On dit que ces microprocesseurs sont ***gros-boutistes***, ou fonctionne en mode ***big-endian***.

Inversement, par convention, d'autre CPU (Pentium par exemple) stockent les octes de poids faibles en premier. On parle de microprocesseurs ***petits-boutistes***, ou qu'ils fonctionnent en ***little-endian***.

Or, sans y faire attention, si un ordinateur gros-boutiste envoie un mot de 4 octets à un ordinateur petit-boutiste, les octets des mots se trouvent inversés, ce qui serait catastrophique.

Aussi, une convention a été choisie pour représenter les mot sur les réseaux IP (en fait, il s'agit de la convention « ***octets de poids fort en premier*** »). Il existe 4 fonctions (`htonl()`, `htons()`, `ntohl()`, `ntohs()`) qui permettent de normaliser les entiers (adresses IP, numéros de port, etc.), afin de les convertir dans la convention d'Internet. Si la machine locale a un CPU qui a la même convention que celle d'Internet, ces 4 fonctions ne font rien. Sinon, elles inversent les octets de poid faibles avec les octets de poids fort.

```
<<
SYNOPSIS
       #include <arpa/inet.h>
       uint32_t htonl(uint32_t hostlong);
       uint16_t htons(uint16_t hostshort);
       uint32_t ntohl(uint32_t netlong);
       uint16_t ntohs(uint16_t netshort);

DESCRIPTION
       La fonction  htonl()  convertit  l'entier  non  signé
       hostlong  depuis l'ordre des octets de l'hôte vers celui du
       réseau.

       La fonction htons() convertit l'entier court non signé
       hostshort depuis l'ordre des octets de l'hôte vers celui du
       réseau.
```

```
       La fonction ntohl() convertit l'entier non signé netlong
       depuis l'ordre des octets du réseau vers celui de l'hôte.

       La fonction ntohs() convertit un entier court non signé
       netshort depuis l'ordre des octets du réseau vers celui de
       l'hôte.

>>
```

11.3.2 La résolution de nom

Les fonctions qui font appel au DNS utilisent la structure `hostent` qui est définie par un « `#include <netdb.h>` » :

```
     struct hostent
     {
         char  *h_name;       /* nom officiel de l'hôte */
         char  **h_aliases;   /* liste d'alias */
         int   h_addrtype;    /* type d'adresse (tjs AF_INET */
                              /* pour IP v4 ou AF_INET6 pour */
                              /* IP v6) */
         int   h_length;      /* longueur de l'adresse */
         char  **h_addr_list; /* list d'addresses */
     };
```

Les tableaux `h_aliases` et `h_addr_list` ont comme dernier élément la valeur `NULL`.

La traduction IP vers nom se fait à l'aide de la fonction `gethostbyaddr()`, et la traduction nom vers adresse IP se fait avec la fonction `gethostbyname()` :

```
<<
SYNOPSIS
       #include <netdb.h>
       #include <sys/socket.h> /* pour avoir AF_INET */
       extern int h_errno;
       struct hostent *gethostbyname(const char *name);
       struct hostent *gethostbyaddr(const void *addr, \
                                     int len, int type);
DESCRIPTION
       La fonction gethostbyname() renvoie une structure de type
       hostent pour le nom d'hôte. La  chaîne name est soit un
       nom d'hôte, soit une adresse IPv4 en notation pointée
       standard, soit une adresse IPv6 avec la  notation
       points-virgules et points  (Cf RFC 1884 pour la description
       des adresses IPv6). Si name est une adresse IPv4 ou IPv6,
       aucune  recherche supplémentaire  n'a  lieu et
       gethostbyname() copie simplement la chaîne name dans le
       champ h_name et le champ équivalent struct in_addr dans
       le champ  h_addr_list[0]  de  la structure hostent renvoyée.
       Si name ne se termine pas par un point, et si la variable
       d'environnement HOSTALIASES est  configurée, le fichier
       d'alias pointé par HOSTALIASES sera d'abord parcouru à la
       recherche de nom (voyez hostname(7) pour  le  format  du
       fichier).  Le domaine courant et ses parents sont parcourus
       si name ne se termine pas par un point.

       La fonction gethostbyaddr() renvoie une structure du type
       hostent pour l'hôte  d'adresse addr.  Cette  adresse est de
       longueur len et du type type. Les types d'adresse valides
       sont AF_INET et AF_INET6.  L'argument adresse  de l'hôte est
```

un pointeur vers une structure de type dépendant du type de l'adresse, par exemple **struct in_addr *** (probablement obtenu via un appel à inet_addr()) pour une adresse de type **AF_INET**.
[...]

La structure hostent est définie ainsi dans <netdb.h> :

```
struct hostent {
   char    *h_name;     /* Nom officiel de l'hôte. */
   char   **h_aliases;  /* Liste d'alias. */
   int      h_addrtype; /* Type d'adrs. de l'hôte. */
   int      h_length;   /* Longueur de l'adresse. */
   char   **h_addr_list;/* Liste d'adresses. */
};
#define h_addr h_addr_list[0] /* pour compatibilité*/
```

Les membres de la structure hostent sont :
h_name Nom officiel de l'hôte.
h_aliases
Un tableau, terminé par un pointeur NULL, d'alternatives au nom officiel de l'hôte.
h_addrtype
Le type d'adresse : toujours AF_INET ou AF_INET6.
h_length
La longueur, en octets, de l'adresse.
h_addr_list
Un tableau, terminé par un pointeur NULL, d'adresses réseau pour l'hôte, avec l'ordre des octets du réseau.
h_addr La première adresse dans h_addr_list pour respecter la compatibilité ascendante.

VALEUR RENVOYÉE
Les fonctions gethostbyname() et gethostbyaddr() renvoient un pointeur vers la structure **hostent**, ou un pointeur NULL si une erreur se produit, auquel cas **h_errno** contient le code d'erreur. Lorsqu'elle n'est pas NULL, la valeur de retour peut pointer sur une donnée statique.

ERREURS
La variable **h_errno** peut prendre les valeurs suivantes :
HOST_NOT_FOUND
L'hôte indiqué est inconnu.
NO_ADDRESS ou **NO_DATA**
Le nom est valide mais ne possède pas d'adresse IP.
NO_RECOVERY
Une erreur fatale du serveur de noms est apparue.
TRY_AGAIN
Une erreur temporaire du serveur de noms est apparue, essayez un peu plus tard.

NOTES
Les fonctions gethostbyname() et gethostbyaddr() peuvent renvoyer des pointeurs sur des données statiques susceptibles d'être écrasées d'un appel à l'autre. Copier la structure struct hostent ne suffit pas car elle contient elle-même des pointeurs. Une copie en

```
       profondeur est indispensable.
>>
```

Pour IP version 4, les adresses sont stockées dans des structures de type `struct sockaddr_in`, qui est constituée des champs suivants :

```
struct sockaddr_in
{
    short sin_family;        /* AF_INET */
    u_short sin_port;        /* Num. de port en endian Internet */
    struct in_addr sin_addr; /* Adresse IP (en endian Internet) */
    char sin_zéro [8];       /* Bourrage */
};
struct in_addr
{
      u_long s_addr;
};
```

11.3.3 La résolution de service

Comme nous l'avons vu, chaque service est identifié par un numéro de port, choisi par convention. Sous Unix/Linux, les numéros de ports standards sont listés dans le fichier « `/etc/services` ».

Pour connaître le numéro de port associé à un service (et réciproquement), il existe deux fonctions : `getservbyname()` et `getservbyport()` :

```
<<
SYNOPSIS
       #include <netdb.h>
       struct servent *getservbyname (const char *name, \
                                      const char *proto);
       struct servent *getservbyport (int port, const char *proto);

DESCRIPTION
       La fonction getservbyname() renvoie une structure servent
       pour l'enregistrement du fichier /etc/services qui
       correspond au service nommé name et utilisant le protocole
       proto. Si proto est NULL, n'importe quel protocole sera
       accepté.

       La fonction getservbyport() renvoie une structure servent
       pour l'enregistrement correspondant au port indiqué
       (dans l'ordre des octets du réseau) et utilisant le
       protocole proto. Si proto est NULL, n'importe quel
       protocole sera accepté.

       La structure servent est définie dans <netdb.h> ainsi :
              struct servent {
                char  *s_name;    /* Nom officiel du service */
                char **s_aliases; /* Liste d'alias           */
                int    s_port;    /* Numéro de port          */
                char  *s_proto;   /* Protocole utilisé       */
              }

       Les membres de la structure servent sont :
       s_name Le nom officiel du service.
       s_aliases
```

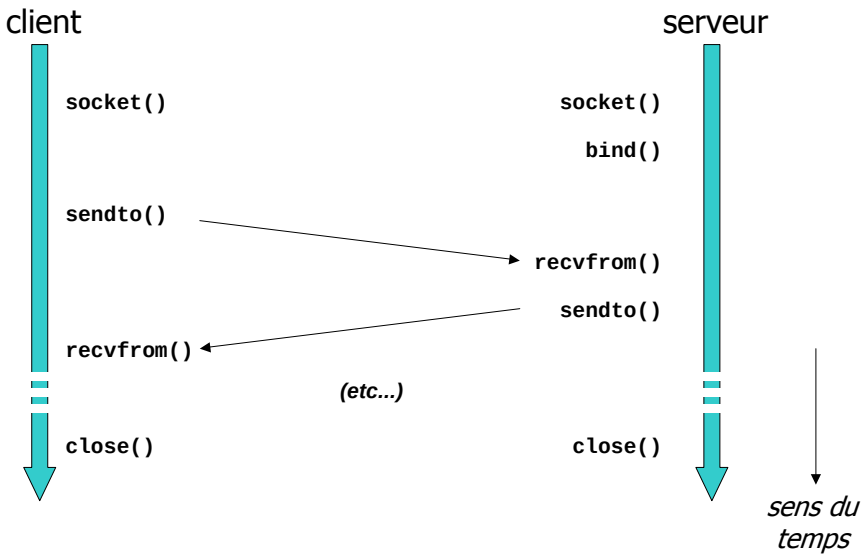
```
              Une liste terminée par zéro contenant d'autres noms
              utilisables pour le service.
       s_port Le numéro de port, donné dans l'ordre des octets du
              réseau.
       s_proto
              Le nom du protocole utilisé par ce service.

VALEUR RENVOYÉE
       Les  fonctions getservbyname() et getservbyport() renvoient
       une structure servent, ou un pointeur NULL si une erreur se
       produit, ou si la fin du fichier est atteinte.

>>
```

11.4 La communication par datagrammes

Coté serveur, la création d'un socket() est réalisée à l'aide de la fonction `socket()`. Puis, le socket est attaché à une adresse à l'aide de la fonction `bind()`. L'échange de datagrammes se fait à l'aide des fonctions `sendto()` et `recvfrom()`, qui permettent de spécifier le numéro de port sur lequel l'écoute est faite (avec le protocole UDP). Enfin, le socket est détruit à l'aide de la fonction `close()`.

Coté client, comme pour le serveur, la création d'un socket() est réalisée à l'aide de la fonction `socket()`.L'échange de datagrammes se fait à l'aide des fonctions `sendto()` et `recvfrom()`. Enfin, le socket est détruit à l'aide de la fonction `close()`.

Un échange client-serveur pour se schématiser ainsi :



Socket : communication par datagramme

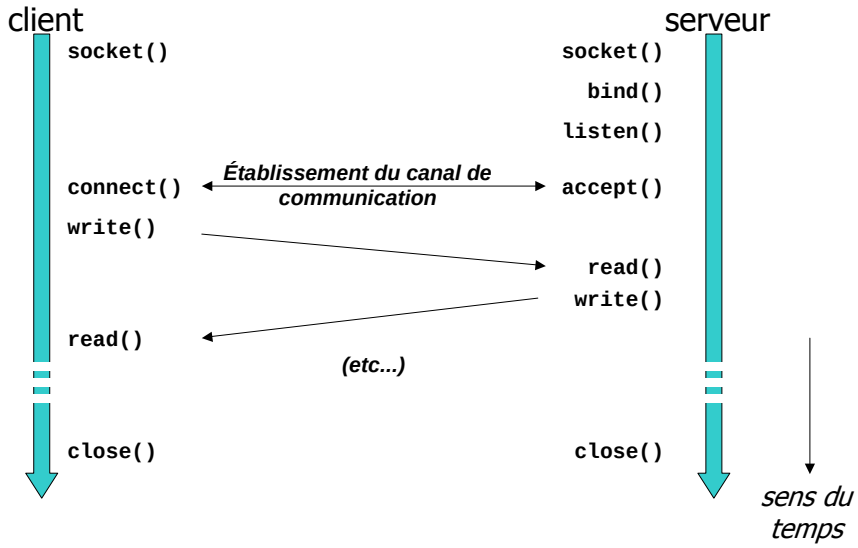
A noter qu'un socket est identifier par un descripteur, comme les pipes et des fichiers.

11.5 La communication en mode connecté

Coté serveur, le socket est créé par la fonction `socket()`. La fonction `bind()` associe le socket à une adresse. La fonction `listen()` met le processus en attente (création de la file d'attente des requêtes reçues). La fonction `accept()` permet d'accepter

l'établissement d'une connexion. Comme pour les fichiers, les fonctions `read()` et `write()` permettent d'envoyer ou de recevoir des flux de données dans la connexion. L'appel de `close()` ferme proprement le socket.

Coté client, le socket est créé par la fonction `socket()`.La fonction `connect()` permet d'établir une connexion avec le serveur. Les fonctions `read()` et `write()` permettent d'envoyer ou de recevoir des flux de données dans la connexion, et `close()` ferme proprement le socket.



Socket : communication par connexion

11.6 Manuel de l'API des socket

```
<<
SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>
       int socket(int domain, int type, int protocol);

DESCRIPTION
       socket() crée un point de communication, et renvoie un
       descripteur.

       Le  paramètre  domain  indique le domaine de communication
       pour le dialogue ; ceci sélectionne la famille de protocole
       à employer. Elles sont définies  dans le fichier
       <sys/socket.h>. Les formats actuellement proposés sont :

       Nom                Utilisation                      Page
       PF_UNIX, PF_LOCAL  Communication locale             unix(7)
       PF_INET            Protocoles Internet IPv4         ip(7)
       PF_INET6           Protocoles Internet IPv6         ipv6(7)
       PF_IPX             IPX - Protocoles Novell
       PF_NETLINK         Interface utilisateur noyau    netlink(7)
       PF_X25             Protocole ITU-T X.25 / ISO-8208  x25(7)
       PF_AX25            Protocole AX.25 radio amateur
       PF_ATMPVC          Accès direct ATM PVCs
```

PF_APPLETALK	Appletalk	ddp(7)
PF_PACKET	Interface paquet bas-niveau	packet(7)

La socket a le **type** indiqué, ce qui fixe la sémantique des communications. Les types définis actuellement sont :

SOCK_STREAM :
Support de dialogue garantissant l'intégrité, fournissant un flux de données binaires, et intégrant un mécanisme pour les transmissions de données hors-bande. [NdR : TCP]

SOCK_DGRAM :
Transmissions sans connexion, non garantie, de datagrammes de longueur maximale fixe. [NdR : UDP]

SOCK_SEQPACKET :
Dialogue garantissant l'intégrité, pour le transport de datagrammes de longueur fixe. Le lecteur doit lire le paquet de données complet à chaque appel système read.

SOCK_RAW :
Accès direct aux données réseau.

SOCK_RDM :
Transmission fiable de datagrammes, sans garantie de l'ordre de délivrance.

SOCK_PACKET :
Obsolète, à ne pas utiliser dans les programmes actuels.

Certains types de sockets peuvent ne pas être implémentés par toutes les familles de protocoles. Par exemple, **SOCK_SEQPACKET** n'est pas implémenté pour **AF_INET**.

Le protocole à utiliser sur la socket est indiqué par l'argument **protocol**. Normalement, il n'y a qu'un seul protocole par type de socket pour une famille donnée, auquel cas l'argument protocol peut être nul.
Néanmoins, rien ne s'oppose à ce que plusieurs protocoles existent, auquel cas il est nécessaire de le spécifier. Le numéro de protocole dépend du domaine de communication de la socket ; voir protocols(5).
Voir getprotoent(3) pour savoir comment associer un nom de protocole à un numéro.

Une socket de type **SOCK_STREAM** est un flux d'octets full-duplex, similaire aux tubes (pipes). Elle ne préserve pas les limites d'enregistrements. Une socket **SOCK_STREAM** doit être dans un état connecté avant que des données puissent y être lues ou écrites. Une connexion sur une autre socket est établie par l'appel système connect(2). Une fois connectée, les données y sont transmises par read(2) et write(2) ou par des variantes de send(2) et recv(2). Quand une session se termine, on referme la socket avec close(2). Les données hors-bande sont envoyées ou reçues en utilisant send(2) et recv(2).

Les protocoles de communication qui implémentent les sockets **SOCK_STREAM** garantissent qu'aucune donnée n'est perdue ou dupliquée. Si un bloc de données, pour lequel

le correspondant a suffisamment de place dans son tampon, n'est pas transmis correctement dans un délai raisonnable, la connexion est considérée comme inutilisable. Si l'option **SO_KEEPALIVE** est activée sur la socket, le protocole vérifie, d'une manière qui lui est spécifique, si le correspondant est toujours actif. Un signal **SIGPIPE** est envoyé au processus tentant d'écrire sur une socket inutilisable, forçant les programmes ne gérant pas ce signal à se terminer. Les sockets de type **SOCK_SEQPACKET** emploient les mêmes appels systèmes que celles de types **SOCK_STREAM**, à la différence que la fonction read(2) ne renverra que le nombre d'octets requis, et toute autre donnée restante dans le paquet sera éliminée. De plus, les frontières des messages seront préservées.

Les sockets de type **SOCK_DGRAM** ou **SOCK_RAW** permettent l'envoi de datagrammes aux correspondants indiqués dans l'appel système sendto(2). Les datagrammes sont généralement lus par la fonction recvfrom(2), qui fournit également l'adresse du correspondant.

Les sockets **SOCK_PACKET** sont obsolètes. Elles servent à recevoir les paquets bruts directement depuis le gestionnaire de périphérique. Utilisez plutôt packet(7).

Un appel à fcntl(2) avec l'argument **F_SETOWN** permet de préciser un processus ou un groupe de processus qui recevront un signal SIGURG lors de l'arrivée de données hors-bande, ou le signal SIGPIPE lorsqu'une connexion sur une socket **SOCK_STREAM** se termine inopinément. Cette fonction permet également de fixer le processus ou groupe de processus qui recevront une notification asynchrone des événements d'entrées-sorties par le signal SIGIO. L'utilisation de **F_SETOWN** est équivalent à un appel ioctl(2) avec l'argument **FIOSETOWN** ou **SIOCSPGRP**.

Lorsque le réseau indique une condition d'erreur au module du protocole (par exemple avec un message ICMP pour IP), un drapeau signale une erreur en attente sur la socket. L'opération suivante sur cette socket renverra ce code d'erreur. Pour certains protocoles, il est possible d'activer une file d'attente d'erreurs par socket. Pour plus de détails, voir **IP_RECVERR** dans ip(7).

Les opérations sur les sockets sont contrôlées par des options du niveau socket. Ces options sont définies dans <sys/socket.h>. Les fonctions setsockopt(2) et getsockopt(2) sont utilisées respectivement pour fixer ou lire les options.

VALEUR RENVOYÉE

socket() renvoie un descripteur référençant la socket créée en cas de réussite. En cas d'échec -1 est renvoyé, et errno contient le code d'erreur.

ERREURS

EACCES La création d'une socket avec le type et le protocole indiqués n'est pas autorisée.

EAFNOSUPPORT
L'implémentation ne supporte pas la famille d'adresses indiquée.
EINVAL Protocole inconnu, ou famille de protocole inexistante.
EMFILE La table des fichiers est pleine.
ENFILE La limite du nombre total de fichiers ouverts sur le système a été atteinte.
ENOBUFS ou **ENOMEM**
Pas suffisamment d'espace pour allouer les tampons nécessaires. La socket ne peut être créée tant que suffisamment de ressources ne sont pas libérées.
EPROTONOSUPPORT
Le type de protocole, ou le protocole lui-même n'est pas disponible dans ce domaine de communication.

/***/

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
int   bind(int sockfd, const  struct sockaddr  *my_addr, \
           socklen_t addrlen);
```

DESCRIPTION

bind() fournit à la socket **sockfd**, l'adresse locale **my_addr**. **my_addr** est longue de **addrlen** octets. Traditionnellement cette opération est appelée « affectation d'un nom à une socket ». Quand une socket est créée, par l'appel système socket(2), elle existe dans l'espace des noms mais n'a pas de nom assigné.

Il est normalement nécessaire d'affecter une adresse locale avec bind() avant qu'une socket SOCK_STREAM puisse recevoir des connexions (voir accept(2)).

Les règles d'affectation de nom varient suivant le domaine de communication. Consultez le manuel Linux section 7 pour de plus amples informations. Pour **AF_INET** voir ip(7), pour **AF_INET6** voir ipv6(7), pour **AF_UNIX** voir unix(7), pour **AF_APPLETALK** voir ddp(7), pour **AF_PACKET** voir packet(7), pour **AF_X25** voir x25(7) et pour **AF_NETLINK** voir netlink(7).

La structure réellement passée dans le paramètre **my_addr** dépend du domaine de communication. La structure sockaddr est définie comme :

```
  struct sockaddr {
      sa_family_t sa_family;
      char        sa_data[14];
  }
```

Le seul objet de cette structure est de transtyper le pointeur passé dans **my_addr** pour éviter les avertissements du compilateur.

VALEUR RENVOYÉE

L'appel renvoie 0 quand il réussit, ou -1 s'il échoue, auquel cas **errno** contient le code d'erreur.

ERREURS

EACCES L'adresse est protégée et l'utilisateur n'est pas le superutilisateur.

EADDRINUSE
L'adresse est déjà utilisée.

EBADF **sockfd** n'est pas un descripteur valide.

EINVAL La socket est déjà connectée.

ENOTSOCK
sockfd est un descripteur de fichier, pas une socket.

Les erreurs suivantes sont spécifiques aux sockets du domaine UNIX (**AF_UNIX**) :

EACCES L'accès à un élément du chemin est interdit. (Voir aussi path_resolution(2).)

EADDRNOTAVAIL
Une interface inexistante est demandée, ou bien l'adresse demandée n'est pas locale.

EFAULT **my_addr** pointe en dehors de l'espace d'adresse accessible.

EINVAL La longueur **addr_len** est fausse, ou la socket n'est pas de la famille **AF_UNIX**.

ELOOP **my_addr** contient des références circulaires (à travers un lien symbolique).

ENAMETOOLONG
my_addr est trop long.

ENOENT Le fichier n'existe pas.

ENOMEM Pas assez de mémoire pour le noyau.

ENOTDIR
Un élément du chemin d'accès n'est pas un répertoire.

EROFS L'inoeud se trouverait dans un système de fichiers en lecture seule.

/***/

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send(int s, const void *buf, size_t len, int flags);
ssize_t  sendto(int s, const void *buf, size_t len, \
                int flags, const struct sockaddr *to, \
                socklen_t tolen);
ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
```

DESCRIPTION

Les appels système send(), sendto() et sendmsg() permettent de transmettre un message à destination d'une autre socket.

L'appel send() ne peut être utilisé qu'avec les sockets connectées (ainsi, le destinataire visé est connu). La seule différence entre send() et write() est la présence de **flags**. Si **flags** est nul, send() est équivalent à write(). De plus, send(**s**,**buf**,**len**,**flags**) est équivalent à sendto(**s**,**buf**,**len**,**flags**,NULL,0).

Le paramètre **s** est le descripteur de fichier de la socket émettrice.

Si sendto() est utilisée sur une socket en mode connexion (**SOCK_STREAM**, **SOCK_SEQPACKET**), les paramètres **to** et **tolen** sont ignorés (et l'erreur **EISCONN** peut être retournée s'il n'y pas NULL ou 0), et l'erreur **ENOTCONN** est retournée lorsque la socket n'est pas vraiment connectée. Autrement, l'adresse de la cible est fournie par **to**, **tolen** spécifiant sa taille. Pour sendmsg(), l'adresse de la cible est fournie par **msg**.msg_name, **msg**.msg_namelen spécifiant sa taille.

Pour send() et sendto(), le message se trouve dans **buf** et a pour longueur **len**. Pour sendmsg(), le message est pointé par les éléments du tableau **msg**.msg_iov. L'appel sendmsg() permet également l'envoi de métadonnées (également appelées données de contrôle).

Si le message est trop long pour être transmis intégralement par le protocole sousjacent, l'erreur **EMSGSIZE** sera déclenchée et rien ne sera émis.

Aucune indication d'échec de distribution n'est fournie par send(). Seules les erreurs locales sont détectées, et indiquées par une valeur de retour -1.

Si la socket ne dispose pas de la place suffisante pour le message, alors send() va bloquer, à moins que la socket ait été configurée en mode d'entrées-sorties non-bloquantes auquel cas elle renverra **EAGAIN**. On peut utiliser l'appel système select(2) pour vérifier s'il est possible d'émettre des données.

Le paramètre flags est un OU bit à bit de zéro ou plusieurs des options suivantes :

MSG_CONFIRM (Depuis Linux 2.3)
: Indiquer à la couche liaison qu'une réponse correcte a été reçue du correspondant. Si la couche de liaison n'a pas cette confirmation, elle va ré-interroger régulièrement le voisinage (par exemple avec un ARP unicast). Seulement valide pour les sockets **SOCK_DGRAM** et **SOCK_RAW** et uniquement implémenté pour IPv4 et IPv6. Voir arp(7) pour plus de détails.

MSG_DONTROUTE
: Ne pas utiliser de passerelle pour transmettre le paquet, n'envoyer de données que vers les hôtes sur des réseaux directement connectés. Ceci n'est normalement employé que par les programmes de diagnostic ou de routage. Cette option n'est définie que pour les familles de protocoles employant le routage, pas les sockets par paquets.

MSG_DONTWAIT
: Activer les opérations non-bloquantes. Si l'opération devait bloquer, **EAGAIN** sera renvoyé (on peut aussi activer ce comportement avec l'option

O_NONBLOCK de la fonction **F_SETFL** de fcntl(2)).

MSG_EOR
Termine un enregistrement (lorsque cette notion est supportée, comme pour les sockets de type **SOCK_SEQPACKET**).

MSG_MORE (Depuis Linux 2.4.4)
L'appelant a d'autres données à envoyer. Cet attribut est utilisé avec les sockets TCP pour obtenir le même comportement qu'avec l'option socket TCP_CORK (voir tcp(7)), à la différence que cet attribut peut être positionné par appel.

Depuis Linux 2.6, cet attribut est également géré pour les sockets UDP et demande au noyau d'empaqueter toutes les données envoyées dans des appels avec cet attribut positionné dans un seul datagramme qui ne sera transmis que quand un appel sera effectué sans cet attribut. Voir aussi la description de l'option de socket UDP_CORK dans udp(7).

MSG_NOSIGNAL
Demande de ne pas envoyer de signal **SIGPIPE** d'erreur sur les sockets connectées lorsque le correspondant coupe la connexion. L'erreur **EPIPE** est toutefois renvoyée.

MSG_OOB
est utilisée pour émettre des données horsbande sur une socket qui l'autorise (par ex : de type **SOCK_STREAM**). Le protocole sousjacent doit également autoriser l'émission de données horsbande.

La définition de la structure msghdr se trouve ci-dessous. Voir recv(2) pour une description exacte de ses champs.

```
  struct msghdr {
      void         *msg_name;      /* adresse optionnelle */
      socklen_t     msg_namelen;   /* taille de l'adresse */
      struct iovec *msg_iov;    /* tableau scatter/gather */
      size_t        msg_iovlen;    /* # élém. ds msg_iov */
      void         *msg_control;   /* métadonnées  */
      socklen_t     msg_controllen;/* taille du tampon */
      int           msg_flags;     /* attributs msg reçu */
  };
```

On peut transmettre des informations de service en employant les membres msg_control et msg_controllen. La longueur maximale du tampon de service que le noyau peut gérer est limité par socket par la valeur net.core.optmem_max de sysctl(). Voir socket(7).

VALEUR RENVOYÉE
En cas d'envoi réussi, ces fonctions renvoient le nombre de caractères envoyés. En cas d'erreur, -1 est renvoyé, et **errno** contient le code d'erreur.

ERREURS
Voici les erreurs standards engendrées par la couche socket. Des erreurs supplémentaires peuvent être déclenchées

par les protocoles sous-jacents. Voir leurs pages de manuel respectives.

EACCES (Pour les sockets de domaine Unix qui sont identifiées par un nom de chemin) La permission d'écriture est refusée sur le fichier socket de destination ou la permission de parcours est refusée pour un des répertoires du chemin (voir path_resolution(2)).

EAGAIN ou **EWOULDBLOCK**
La socket est non-bloquante et l'opération demandée bloquerait.

EBADF Descripteur de socket invalide.

ECONNRESET
Connexion réinitialisée par le correspondant.

EDESTADDRREQ
La socket n'est pas en mode connexion et aucune adresse de correspondant n'a été positionnée.

EFAULT Un paramètre pointe en dehors de l'espace d'adressage accessible.

EINTR Un signal a été reçu avant que la moindre donnée n'ait été transmise.

EINVAL Un argument invalide a été transmis.

EISCONN
La socket en mode connexion est déjà connectée mais un destinataire a été spécifié. (Maintenant, soit cette erreur est retournée, soit la spécification du destinataire est ignorée.)

EMSGSIZE
Le type de socket nécessite une émission intégrale du message mais la taille de celui-ci ne le permet pas.

ENOBUFS
La file d'émission de l'interface réseau est pleine. Ceci indique généralement une panne de l'interface réseau, mais peut également être dû à un engorgement passager. Ceci ne doit pas se produire sous Linux, les paquets sont silencieusement éliminés.

ENOMEM Pas assez de mémoire pour le noyau.

ENOTCONN
La socket n'est pas connectée et aucune cible n'a été fournie.

ENOTSOCK
L'argument **s** n'est pas une socket.

EOPNOTSUPP
Au moins un bit de l'argument **flags** n'est pas approprié pour le type de socket.

EPIPE L'écriture a été terminée du côté local sur une socket orientée connexion. Dans ce cas, le processus recevra également un signal **SIGPIPE** sauf s'il a activé l'option **MSG_NOSIGNAL**.

/***/

SYNOPSIS

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

```
#include <sys/socket.h>
ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t recvfrom(int s, void *buf, size_t len, int flags, \
                 struct sockaddr *from, socklen_t *fromlen);
ssize_t recvmsg(int s, struct msghdr *msg, int flags);
```

DESCRIPTION

Les appels système recvfrom() et recvmsg() sont utilisés pour recevoir des messages depuis une socket, et peuvent servir sur une socket orientée connexion ou non.

Si **from** n'est pas NULL, et si le protocole sous-jacent fournit l'adresse de la source, celle-ci y est insérée. L'argument **fromlen** est un paramètre résultat, initialisé à la taille du tampon **from**, et modifié en retour pour indiquer la taille réelle de l'adresse enregistrée.

L'appel recv() est normalement utilisé sur une socket connectée (voir connect(2)) et est équivalent à recvfrom() avec un paramètre from NULL.

Ces trois routines renvoient la longueur du message si elles réussissent. Si un message est trop long pour tenir dans le tampon, les octets supplémentaires peuvent être abandonnés suivant le type de socket utilisé.

Si aucun message n'est disponible sur la socket, les fonctions de réception se mettent en attente, à moins que la socket soit non bloquante (voir fcntl(2)) auquel cas la valeur -1 est renvoyée, et **errno** est positionnée à EAGAIN. Les fonctions de réception renvoient normalement les données disponibles sans attendre d'avoir reçu le nombre exact réclamé.

Les appels système select(2) ou poll(2) peuvent permettre de déterminer si des données supplémentaires sont disponibles.

L'argument **flags** de l'appel recv() est constitué par un OU binaire entre une ou plusieurs des valeurs suivantes :

MSG_DONTWAIT

Activer les opérations non-bloquantes. Si l'opération devait bloquer, **EAGAIN** sera renvoyé (on peut aussi activer ce comportement avec l'option **O_NONBLOCK** de la fonction **F_SETFL** de fcntl(2)).

MSG_ERRQUEUE

Cet attribut demande que les erreurs soient reçues depuis la file d'erreur de la socket. Les erreurs sont transmises dans un message annexe dont le type dépend du protocole (IP_RECVERR pour IPv4). Il faut alors fournir un tampon de taille suffisante. Voir cmsg(3) et ip(7) pour plus de détails. Le contenu du paquet original qui a causé l'erreur est passé en tant que données normales dans msg_iovec. L'adresse de destination originale du datagramme ayant causé l'erreur est fournie dans msg_name.

Pour les erreurs locales, aucune adresse n'est passée (ceci peut être vérifié dans le membre cmsg_len de

cmsghdr). À la réception d'une erreur, **MSG_ERRQUEUE** est placé dans **msghdr**. Après réception d'une erreur, l'erreur en attente sur la socket est régénérée en fonction de la prochaine erreur dans la file, et sera transmise lors de l'opération suivante sur la socket.

L'erreur est contenue dans une structure **sock_extended_err** :

```
#define SO_EE_ORIGIN_NONE    0
#define SO_EE_ORIGIN_LOCAL   1
#define SO_EE_ORIGIN_ICMP    2
#define SO_EE_ORIGIN_ICMP6   3

struct sock_extended_err
{
    u_int32_t ee_errno;  /* numéro d'erreur  */
    u_int8_t  ee_origin; /* origine de l'erreur */
    u_int8_t  ee_type;   /* type */
    u_int8_t  ee_code;   /* code  */
    u_int8_t  ee_pad;    /* remplissage */
    u_int32_t ee_info;   /* données suplém. */
    u_int32_t ee_data;   /* autres données */
    /* des données supplément. peuvent suivre */
};
struct sockaddr *SO_EE_OFFENDER(struct \
                                sock_extended_err *);
```

ee_errno contient le code errno de l'erreur en file. ee_origin est le code d'origine de l'erreur. Les autres champs sont spécifiques au protocole. La macro SOCK_EE_OFFENDER renvoie un pointeur sur l'adresse de l'objet réseau ayant déclenché l'erreur, à partir d'un pointeur sur le message. Si l'adresse n'est pas connue, le membre sa_family de la structure sockaddr contient **AF_UNSPEC** et les autres champs de la structure sockaddr sont indéfinis. Le contenu du paquet ayant déclenché l'erreur est transmis en données normales.

Pour les erreurs locales, aucune adresse n'est passée (ceci peut être vérifié dans le membre cmsg_len de **cmsghdr**). À la réception d'une erreur, **MSG_ERRQUEUE** est placé dans **msghdr**. Après réception d'une erreur, l'erreur en attente sur la socket est régénérée en fonction de la prochaine erreur dans la file, et sera transmise lors de l'opération suivante sur la socket.

MSG_OOB

Cette option permet la lecture des données hors-bande qui ne seraient autrement pas placées dans le flux de données normales.
Certains protocoles placent ces données hors-bande en tête de la file normale, et cette option n'a pas lieu d'être dans ce cas.

MSG_PEEK

Cette option permet de lire les données en attente dans la file sans les enlever de cette file. Ainsi une lecture ultérieure renverra à nouveau les mêmes données.

MSG_TRUNC

Renvoyer la longueur réelle du paquet, même s'il était plus long que le tampon transmis. Valide uniquement pour les sockets paquets.

MSG_WAITALL

Cette option demande que l'opération de lecture soit bloquée jusqu'à ce que la requête complète soit satisfaite. Toutefois, la lecture peut renvoyer quand même moins de données que prévu si un signal est reçu, ou si une erreur ou une déconnexion se produisent.

L'appel recvmsg() utilise une structure **msghdr** pour minimiser le nombre de paramètres à fournir directement. Cette structure a la forme suivante, définie dans <sys/socket.h> :

```
  struct msghdr {
      void         *msg_name;   /* adresse optionnelle */
      socklen_t     msg_namelen;/* taille de l'adresse */
      struct iovec *msg_iov;    /* tableau scatter/gather */
      size_t        msg_iovlen; /* # éléments ds msg_iov */
      void         *msg_control;/* métadonnées */
      socklen_t     msg_controllen; /* taille tampon */
      int           msg_flags;/* attributs message reçu */
  };
```

Ici msg_name et msg_namelen spécifient l'adresse d'origine si la socket n'est pas connectée ; msg_name peut être un pointeur nul si le nom n'est pas nécessaire. msg_iov et msg_iovlen décrivent les tampons de réception comme décrit dans readv(2). msg_control, de longueur msg_controllen, pointe sur un tampon utilisé pour les autres messages relatifs au protocole, ou à d'autres données annexes. Lorsqu'on invoque **recvmsg**, msg_controllen doit contenir la longueur disponible dans le tampon msg_control ; au retour il contiendra la longueur de la séquence de message de contrôle.

Les messages ont la forme

```
  struct cmsghdr {
      socklen_t cmsg_len;  /* nombre d'octets de données, */
                           /* y compris l'en-tête */
      int       cmsg_level;/* protocole d'origine */
      int       cmsg_type; /* type dépendant du protocole */
  /* suivi de
      u_char    cmsg_data[]; */
  };
```

Les données de service ne doivent être manipulées qu'avec les macros de cmsg(3).

À titre d'exemple, Linux utilise ce mécanisme pour transmettre des erreurs étendues, des options IP, ou des

descripteurs de fichiers sur des sockets Unix.

Le champ msg_flags du msghdr est rempli au retour de recvmsg(). Il peut contenir plusieurs attributs :

MSG_EOR
indique une fin d'enregistrement, les données reçues terminent un enregistrement (utilisé généralement avec les sockets du type **SOCK_SEQPACKET**).

MSG_TRUNC
indique que la portion finale du datagramme a été abandonnée car le datagramme était trop long pour le tampon fourni.

MSG_CTRUNC
indique que des données de contrôle ont été abandonnées à cause d'un manque de place dans le tampon de données annexes.

MSG_OOB
indique que des données hors-bande ont été reçues.

MSG_ERRQUEUE
indique qu'aucune donnée n'a été reçue, sauf une erreur étendue depuis la file d'erreurs.

VALEUR RENVOYÉE

Ces fonctions renvoient le nombre d'octets reçus si elles réussissent, ou -1 si elles échouent. La valeur de retour sera 0 si le pair a effectué un arrêt normal.

ERREURS

Il y a des erreurs standards déclenchées par le niveau socket, et des erreurs supplémentaires spécifiques aux protocoles. Voyez leurs pages de manuel.

EAGAIN La socket est non-bloquante et aucune donnée n'est disponible, ou un délai de timeout a été indiqué, et il a expiré sans que l'on ait reçu quoi que ce soit.

EBADF L'argument **s** n'est pas un descripteur valide.

ECONNREFUSED
Un hôte distant a refusé la connexion réseau (généralement parce qu'il n'offre pas le service demandé).

EFAULT Un tampon pointe en dehors de l'espace d'adressage accessible.

EINTR Un signal a interrompu la lecture avant que des données soient disponibles.

EINVAL Un argument invalide a été transmis.

ENOMEM Pas assez de mémoire pour recvmsg().

ENOTCONN
La socket est associée à un protocole orienté connexion et n'a pas encore été connectée (voir connect(2) et accept(2)).

ENOTSOCK
L'argument **s** ne correspond pas à une socket.

/***/

SYNOPSIS

```
       #include <sys/socket.h>
       int listen(int sockfd, int backlog);

DESCRIPTION
       Pour accepter des connexions, une socket est d'abord
       créée avec socket(2), puis le désir d'accepter des
       connexions entrantes, et la limite de la file
       d'entrée sont indiqués avec listen(), ensuite les
       connexions seront acceptées avec accept(2). L'appel
       système listen() s'applique seulement aux sockets de type
       SOCK_STREAM ou SOCK_SEQPACKET.

       Le paramètre backlog définit une longueur maximale pour la
       file des connexions en attente. Si une nouvelle
       connexion arrive alors que la file est pleine, le client
       reçoit une erreur indiquant ECONNREFUSED, ou, si le
       protocole sousjacent supporte les retransmissions, la
       requête peut être ignorée afin qu'un nouvel essai réussisse.

VALEUR RENVOYÉE
       L'appel renvoie 0 quand il réussit, ou -1 s'il échoue,
       auquel cas errno contient le code d'erreur.

ERREURS
       EADDRINUSE
              Une autre socket est déjà à l'écoute sur le même
              port.
       EBADF  sockfd n'est pas un descripteur valide.
       ENOTSOCK
              L'argument sockfd n'est pas une socket.
       EOPNOTSUPP
              Le type de socket ne supporte pas l'appel système
              listen().

/*************************************************************/

SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>
       int accept(int sockfd, struct sockaddr *addr, \
       socklen_t *addrlen);

DESCRIPTION
       L'appel système accept() est employé avec les sockets
       utilisant un protocole en mode connecté (SOCK_STREAM,
       SOCK_SEQPACKET). Il extrait la première connexion de la
       file des connexions en attente, crée une nouvelle socket et
       alloue pour cette socket un nouveau descripteur de
       fichier qu'il renvoie. La nouvelle socket n'est pas en
       état d'écoute.
       La socket originale sockfd n'est pas modifiée par l'appel
       système.

       L'argument sockfd est une socket qui a été créée avec
       la fonction socket(2), attachée à une adresse avec
       bind(2), et attend des connexions après un appel listen(2).

       L'argument addr est un pointeur sur une structure sockaddr.
```

La structure sera remplie avec l'adresse du correspondant se connectant, telle qu'elle est connue par la couche de communication. Le format exact du paramètre addr dépend du domaine dans lequel la communication s'établit (voir socket(2) et la page de manuel correspondant au protocole). L'argument **addrlen** est un paramètre-résultat: il doit contenir initialement la taille de la structure pointée par addr, et est renseigné au retour par la longueur réelle (en octet) de l'adresse remplie. Quand **addr** vaut NULL, rien n'est rempli.

S'il n'y a pas de connexion en attente dans la file, et si la socket n'est pas marquée comme non-bloquante, accept() se met en attente d'une connexion. Si la socket est non-bloquante, et qu'aucune connexion n'est présente dans la file, accept() retourne une erreur EAGAIN.

Pour être prévenu de l'arrivée d'une connexion sur une socket on peut utiliser select(2) ou poll(2). Un événement « lecture » sera délivré lorsqu'une tentative de connexion aura lieu, et on pourra alors appeler accept() pour la valider. Autrement, on peut configurer la socket pour qu'elle envoie un signal **SIGIO** lorsqu'une activité la concernant se produit, voir socket(7) pour plus de détails.

Pour certains protocoles nécessitant une confirmation explicite, comme DECNet, accept() peut être considéré comme extrayant simplement la connexion suivante de la file, sans demander de confirmation. On peut effectuer la confirmation par une simple lecture ou écriture sur le nouveau descripteur, et le rejet en fermant la nouvelle socket.

VALEUR RENVOYÉE

Si elle réussit, la fonction accept() renvoie un entier positif ou nul, qui est un descripteur pour la socket acceptée. En cas d'erreur, elle renvoie -1 et remplit **errno** avec le code d'erreur.

GESTION DES ERREURS

Sous Linux, accept() renvoie les erreurs réseau déjà en attente sur la socket comme une erreur de l'appel système. Ce comportement diffère d'autres implémentations des sockets BSD. Pour un comportement fiable, une application doit détecter les erreurs réseau définies par le protocole après le accept() et les traiter comme des erreurs **EAGAIN**, en réitérant le mécanisme. Dans le cas de TCP/IP, ces erreurs sont **ENETDOWN**, **EPROTO**, **ENOPROTOOPT**, **EHOSTDOWN**, **ENONET**, **EHOSTUNREACH**, **EOPNOTSUPP**, et **ENETUNREACH**.

>>

12 Les problématiques de synchronisation

12.1 Problématique

Les systèmes multiprogrammés et temps-partagé autorisent la présence de plusieurs processus en même temps dans le système. Il en résulte une nouvelle situation qui n'existait pas auparavant dans les systèmes monoprogrammés. En effet, il peut arriver que plusieurs programmes aient besoin des mêmes ***ressources*** :

- mémoire centrale,
- CPU,
- périphériques,
- etc.

De même, certains de ces processus coopèrent ensembles pour satisfaire les besoins d'une seule application. Cette coopération passe très souvent par l'utilisation et le partage de données communes (Cf. exemple avec les régions de mémoire partagée – *shared memory* – vu dans le chapitre précédent).

Le partage de ressources et l'utilisation de données communes posent de sérieux problèmes. Le système d'exploitation doit proposer des techniques et des outils permettant de contrôler l'accès et l'utilisation de ces ressources. Ces outils et techniques doivent garantir l'absence de blocage et un partage équitable.

Les systèmes d'exploitation attribuent deux propriétés aux ressources :

- leur état (libre ou occupé),
- leur nombre de points d'accès (nombre de processus qui peuvent y accéder en même temps).

Une ressource est dite ***critique*** quand elle ne peut être utilisée que par un seul processus à la fois.

Dès lors, dans les systèmes multiprogrammés ou temps-partagé, l'utilisation d'une ressource par un processus se fait en trois étapes :

1. l'***allocation*** de la ressource par le processus,
2. utilisation de la ressource,
3. la ***libération*** de la ressource, la rendant disponible pour les autres programmes.

Pour garantir une bonne utilisation des ressources, plusieurs schémas de synchronisation classiques sont proposés, schémas que nous allons détailler dans les prochains chapitres :

- l'exclusion mutuelle,
- l'allocation de ressources,
- les lecteurs-rédacteurs,
- les producteurs-consommateurs.

12.2 L'exclusion mutuelle

12.2.1 Exemple de problématique

Pour comprendre l'exclusion mutuelle, la littérature utilise souvent un problème tout simple : un programme de vente de billets à un spectacle. Au départ, il y a N billets de disponibles. L'objectif est de programmer une fonction de réservation qui réserve une place de

spectacle, et qui retourne la valeur 1 si la réservation s'est déroulée normalement, et 0 s'il n'y a plus de place disponible. En C, le programme pourrait s'écrire :

```
int   nb_place=N;
/* bla bla bla... */
int   reservation(void)
{
      int   ret=0;
      if (nb_place > 0)   /* !!! */
      {
            /* Il reste au moins une place libre, */
            /* aussi, on réserve une place */
            nb_place-- ;
            ret = 1;
      }
      return(ret);
}
```

Dans un système monotâche, ce programme fonctionnerait parfaitement. Mais, dans un système multiprogrammé, voici ce qu'il peut se produire :

1. le programme se déroule normalement, jusqu'à ce qu'il ne reste plus qu'une seule place ;
2. un client C1 souhaite réserver. Il appelle la fonction `reservation()`. Le registre de compteur ordinal se branche donc à l'adresse de cette fonction `reservation()`. Il effectue le test « `if (nb_place > 0)` ». Evidemment, comme il reste une place, le résultat de ce test est « VRAI » ;
3. puis, parce que le processus du client C1 a consommé tout son temps CPU dans le cycle d'ordonnancement, il y a préemption ;
4. le *scheduler* élit un processus client C2 qui, lui aussi, a besoin de réserver une place de spectacle. Il effectue lui aussi un appel à la fonction `reservation()`. Comme il reste encore une place, cette fonction lui reservera cette dernière place, et retournera la valeur 1 ;
5. au tour suivant, le *scheduler* redonnera la main au processus C1 qui, rappelons nous, en était au point « `/* !!! */` » dans le code ci-dessus. Le résultat du test avait été vrai, aussi, il continue d'exécuter le code correspondant à « *then* ». La variable `nb_place` est décrémentée (elle vaut alors la valeur -1 !!!), et la fonction retourne aussi la valeur 1. C1 et C2 ont réservé en même temps la dernière place, ce qui n'est pas acceptable.

Evidemment, nous pourrions penser que cette situation de préemption au point « `/* !!! */` » couplé au fait qu'un autre processus ait besoin aussi de réserver une place dans le même tour d'élection de l'ordonnanceur est exceptionnelle. Pas si sûr. Pensez au programme de réservation de la billèterie du dernier concert de U2 (100'000 billets vendus en 15 minutes)... De plus, même exceptionnelle, la situation peut néanmoins engendrer des conséquences catastrophiques (avions qui s'écrasent, équipement radiologique qui irradie le patient, etc.).

12.2.2 Recherche de solutions à l'exclusion mutuelle

En fait, dans l'exemple précédent, nous avons identifié le fait que la variable `nb_place` ne devait être manipulée que par un et un seul processus à la fois. Il s'agit d'une ***ressource critique***.

Dans le programme ci-dessus, l'ensemble de toutes les instructions qui se suivent, et qui ont besoin d'accéder à cette ressource critique s'appelle une ***section critique***. Dans l'exemple, il s'agit du code :

```
int  reservation(void)
{
     int  ret=0;
     /* début de section critique */
     if (nb_place > 0)
     {
          nb_place--;
          ret = 1;
     }
     /* fin de la section critique */
     return(ret);
}
```

Demander à un système d'exploitation de gérer une section critique doit l'obliger à assurer 3 propriétés :

- assurer l'*exclusion mutuelle* (lorsqu'un processus utilise une ressource critique, un autre processus ne peut l'utiliser),
- assurer une *attente bornée* aux autres processus qui souhaitent utiliser une ressource critique déjà assignée (chaque processus doit pouvoir accéder à la ressource en un temps maximum),
- assurer la propriété de *bon déroulement* : lorsqu'un processus libère une ressource et que plusieurs autres processus sont en attente de cette ressource, l'élection du processus comme étant celui qui s'approprie la ressource se fait en ne considérant comme critère que la liste des candidats potentiels. Aucun événement extérieur ne peut influer ou bloquer ce choix.

Comme nous l'avons vu en introduction, l'accès à une ressource critique doit se faire en en verrouillant sont accès (en se l'allouant) avec un mécanisme appelé « *prélude* », et doit le libérer avec un mécanisme appelé « *postlude* ».

La première solution consisterait à interdir toute interruption durant la section critique :

```
int  reservation(void)
{
     int  ret=0;
     disable_interrupt; /* masque les interruptions */
     if (nb_place > 0)
     {
          nb_place--;
          ret = 1;
     }
     enable_interrupt; /* réactive les interruptions */
     return(ret);
}
```

L'inconvénient de cette solution est que tous les processus sont bloqués, même ceux qui n'ont pas besoin de cette ressource. Cette solution est réservée uniquement aux parties de code sensibles du noyau (comme par exemple, la manipulation des files d'attente de l'ordonnanceur, ou la protection des buffeur de cache).

Une autre solution (appelée « *Test & Set* »), qui vaut une note éliminatoire à l'examen, consiste à obliger le processus souhaitant mettre un verrou à une ressource critique à faire du « *polling* » en attendant que la ressource se libère.

Pour ce, il faut créer une fonction « `Test_and_Set()` » de la façon suivante :

```
int  Test_and_Set(int *verrou)
{
     disable_interrupt;
     int  ret = *verrou;
     *verrou = 1;
     return(ret);
```

```
        enable_interrupt;
    }
```

Pour chaque ressource, il faut définir une variable globale « `int cadena = 0;` ». Le prélude devient alors :

```
    while (Test_and_Set(&cadena));
```

A la fin de la section critique, le postlude devient :

```
    cadena = 0;
```

Le premier processus recevra de la fonction « `Test_and_Set()` » une valeur 0. Tous les autres recevront une valeur 1, tant que le processus ayant verrouillé la ressource ne positionnera pas la variable faisant office de verrou à 0. Cette solution est inacceptable : si plusieurs processus sont en attente d'une ressource verrouillée, la boucle de polling occupe le CPU, ce qui fait s'écrouler les performances de la machine.

D'autres algorithmes n'utilisant pas de propriétés matérielles du CPU permettent de résoudre ce problème pour N processus. Néanmoins, ici aussi, les processus exclus doivent faire du *polling* avant d'entrer en section critique (Cf. algorithme de Peterson).

12.3 L'allocation de ressources – les sémaphores

12.3.1 Principe

Comme nous l'avons vu, tout le problème vient du fait que si l'état de réservation d'une ressource critique est stockée dans une variable, il peut y avoir une interruption (donc préemption) entre le moment où on effectue le test « *la variable indique-t-elle que la ressource est disponible* » et le moment ou on « *assigne à la variable une valeur indiquant que la ressource est assignée en exclusivité à un processus* ».

Or, les CPU modernes permettent de réaliser ces deux opérations de façon unitaire, sans qu'une interruption puisse perturber le résultat. Ainsi, les systèmes d'exploitation modernes utilisent cette opération pour proposer un mécanisme d'exclusion mutuelle : ***le sémaphore***. En voici le principe.

Les trois opérations liées aux sémaphores sont « **`Init`** », « **`P`** » et « **`V`** » (P et V signifient en néerlandais Proberen – tester – et Verhogen – incrémenter –, ce qui se traduirait en français par « *Puis-je ?* » et « *Vas-y !* »). La valeur d'un sémaphore est le nombre d'unités de ressource (exemple : imprimantes...) libres. S'il n'y a qu'une ressource, un sémaphore devient à système numérique binaire avec les valeurs 0 ou 1. Explication de ces trois fonctions :

- l'opération **`P`** est en attente jusqu'à ce qu'une ressource soit disponible, ressource qui sera immédiatement allouée au processus courant ;
- **`V`** est l'opération inverse. Elle rend simplement une ressource disponible à nouveau après que le processus a terminé de l'utiliser ;
- **`Init`** est seulement utilisé une (et une seule fois) pour initialiser le sémaphore.

Les opérations **`P`** et **`V`** sont indivisibles ; les différentes opérations ne peuvent pas être exécutées plusieurs fois de manière concurrente.

Dans la littérature anglaise, les opérations **`V`** et **`P`** sont quelques fois appelées respectivement ***up*** et ***down***. En conception logicielle, elles sont parfois appelées ***release*** et ***take***.

Un sémaphore a généralement une file de processus associée (file de type FIFO). Si un processus exécute l'opération **`P`** sur un sémaphore qui a la valeur zéro, le processus est ajouté à la file d'attente du sémaphore (le processus ne consomme pas de CPU, et la désactivation de l'attente est faite communément par l'ordonnanceur : quand un autre

processus incrémente le sémaphore en exécutant l'opération **V**, et qu'il y a des processus dans la file, l'un d'eux est retiré de la file et reprend la suite de son exécution).

Voici les algorithmes (ça n'est pas du C propre) de ces 3 opérations :

- Opération Init(sem, val) :

```
void Init(semaphore sem, int val)
{
      disable_interrupt;
      sem.K = val;
      sem.L = NULL;
      enable_interrupt;
}
```

- Opération P(sem) :

```
void P(semaphore sem)
{
      disable_interrupt;
      sem.K--;
      if (sem.K < 0)
      {
            sem.L.suivant = processus_courant;
            processus_courant.state = bloque;
            reordonnancement = vrai;
      }
      enable_interrupt;
}
```

- Opération V(sem) :

```
void V(semaphore sem)
{
      disable_interrupt;
      sem.K++;
      if (sem.K <= 0) /* au moins un processus en attente */
      {
            processus_reveille = sem.L.tete;
            processus_reveille.state = prêt;
            reordonnancement = vrai;
      }
      enable_interrupt;
}
```

Dans cet algorithme :

- `sem.K` est un compteur, qui, s'il est positif, indique le nombre d'accès disponible (i.e. nombre de fois ou un processus peut appeler `P(sem)` sans se retrouver bloqué). S'il est négatif, il indique le nombre de processus en attente de libération de la ressource ;
- `sem.L` est la file des processus en attente.

Avec les sémaphores, notre problème de réservation de places de spectacle devient tout simple :

```
int        nb_place=N;
semaphore  sem_places;
Init(sem_places, 1) ;
/* bla bla bla... */
int  reservation(void)
{
      int  ret=0;
      P(sem_places) ; /* prélude, entre en section critique */
```

```
        if (nb_place > 0)
        {
             /* Il reste au moins une place libre, */
             /* aussi, on réserve une place */
             nb_place-- ;
             ret = 1;
        }
        V(sem_places); /* postlude, fin de section critique */
        return(ret);
   }
```

12.3.2 Le danger des sémaphores : l'interblocage

Supposons un processus A dont le code est le suivant :

```
...
P(Sem1); /* !!! */
P(Sem2);
/* section critique utilisant les ressources */
/* verrouillées par les sémaphores Sem1 et Sem2 */
Blablabla...
V(Sem2);
V(Sem1);
```

Et un processus B dont le code est :

```
...
P(Sem2);
P(Sem1);
/* section critique utilisant les ressources */
/* verrouillées par les sémaphores Sem1 et Sem2 */
Blablabla...
V(Sem1);
V(Sem2);
```

Supposons que `Sem1` et `Sem2` soient des sémaphores initialisés à la valeur 1, et que ce soit le processus A qui ait le CPU, et que son exécution se termine juste après l'appel « `P(Sem1);` » et juste avant « `P(Sem2);` » (i.e. qu'il y a préemption au point « `/* !!! */` »).

Le processus B est alors exécuté. Il fait un « `P(Sem2);` » qui lui assure l'exclusivité sur la ressource 2. Puis, il fait un « `P(Sem1);` » qui est bloquant (le processus A a déjà verrouillé la ressource).

L'ordonnanceur élit le processus A, qui fait un « `P(Sem2);` » bloquant (en effet, la ressource 2 est verrouillée par le processus B). Pour résumer, nous nous retrouvons avec la situation suivante :

- le processus A s'est assuré l'exclusivité de la ressource n°1, et est bloqué en attente de la ressource n°2,
- et le processus B s'est assuré l'exclusivité de la ressource n°2, et est bloqué en attente de la ressource n°1.

Il s'agit d'un cas d'interblocage (on parle aussi d'étreinte fatale ou d'étreinte mortelle), où aucun des processus n'obtiendra jamais ce qu'il désire.

Une solution simple dans le cas présent consiste programmer les processus pour qu'ils fassent les réservations toujours dans le même ordre (c'est ce qui est fait au sein du noyau Linux).

De façon générale, il y a interblocage quand :

- il existe une exclusion mutuelle (au moins une ressource doit être dans un mode non partageable),
- occupation et attente : au moins un processus s'est accaparé une ressource, et attend une autre ressource,
- il n'y a pas de réquisition : les ressources sont libérées uniquement par la bonne volonté des processus,
- il existe un cycle dans le graphe d'attente entre au moins deux processus.

Un graphe d'attente est un graphe où sont représentés les processus et les ressources comme étant des nœuds. Des flèches symbolisent les situations « *bloque* » et « *attend* ».

12.3.3 Les sémaphores sous Linux

Comme nous l'avons vu dans les chapitres précédents, les sémaphores sont implémentés sous Linux sous la forme d'IPC.

De plus, les IPC de sémaphores introduisent une nouvelle notion en plus des fonctions `P()` et `V()` : il s'agit de la fonction `ATT()`, qui bloque un processus dans l'attente que le sémaphore soit nul (mais sans décrémenter la valeur du sémaphore).

La création d'un ensemble de sémaphore se fait à l'aide de la fonction `semget()` :

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

DESCRIPTION
       Cette fonction retourne l'identifiant de l'ensemble de
       sémaphores associé à la valeur de clé key. Un nouvel
       ensemble contenant nsems sémaphores est créé si key a la
       valeur IPC_PRIVATE ou si aucun ensemble n'est associé à key,
       et si l'option IPC_CREAT est présente dans semflg.

       Si semflg contient à la fois IPC_CREAT et IPC_EXCL et qu'un
       ensemble de sémaphores existe déjà pour la clé key, semget()
       échoue et  errno  vaut EEXIST (ceci est analogue à l'effet
       de la combinaison O_CREAT | O_EXCL pour open(2).)

       Pendant la création, les 9 bits de poids faibles de
       l'argument  semflg définissent les permissions d'accès (pour
       le propriétaire, le groupe et les autres) au jeu de
       sémaphores, en utilisant le  même  format  et  la même
       signification que les droits d'accès dans open(2). Les
       permissions d'exécution ne sont pas utilisées par le
       système, et  pour  un  jeu  de sémaphores,  l'autorisation
       d'écriture signifie autorisation de modification.

       Les valeurs des sémaphores dans un ensemble qui vient d'être
       créé  sont indéterminées (POSIX.1-2001 est explicite sur ce
       point). Même si Linux, comme de nombreuses autres
       implémentations, initialise les valeurs  des sémaphores  à
       0, une application portable ne peut pas compter sur cette
       initialisation : elle doit initialiser explicitement les
       sémaphores  à la valeur souhaitée.
```

Durant la création, la structure de données semid_ds (voir semctl(2)) contrôlant le jeu de sémaphores est initialisée ainsi :

sem_perm.cuid et sem_perm.uid contiennent l'UID effectif du processus appelant.

sem_perm.cgid et sem_perm.gid contiennent le GID effectif du processus appelant.

Les 9 bits de poids faibles de sem_perm.mode contiennent les 9 bits de poids faibles de semflg.

sem_nsems reçoit la valeur nsems.

sem_otime est mis à 0.

sem_ctime est rempli avec l'heure actuelle.

L'argument **nsems** peut valoir 0 (ignore) si l'appel système n'est pas une création d'ensemble de sémaphores. Autrement **nsems** doit être supérieur à 0 et inférieur ou égal au nombre maximal de sémaphores par ensemble (**SEMMSL**).

Si le jeu de sémaphores existe déjà, les permissions d'accès sont contrôlées.

VALEUR RENVOYÉE

Si l'appel réussit, il renvoie l'identifiant de l'ensemble (un entier positif), sinon il renvoie -1 et **errno** contient le code d'erreur.

ERREURS

En cas d'erreur, errno prend l'une des valeurs suivantes :

EACCES Le jeu de sémaphore associé à **key** existe, mais le processus n'a aucun droit d'accès sur lui et n'a pas la capacité **CAP_IPC_OWNER**.

EEXIST Le jeu de sémaphore associé a **key** existe mais l'argument **semflg** précise à la fois **IPC_CREAT** et **IPC_EXCL**.

EINVAL **nsems** est inférieur à zéro ou supérieur à la limite sur le nombre de sémaphores par ensemble, (**SEMMSL**), ou l'ensemble de sémaphores identifié par **key** existe déjà, et **nsems** est plus grand que le nombre de sémaphores par ensemble.

ENOENT Aucun jeu de sémaphore associé a **key** n'existe et l'argument **semflg** ne précise pas **IPC_CREAT**.

ENOMEM Pas assez de mémoire pour créer les structures nécessaires.

ENOSPC Le nombre maximal de jeux de sémaphores sur le système (**SEMMNI**) est atteint, ou le nombre maximal de sémaphores sur le système est atteint (**SEMMNS**).

>>

L'initialisation d'un sémaphore sem à une valeur v0 se fait via la fonction semctl() :

```
semctl(sem, 0, SETVAL, v0);
```

La destruction d'un semaphore sem se fait par l'appel de cette même fonction semctl() :

```
semctl(sem, 0, IPC_RMID, 0);
```

Lire le manuel de cette function pour plus de details. Enfin, les fonctions P(), V(), et ATT() d'un sémaphore se font à l'aide de la fonction semop() :

```
<<
SYNOPSIS
       #include <sys/types.h>
       #include <sys/ipc.h>
       #include <sys/sem.h>
       int semop(int semid, struct sembuf *sops, unsigned nsops);
       int semtimedop(int  semid, struct sembuf *sops, \
                      unsigned nsops, struct timespec *timeout);

DESCRIPTION
       Chaque sémaphore dans un ensemble de sémaphores se  voit
       associer  les valeurs suivantes :
           unsigned short  semval; /* valeur du sémaphore   */
           unsigned short  semzcnt;/* # Attente pour zéro   */
           unsigned short  semncnt;/* # Attente d'incrément */
           pid_t           sempid; /* dernier processus agissant */

       La fonction semop() effectue des opérations sur les membres
       de l'ensemble de sémaphores identifié par semid. Chacun des
       nsops  éléments  dans le  tableau  pointé  par  sops indique
       une opération à effectuer sur un sémaphore en utilisant une
       structure struct sembuf contenant les membres suivants :
           unsigned short sem_num; /* Numéro du sémaphore       */
           short          sem_op;  /* Opération sur le sémaphore */
           short          sem_flg; /* Options pour l'opération   */

       Les  options possibles pour sem_flg sont IPC_NOWAIT et
       SEM_UNDO. Si une opération indique l'option SEM_UNDO, elle
       sera annulée lorsque le processus se terminera.

       L'ensemble  des  opérations  contenues dans sops est
       effectué atomiquement. Les opérations sont toutes réalisées
       en même temps, et  seulement si  elle  peuvent  toutes  être
       effectuées. Le comportement de l'appel système si toutes les
       opérations ne sont pas réalisables dépend  de  la présence
       de l'attribut IPC_NOWAIT dans les champs sem_flg décrits
       plus haut.

       Chaque  opération  est  effectuée  sur  le  sem_num-ième
       sémaphore  de l'ensemble.  Le  premier  sémaphore  est  le
       numéro  0.  Pour  chaque sémaphore, l'opération est l'une
       des trois décrites ci-dessous.

       Si l'argument sem_op est un entier positif, la  fonction
       ajoute  cette valeur à semval. De plus si SEM_UNDO est
       demandé, le système met à jour le compteur « undo »  du
       sémaphore  (semadj).  Cette  opération  n'est jamais
       bloquante.  Le  processus appelant doit avoir l'autorisation
       de modification sur le jeu de sémaphores.

       Si sem_op vaut zéro le processus doit avoir l'autorisation
       de  lecture sur l'ensemble de sémaphores. Le processus
       attend que semval soit nul ; si semval vaut zéro, l'appel
```

système continue immédiatement. Sinon, si l'on a réclamé **IPC_NOWAIT** dans sem_flg, l'appel système échoue (en annulant les actions précédentes) et **errno** contient le code d'erreur **EAGAIN**. Autrement semzcnt est incrémenté de 1 et le processus s'endort jusqu'à ce que l'un des événements suivants se produise :

- semval devient égal à 0, alors semzcnt est décrémenté.
- Le jeu de sémaphores est supprimé. L'appel système échoue et **errno** contient le code d'erreur **EIDRM**.
- Le processus reçoit un signal à intercepter, la valeur de semzcnt est décrémentée et l'appel système échoue avec **errno** contenant le code d'erreur **EINTR**.
- La limite temporelle indiquée par **timeout** dans un semtimedop() a expiré : l'appel système échoue avec **errno** contenant **EAGAIN**.

Si sem_op est inférieur à zéro, le processus appelant doit avoir l'autorisation de modification sur le jeu de sémaphores. Si semval est supérieur ou égal à la valeur absolue de sem_op, la valeur absolue de sem_op est soustraite de semval, et si SEM_UNDO est indiqué, le système met à jour le compteur « undo » du sémaphore (semadj). Puis l'appel système continue. Si la valeur absolue de sem_op est plus grande que semval, et si l'on a réclamé IPC_NOWAIT dans sem_flg, l'appel système échoue (annulant les actions précédentes) et **errno** contient le code d'erreur **EAGAIN**. Sinon semncnt est incrémenté de un et le processus s'endort jusqu'à ce que l'un des événements suivants se produise :

- semval devient supérieur ou égal à la valeur absolue de sem_op, alors la valeur semncnt est décrémentée, la valeur absolue de sem_op est soustraite de semval et si SEM_UNDO est demandé le système met à jour le compteur « undo » (semadj) du sémaphore.
- Le jeu de sémaphores est supprimé. L'appel système échoue et errno contient le code d'erreur EIDRM.
- Le processus reçoit un signal à intercepter, la valeur de semncnt est décrémentée et l'appel système échoue avec errno contenant le code d'erreur EINTR.
- La limite temporelle indiqué par timeout dans un semtimedop() a expiré : l'appel système échoue avec errno contenant EAGAIN.

En cas de succès, le membre sempid de chacun des sémaphores indiqués dans le tableau pointé par **sops** est rempli avec le PID du processus appelant. Enfin sem_otime est fixé à l'heure actuelle.

La fonction semtimedop() se comporte comme semop() sauf que dans le cas où le processus doit dormir, la durée maximale du sommeil est limitée par la valeur spécifiée dans la structure timespec dont l'adresse est transmise dans le paramètre timeout. Si la limite indiquée a été atteinte, l'appel système échoue avec **errno** contenant **EAGAIN** (et aucune opération de **sops** n'est réalisée). Si le paramètre **timeout** est NULL, alors semtimedop() se comporte exactement comme semop().

VALEUR RENVOYÉE
En cas de réussite, semop() et semtimedop() renvoient 0. Sinon ils renvoient -1 et errno contient le code d'erreur.

ERREURS
En cas d'erreur, **errno** prend l'une des valeurs suivantes :

E2BIG l'argument **nsops** est supérieur à **SEMOPM**, le nombre maximal d'opérations par appel système.

EACCES Le processus appelant n'a pas les permissions nécessaires pour effectuer les opérations sur les sémaphores spécifiés et n'a pas la capacité **CAP_IPC_OWNER**.

EAGAIN Une opération ne pouvait pas être effectuée immédiatement et **IPC_NOWAIT** a été indiqué dans l'argument sem_flg, ou la durée limite indiquée dans **timeout** a expiré.

EFAULT **sops** ou **timeout** pointent en dehors de l'espace d'adressage accessible.

EFBIG La valeur de sem_num est inférieure à 0 ou supérieure ou égale au nombre de sémaphores dans l'ensemble.

EIDRM Le jeu de sémaphores a été supprimé.

EINTR Un signal a été reçu pendant l'attente.

EINVAL L'ensemble de sémaphores n'existe pas ou **semid** est inférieur à zéro, ou **nsops** n'est pas strictement positive.

ENOMEM L'argument sem_flg de certaines opérations demande SEM_UNDO et le système n'a pas assez de mémoire pour allouer les structures nécessaires.

ERANGE sem_op+semval est supérieur à **SEMVMX** (la valeur maximale de semval autorisée par l'implémentation) pour l'une des opérations.

NOTES
Les structures sem_undo d'un processus ne sont pas héritées par ses enfants lors d'un fork(2). Par contre elles sont transmises lors d'un execve(2).

semop() n'est jamais relancé automatiquement après avoir été interrompu par un gestionnaire de signal quelque soit l'attribut **SA_RESTART** durant l'installation du gestionnaire.

>>

12.4 Les lecteurs-rédacteurs

12.4.1 Principe

La problématique des lecteurs-rédacteurs est relativement simple. Deux classes de processus sont en compétition pour accéder à une ressource (par exemple, un fichier, ou une région de mémoire partagée) :

- les *lecteurs*, qui ne peuvent être concurrents qu'entre eux (plusieurs processus peuvent accéder à la ressource en lecture, avec l'assurance que le résultat obtenu sera toujours le même),

- et les *rédacteurs*, qui sont exclusifs vis-à-vis de tous (un rédacteur ne peut écrire que lorsqu'aucun lecteur ou rédacteur accède à la ressource).

Une solution de ce problème peut se faire à l'aide :

- d'une variable globale (qui est une ressource critique) `nb_lect` qui indique combien de lecteurs accèdent à la ressource (la variable est initialisée à 0),
- d'un sémaphore `sem_nb_lect`, qui assure l'allocation de la variable `nb_lect`,
- et d'un sémaphore `sem_exclu`, qui assure l'exclusivité à la ressource (soit au groupe de lecteur(s), soit à un rédacteur).

Le code du rédacteur devient :

```
int  nb_lect = 0 ;
Init(sem_nb_lect, 1);
Init(sem_exclu, 1);
...
bla bla bla...
...
void Redacteur(void)
{
     P(sem_exclu);
     Section critique, je mets ici le code d'écriture...
     V(sem_exclu);
}
```

Le code du lecteur est :

```
void Lecteur(void)
{
     /* je me garantis l'exclu de la variable nb_lect : */
     P(sem_nb_lect);
     nb_lect++;
     if (nb_lect == 1) /* je suis le premier lecteur */
     {
          /* je donne l'exclusion au groupe des lecteurs */
          P(sem_exclu);
     }
     /* je n'ai plus besoin d'accéder à nb_lect : */
     V(sem_nb_lect);

     Bla bla bla (code d'accès en lecture seule)

     /* je me garantis l'exclu de la variable nb_lect : */
     P(sem_nb_lect);
     nb_lect--;
     if (nb_lect == 0) /* je suis le dernier lecteur */
     {
          /* je libère l'exclusion au groupe des lecteurs */
          V(sem_exclu);
     }
     /* je n'ai plus besoin d'accéder à nb_lect : */
     V(sem_nb_lect);
}
```

Remarque : avec cette solution, le rédacteur peut crier famine, s'il y a toujours au moins un lecteur. Exercice : chercher une solution qui assure la priorité d'utilisation aux rédacteurs (et tant pis si les lecteurs crient famine).

12.4.2 Les verrous de fichiers sous Linux

Le système d'exploitation Linux/Unix propose deux solutions simples de verrous sur les fichiers :

- les verrous partagés (qui peuvent être utilisés pour accéder à un fichier en lecture seule, évitant ainsi qu'un rédacteur vienne y écrire pendant un lecteur),
- et les verrous exclusifs (utilisés par les rédacteurs, pour éviter qu'un lecteur ou qu'un autre rédacteur y accède pendant son écriture).

Positionner un verrou sur un fichier se fait avec la fonction `flock()` :

```
<<
SYNOPSIS
       #include <sys/file.h>
       int flock(int fd, int operation);

DESCRIPTION
       Place ou enlève un verrou consultatif sur un fichier
       ouvert dont le descripteur est fd. Le paramètre operation
       est l'un des suivants :

              LOCK_SH   Verrouillage partagé. Plusieurs processus
                        peuvent disposer d'un verrouillage
                        partagé simultanément sur un même fichier.
              LOCK_EX   Verrouillage exclusif. Un seul processus
                        dispose d'un verrouillage exclusif sur
                        un fichier, à un moment donné.
              LOCK_UN   Déverrouillage d'un verrou tenu par le
                        processus.

       Un appel flock() peut bloquer si un verrou incompatible est
       tenu par un autre processus. Pour que la requête soit
       non-bloquante, il faut inclure LOCK_NB (par un OU binaire
       « | » ) avec la constante précisant l'opération.

       Un même fichier ne peut pas avoir simultanément des verrous
       partagés et exclusifs.

       Les verrous créés avec flock() sont associés à un fichier,
       ou plus précisément une entrée de la table des fichiers
       ouverts. Ainsi, les descripteurs de fichiers dupliqués (par
       exemple avec fork(2) ou dup(2)) réfèrent au même verrou,
       et celui-ci peut être relâché ou modifié à travers n'importe
       lequel des descripteurs. De plus, un verrou est
       relâché par une opération explicite LOCK_UN sur l'un
       quelconque des descripteurs, ou lorsqu'ils ont tous été
       fermés.

       Si un processus utilise open(2) (ou équivalent) plusieurs
       fois pour obtenir plusieurs descripteurs sur le même
       fichier, ces descripteurs sont traités indépendamment par
       flock(). Une tentative de verrouiller le fichier avec l'un
       de ces descripteurs peut être refusée si le processus
       appelant a déjà placé un verrou en utilisant un autre
       descripteur.

       Un processus ne peut avoir qu'un seul type de verrou
```

(partagé ou exclusif) sur un fichier. En conséquence un appel flock() sur un fichier déjà verrouillé modifiera le type de verrouillage.

Les verrous créés par flock() sont conservés lors d'un appel execve(2).

Un verrou partagé ou exclusif peut être placé sur un fichier quel que soit le mode d'ouverture du fichier.

VALEUR RENVOYÉE
L'appel renvoie 0 quand il réussit, ou -1 s'il échoue, auquel cas **errno** contient le code d'erreur.

ERREURS
EBADF **fd** n'est pas un descripteur de fichier ouvert.
EINTR Durant l'attente pour acquérir le verrou, l'appel a été interrompu par un signal capturé par un gestionnaire.
EINVAL **operation** est invalide.
ENOLCK Le noyau n'a pas assez de mémoire pour les allocations de verrou.
EWOULDBLOCK
Le fichier est verrouillé et l'attribut **LOCK_NB** a été précisé.

NOTES
flock(2) ne verrouille par les fichiers à travers NFS. Utilisez fcntl(2) à la place : il fonctionne avec NFS si la version de Linux est suffisamment récente et si le serveur accepte les verrouillages.

La conversion d'un verrou (de partagé à exclusif et vice versa) n'est pas toujours atomique : tout d'abord le verrou existant est supprimé, puis un nouveau verrou est établi. Entre ces deux étapes, un verrou demandé par un autre processus peut être accordé, ce qui peut causer soit un blocage de la conversion, soit son échec, si LOCK_NB était indiqué (C'est est le comportement BSD d'origine et de nombreuses implémentations.)

>>

12.5 Le schéma producteur-consommateur

12.5.1 Principe et résolution pour 1 producteur et 1 consommateur

Un processus, désigné comme le producteur, est chargé d'emmagasiner des données dans une file d'attente de type FIFO circulaire. Un second processus, le consommateur, est chargé de les déstocker. Evidemment, les vitesses de remplissage et de déremplissage sont aléatoires, et n'ont rien en commun. Problème : comment faire pour que chaque intervention exclue les autres ?

Rappel sur les files d'attente circulaires : la file est réalisée à l'aide d'un tableau de N cases (numérotées de 0 à N-1). Il existe deux pointeurs : le début de la file, et la fin de la file, tous deux initialisés à 0. Lorsque le producteur remplit la file, s'il reste de la place, il place la donnée dans la case pointée par la fin de file. Puis il ajoute 1 à ce pointeur. S'il vaut N, le

pointeur passe à 0. Lorsque le consommateur accède à la file pour y lire des données, s'il y a des données à lire, il lit le contenu de la case pointée par le début de file. Il ajoute 1 à ce pointeur. S'il vaut N, le pointeur passe à 0.

Pour résoudre ce problème, il suffit de le voir de la façon suivante : il existe des cases vides et des cases pleines. Le producteur produit des cases pleines et consomme des cases vides, alors que le consommateur produit des cases vides et consomme des cases pleines. Il existe deux types de ressources : les cases vides, et les cases pleines.

Il suffit alors d'utiliser deux sémaphores : un sémaphore des cases pleines, initialisé à 0, et le sémaphore des cases vides, initialisé à N :

```
Init(sem_pleines, 0);
Init(sem_vides, N);
int Ptr_debut=0;
int Ptr_fin=0;
int Cases[N];
```

Le code du producteur devient :

```
void Producteur(int message_a_enfiler)
{
     /* on attend une case libre */
     P(sem_vides);
     Cases[Ptr_fin] = message_a_enfiler;
     Ptr_fin = (Ptr_fin + 1) % N;
     /* on produit une case pleine */
     V(sem_pleines);
}
```

Le code du consommateur est :

```
int  Consommateur(void)
{
     int  message_lu;
     /* on attend une case pleine */
     P(sem_pleines);
     message_lu = Cases[Ptr_debut];
     Ptr_debut = (Ptr_debut + 1) % N;
     /* on produit une case vide */
     V(sem_vides);
     return(message_lu);
}
```

12.5.2 Extension du problème à X producteurs et Y consommateurs

Dans ce problème, plusieurs processus peuvent écrire à n'importe quel moment, et plusieurs consommateurs peuvent écrire à n'importe quel moment.

Dans ce cas, il faut s'assurer que les modifications des variables `Ptr_debut` et `Ptr_fin` ne se fassent pas en même temps par deux processus. Pour ce, il suffit de protéger chacun par son sémaphore :

```
Init(sem_Ptr_debut, 1);
Init(sem_Ptr_fin, 1);
```

Le code devient :

```
void Producteur(int message_a_enfiler)
{
     /* on attend une case libre */
     P(sem_vides);
     /* On s'assure l'exclusivité de Ptr_fin */
     P(sem_Ptr_fin);
     Cases[Ptr_fin] = message_a_enfiler;
```

```
        Ptr_fin = (Ptr_fin + 1) % N;
        /* On n'a plus besoin d'utiliser Ptr_fin */
        V(sem_Ptr_fin);
        /* on produit une case pleine */
        V(sem_pleines);
    }

    int  Consommateur(void)
    {
        int  message_lu;
        /* on attend une case pleine */
        P(sem_pleines);
        /* On s'assure l'exclusivité de Ptr_debut */
        P(sem_ Ptr_debut);
        message_lu = Cases[Ptr_debut];
        Ptr_debut = (Ptr_debut + 1) % N;
        /* On n'a plus besoin d'utiliser Ptr_debut */
        V(sem_ Ptr_debut);
        /* on produit une case vide */
        V(sem_vides);
        return(message_lu);
    }
```

A noter que les *messages queues* et les *pipes* (vus au chapitre précédent) sont aussi deux autres solutions au problème du producteur/consommateur.

12.6 L'exclusion mutuelle chez les thread

Comme nous l'avons vu, les threads partagent le même espace mémoire. Il est donc nécessaire de protéger l'accès aux variables globales.

Sans entrer dans les détails, voici les solutions mises à disposition des programmeurs pour gérer les problématiques d'exclusion mutuelle entre threads.

12.6.1 Les mutex

L'expression mutex provient de « *Mutual exclusive object* ». Le mutex est représenté dans les programmes par une variable de type `pthread_mutex_t`.

Pour créer un mutex, il suffit de créer une variable de type `pthread_mutex_t`, et de l'initialiser avec la constante `PTHREAD_MUTEX_INITIALIZER` :

```
pthread_mutex_t      mon_mutex = PTHREAD_MUTEX_INITIALIZER;
```

La destruction d'un mutex se fait à l'aide de l'appel à la function :

```
int pthread_mutex_destroy(pthread_mutex_t *pMutex);
```

Une fois créés, tous les thread aillant accès par adresse à un mutex peuvent le verrouiller lorsqu'ils ont besoin d'un accès exclusif aux données protégées. A n'importe quel instant, un unique thread est autorisé a verrouiller le même mutex. Il existe deux façons de verrouiller un mutex, via deux fonctions :

```
int pthread_mutex_lock(pthread_mutex_t *pMutex);
int pthread_mutex_trylock(pthread_mutex_t *pMutex);
```

Ces deux fonctions ont un comportement différent. Quand un thread appel la fonction `pthread_mutex_lock()`, la fonction retourne si :

- une erreur s'est produite,
- le mutex est correctement verrouillé.

Le thread en question est alors le seul qui a verrouillé le mutex. Si le mutex est déjà verrouille par un autre thread, la fonction ne retourne pas tant que le verrouillage n'est pas obtenu. En cas de succès, 0 est retourné, une valeur non nulle sinon.

La fonction `pthread_mutex_trylock()` retourne toujours même si le mutex est déjà verrouillé par un autre thread. Comme la première fonction, il retourne 0 en cas de succès. Ceci signifie que le mutex a pu être verrouille par cet appel. Si un autre thread as déjà verrouillé le mutex, EBUSY est retourné.

Dès que le thread a fini de travailler avec les données protégées, le mutex doit être déverrouillé pour laisser la place a d'autres thread. Ceci est simplement réalisé en appelant la fonction suivante :

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

12.6.2 Les variables conditions

Les ***variables conditions*** sont des éléments de type `pthread_cond_t` permettant à un processus de se mettre en attente d'un événement provenant d'un autre thread, comme, par exemple, la libération d'un mutex.

Deux opérations sont associées aux *variables conditions* :

- l'attente d'une condition,
- l'action de signaler qu'une condition est remplie.

Une variable condition est toujours associée à un mutex qui la protège des accès concurrents. Son utilisation suit le protocole suivant :

Thread en attente de condition :	**Thread signalant la condition :**
1. initialisation de la condition et du mutex associé	1. travailler jusqu'à réaliser la condition attendue
2. blocage du mutex et mise en attente sur la condition	2. blocage du mutex et signalisation que la condition est remplie
3. libération du mutex	3. libération du mutex

Il faut noter que la primitive effectuant la mise en attente libère atomiquement le mutex associé a la condition, ce qui permet au thread signalant la condition de l'acquérir à son tour (étape 2). Le verrou est de nouveau acquis par le thread en attente, une fois la condition attendue signalée. Plus précisément, l'étape 2, pour le processus attendant la condition, peut être précisée comme suit :

- blocage du mutex ;
- mise en attente sur la condition et déblocage du mutex ;
- condition signalée, réveille du thread et blocage du mutex, une fois celui-ci libéré par le thread signalant la condition (étape 3).

En pratique, l'initialisation d'une variable condition s'effectue a l'aide de la constante `PTHREAD_COND_INITIALIZER` :

```
pthread_cond_t  condition = PTHREAD_COND_INITIALIZER;
```

La mise en attente sur une condition se fait avec la primitive `pthread_cond_wait()`. Le prototype de cette primitive est :

```
pthread_cond_wait(pthread_cond_t *condition, \
                  pthread_mutex_t *mutex);
```

La primitive `pthread_cond_signal()` permet à un thread de signaler une condition remplie. Le prototype de cette primitive est :

```
pthread_cond_signal(pthread_cond_t *condition);
```

Enfin, la destruction d'une variable condition se fait à l'aide de la function :

```
int pthread_cond_destroy(pthread_cond_t *condition);
```

12.7 Autres problématiques d'interblocages

Les schémas que nous venons de voir permettent de résoudre des problèmes rencontrés classiquement lors de la programmation de systèmes d'exploitation ou de programme multi-processus. Nous allons voir quelques exemples de tels problèmes. Leurs résolutions pourront être faite dans le cadre de TD ou de TP.

12.7.1 Le dîner des philosophes

Le problème des philosophes et des spaghettis est un problème classique en informatique système. Il concerne l'ordonnancement des processus et l'allocation des ressources à ces derniers. Ce problème a été énoncé par Edsger Dijkstra.

La situation est la suivante :

- cinq philosophes (initialement mais il peut y en avoir beaucoup plus) se trouvent autour d'une table ;
- chacun des philosophes a devant lui un plat de spaghetti ;
- à gauche de chaque assiette se trouve une fourchette.

Un philosophe n'a que trois états possibles :

- penser pendant un temps indéterminé ;
- être affamé (pendant un temps déterminé et fini sinon il y a famine) ;
- manger pendant un temps déterminé et fini.

Des contraintes extérieures s'imposent à cette situation :

- quand un philosophe a faim, il va se mettre dans l'état « affamé » (*hungry*) et attendre que les fourchettes soient libres ;
- pour manger, un philosophe a besoin de deux fourchettes : celle qui se trouve à gauche de sa propre assiette, et celle qui se trouve à gauche de celle de son voisin de droite (c'est-à-dire les deux fourchettes qui entourent sa propre assiette) ;
- si un philosophe n'arrive pas à s'emparer d'une fourchette, il reste affamé pendant un temps déterminé, en attendant de renouveler sa tentative.

Le problème consiste à trouver un ordonnancement des philosophes tel qu'ils puissent tous manger, chacun à leur tour.

Remarques :

- Les philosophes, s'ils agissent tous de façons naïves et identiques, risquent fort de se retrouver en situation d'interblocage. En effet, il suffit que chacun saisisse sa fourchette de gauche et, qu'ensuite, chacun attende que sa fourchette de droite se libère pour qu'aucun d'entre eux ne puisse manger, et ce pour l'éternité.
- On considère qu'un philosophe qui meurt (crash du processus) reste dans une phase « penser » infiniment. Il en résulte donc un problème : quid d'un philosophe qui meurt avec ses fourchettes en main ?

12.7.2 L'algorithme du banquier

L'algorithme du banquier est un algorithme qui a été mis au point par Edsger Dijkstra en 1965 pour éviter les problèmes interblocages et gérer l'allocation des ressources.

Cet algorithme est nommé ainsi car il reproduit le modèle du prêt à des clients par un banquier.

Dans l'exemple qui suit, 5 processus sont actifs (A, B, C, D, E) et il existe 4 catégories de périphériques, les ressources P1 à P4. Considérons les deux tableaux suivants qui résument l'état d'un ordinateur à l'instant t :

1) liste des ressources actuellement attribuées à différents processus :

	Ressource P1	Ressource P2	Ressource P3	Ressource P4
Processus A	3	0	1	1
Processus B	0	1	0	0
Processus C	1	1	1	0
Processus D	1	1	0	1
Processus E	0	0	0	0
Total	5	3	2	2

2) liste des ressources demandées par chaque processus pour achever l'exécution :

	Ressource P1	Ressource P2	Ressource P3	Ressource P4
Processus A	1	1	0	0
Processus B	0	1	1	2
Processus C	3	1	0	0
Processus D	0	0	1	0
Processus E	2	1	1	0

3) liste des ressources disponibles au total :

	Ressource P1	Ressource P2	Ressource P3	Ressource P4
Dispo. au total	6	3	4	2

Les tableaux 1 et 3 permettent de calculer les ressources disponibles à l'instant T :

	Ressource P1	Ressource P2	Ressource P3	Ressource P4
Reste dispo.	1	0	2	0

Définition : un état est dit sûr s'il existe une suite d'états ultérieurs qui permette à tous les processus d'obtenir toutes leurs ressources et de se terminer.

Problème : trouver la suite d'états qui permettent à chaque processus de se terminer.

L'algorithme suivant détermine si un état est sûr :

1) Trouver dans le tableau 2 une ligne L dont les ressources demandées sont toutes inférieures à celles de dispo ($L_i <= dispo_i$, pour tout i). S'il n'existe pas de ligne L vérifiant cette condition, il y a interblocage ;
2) Supposer que le processus associé à L obtient les ressources et se termine. Supprimer sa ligne et actualiser le tableau 2.
3) Répéter 1 et 2 jusqu'à ce que tous les processus soient terminés (l'état initial était donc sûr) ou jusqu'à un interblocage (l'état initial n'était pas sûr)

Dans cet exemple, l'état actuel est sûr car :

- on allouera à D les ressources demandées et il s'achèvera,
- puis on allouera à A ou E les ressources demandées et A ou E s'achèvera,
- enfin, les autres.

L'inconvénient de cet algorithme est le caractère irréaliste de la connaissance préalable des ressources nécessaires à l'achèvement d'un processus. Dans bien des systèmes, ce besoin évolue dynamiquement.

13 La gestion de la mémoire

13.1 Rappels

Nous l'avons déjà vu, la mémoire accessible par le CPU est de la mémoire ***RAM*** (*Random Access Memory* - mémoire à accès arbitraire -), et plus précisément, de la ***DRAM*** (*Dynamic Random Access Memory* - mémoire à accès arbitraire dynamique -), qui stocke les bits dans des tous petits condensateurs.

Le CPU accède à cette mémoire DRAM en lisant des ***mots*** de 16, 32, 64, voire 128 bits (2, 4, 8, 16 octets).

Cette mémoire étant relativement lente (le terme « *dynamic* » qualifie le fait que ce type de mémoire nécessite un rafraîchissement régulier des cellules contenant les données, les courants de fuite des pico-condensateurs les rendant instables dans le temps), il n'est pas rare de trouver plusieurs couches de mémoire tampon entre la DRAM et le CPU, appelée ***mémoire cache***. Cette mémoire cache est de type ***SRAM*** (*Static Random Access Memory*, qui est un type de mémoire vive utilisant des bascules pour mémoriser les données, plus rapides – et plus coûteuses – que les condensateurs des mémoires DRAM).

Chaque octet de 8 bits est rangé dans une « case » de la mémoire. Chaque case d'un octet est numérotée (de zéro à N-1, N étant la taille en octet de la mémoire physique disponible) ; ce numéro est appelé « l'***adresse*** », ou plus précisément « ***adresse physique*** » de la case mémoire qui contient la donnée.

Le CPU accède aux données de la RAM en positionnant l'adresse du mot à lire (ou à écrire) dans un ***registre d'adresse*** (souvent symbolisé dans la littérature française par le sigle ***RAD***). Les données elles-mêmes sont stockées au sein du CPU dans des registres de données (appelés souvent ***RDO***).

13.2 Espace d'adressage

Nous l'avons déjà vu, chaque processus possède un certain nombre d'endroits où il a le droit de lire (ou d'écrire) des données, et où du code machine (à exécuter par le CPU) peut être placé. Cet espace de RAM où un processus a le droit d'accéder s'appelle l'***espace d'adressage***. Un processus n'a pas le droit d'accéder à l'espace d'adressage d'un autre processus (sous peine de déclencher une exception au niveau du CPU).

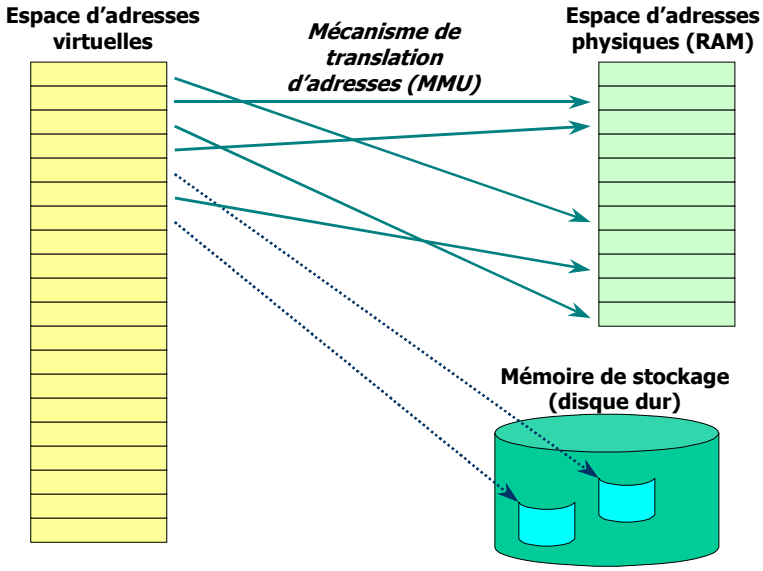
Vous l'aurez compris (et nous reviendrons sur ce point), c'est un mécanisme au sein **du CPU** (mécanisme électronique, câblé) qui vérifie si un processus accède bien aux zones qui lui sont réservées.

L'espace d'adressage d'un processus contient plusieurs zones, qui sont au moins :

- le code machine à exécuter par le CPU,
- la pile à l'exécution du CPU,
- l'espace de données.

En pratique, un processus peut accéder virtuellement à plus de mémoire qu'il en existe dans l'ordinateur. Ceci est possible parce qu'en pratique, chaque processus n'accède pas directement à la mémoire RAM en utilisant des adresses physiques, mais il utilise des ***adresses logiques***. L'ensemble des adresses logiques accessibles par un processus s'appelle l'***espace d'adressage logique*** (ou ***espace d'adressage virtuel***), par opposition à l'ensemble des adresses physiques accessibles, appelé ***espace d'adressage physique***. Il est ainsi possible de faire pointer des zones de l'espace d'adressage logique vers des zones de l'espace d'adressage physique, alors que d'autres pointent sur un espace situé sur le disque dur. Cette zone sur le disque dur s'appelle le ***fichier d'échange de mémoire virtuelle***, ou ***fichier de swap***.

Pour que le processeur accède à la mémoire RAM, le processeur doit convertir chaque adresse logique manipulée par le processus en adresse physique. Ce travail de traduction d'adresses est parfois appelé par le mot d'origine anglaise *mapping*. Cette conversion est faite au sein du processeur par un élément matériel (câblage électronique) appelé ***MMU*** (*Memory Management Unit*).



Principe de traduction d'adresses (MMU)

Pour faciliter l'assignation de la mémoire physique aux différents processus, la mémoire est virtuellement découpée en zones. Il existe deux philosophies différentes (qui peuvent être utilisées simultanément, comme c'est fréquemment le cas) pour « découper » la mémoire :

- la mémoire peut être découpée en morceaux de tailles variables appelés ***segments***, caractérisés par leur adresse de début (décalage) et leur taille. On parle alors de ***segmentation*** ;
- et il est possible de la découper en « blocs » de taille fixe appelés ***pages***. On parle de ***pagination***.

Au sein du CPU, la MMU a donc pour rôle :

- la translation d'adresses logiques en ***adresses linéaires*** par l'unité de segmentation ;
- la translation d'adresses linéaires en adresses physiques par l'unité de pagination ;
- la protection mémoire (éviter qu'un processus n'accède à un espace qui ne lui est pas réservé) ;
- le contrôle de tampon (gestion de la mémoire cache) ;
- l'arbitrage du bus (qui a le droit d'accéder au bus en cas de besoin d'utilisation simultané).

13.3 La segmentation

La ***segmentation*** est une approche de virtualisation de la mémoire physique. Celle-ci est divisée en zones pouvant être de différentes tailles. Un segment est caractérisé par :

- l'adresse physique de sa base (adresse physique du début du segment),
- sa taille.

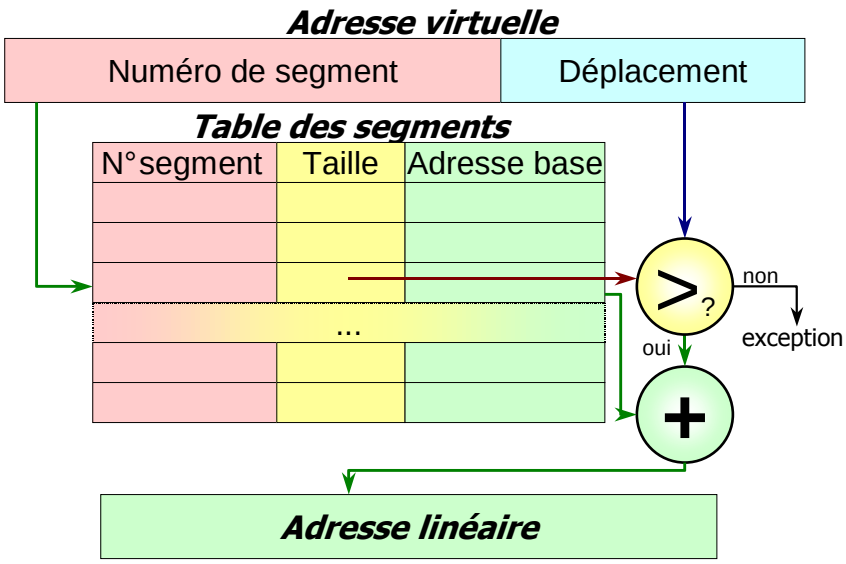
Une adresse virtuelle possède alors deux composantes :

- un numéro de segment,
- un déplacement dans ce segment.

Le MMU contient une table des segments, qui est indexée par le numéro de segment, et qui, pour chaque entrée, contient :

- la taille du segment,
- et son adresse de base.

Pour traduire une adresse virtuelle, le MMU cherche l'entrée correspondant au numéro de segment dans la table des segments. Il vérifie que le déplacement contenu dans l'adresse virtuelle est inférieur à la taille du segment. Si c'est le cas, il additionne le déplacement à l'adresse de base du segment pour obtenir l'adresse physique :



Traduction d'adresse (segmentation)

Remarques :

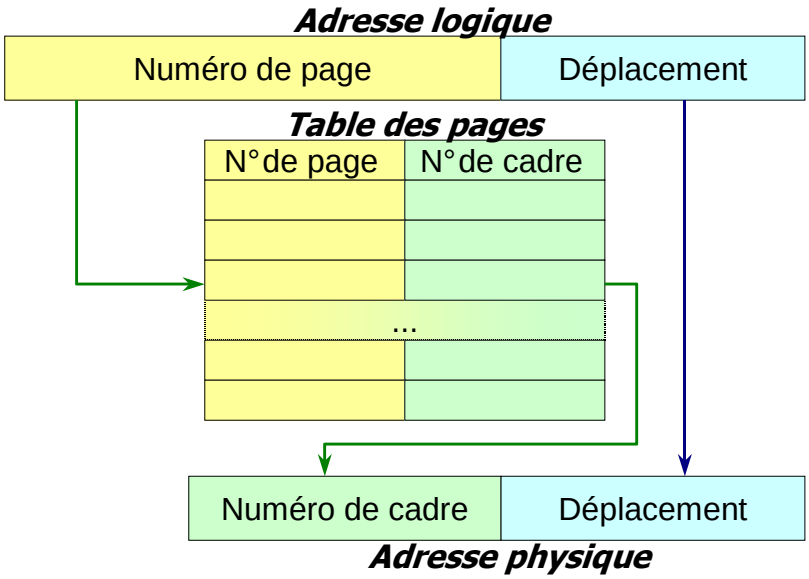
- une même portion de mémoire linéaire peut faire partie de plusieurs segments ;
- la table des segments de chaque processus est chargée dans la MMU par l'ordonnanceur à chaque commutation de contexte ;
- les mécanismes d'allocation de la mémoire s'arrangent allouer des segments les uns à la suite des autres. Lorsqu'un processus rend de la mémoire, cet espace libre peut être alloué lors d'une nouvelle demande. Mais si la quantité demandée est plus petite que la première fois, il peut apparaître des « trous » d'espace mémoire libre, trop petits pour loger de nouveaux segments. La pagination engendre un phénomène de ***fragmentation externe*** de la mémoire. Il est possible de « défragmenter » la mémoire (en déplaçant les segments, tout en reflétant ces modifications dans les tables des segments) de sorte que les segments soient contigus. Néanmoins cette opération est coûteuse.

13.4 La pagination

Le principe de la pagination est le suivant :

- les adresses mémoires émises par le processeur sont des adresses virtuelles linéaires, indiquant la position d'un mot dans la mémoire virtuelle ;

- cette mémoire virtuelle est formée de zones de même taille, appelées pages. Une adresse virtuelle est donc un couple (***numéro de page***, ***déplacement dans la page***). La taille des pages est une puissance de deux (en général, entre 512 et 8'192 octets), de façon à déterminer sans calcul le déplacement dans la page (exemple : 10 bits de poids faible de l'adresse virtuelle pour des pages de 1024 mots), et le numéro de page (les autres bits) ;
- la mémoire physique est également composée de zones de même taille, appelées « ***cadres*** » (***frames*** en anglais), dans lesquelles prennent place les pages. A noter que la taille d'un cadre est égale à la taille d'une page ;
- le MMU assure la conversion des adresses virtuelles en adresses physiques, en consultant une « ***table des pages*** » (***page table*** en anglais) pour connaître le numéro du cadre qui contient la page recherchée. L'adresse physique obtenue est le couple (***numéro de cadre***, ***déplacement***) ;
- il peut y avoir plus de pages que de cadres : les pages qui ne sont pas en mémoire sont stockées sur un autre support (disque dur), elles seront ramenées dans un cadre si besoin.



Traduction d'adresse (pagination)

Chaque processus possède sa propre *table des pages*, qui est indexée par le numéro de page. Chaque ligne est appelée une « ***entrée dans la table des pages*** » (*pages table entry*, abrégé *PTE*), et contient le numéro de cadre. La table des pages pouvant être située n'importe où en mémoire, un registre spécial (le ***PTBR*** pour ***Page Table Base Register***) conserve son adresse.

En pratique, le MMU contient également une partie de la table des pages, stockée dans une mémoire associative (mémoire cache appelée ***TLB*** : *look-aside buffers*) formée de registres associatifs rapides. Ceci évite d'avoir à consulter la table des pages (en mémoire) pour chaque accès mémoire (y compris la consultation de la table des pages).

De plus, de son coté, le système maintient une ***table des cadres de pages*** ou ***table des cases*** pour savoir à chaque instant, pour chaque cadre de la mémoire physique, si le cadre est libre ou occupé, et s'il est occupé, quel processus la possède. Exemple de table des cadres de pages :

N°de cadre/frame number	N°de page	Processus propri étaire
Cadre 1	-1	/
Cadre 2	1	Processus 1
Cadre 3	-1	/
Cadre 4	3	Processus 1
Cadre 5	6	Processus 2
Cadre 6	4	Processus 1
Cadre 7	5	Processus 2

Petit exercice : à partir de la table des cases (ou table des cadres de pages) ci-dessus, pour chaque processus, construite sa table des pages.

Remarques :

- la table des pages de chaque processus est chargée dans la MMU par l'ordonnanceur à chaque commutation de contexte (en pratique, on fait pointer le PTBR vers la table des pages du processus élu, et la mémoire tampon de la MMU est vidée) ;
- lors d'une demande d'allocation de mémoire par un processus, il est rare que ce dernier demande exactement un nombre entier de fois la taille d'une page. Le système alloue tout de même des pages complètes, ce qui explique qu'il existe une ***fragmentation interne*** de la mémoire due à la pagination.

13.5 Protection des accès mémoire entre processus

Il existe trois bits de protection pour chaque partie de l'espace d'adressage d'un processus, permettant de gérer les accès. Les 3 bits signifient : lecture, écriture, et exécution de code. Ces bits de protection existent aussi bien dans les mécanismes de segmentation que dans les mécanismes de pagination. Pour des raisons d'efficacité, c'est souvent dans ce dernier mécanisme qu'ils sont mis en œuvre.

Chaque processus ne peut accéder qu'à sa propre table des pages. Lors de chaque accès à la mémoire, les droits sont vérifiés, et une exception (trappe) est levée si les droits ne sont pas respectés.

Il peut arriver que des processus aient besoin de partager des zones de mémoire (exemple : *shared memory*, ou bibliothèque de fonction, ou père et fils qui exécutent le même code, etc.). Dans ce cas, chacun des processus référence les pages communes dans sa propre table des pages. Et les deux processus doivent être indiqués comme propriétaires dans la table des cases pour chacune de ces cases.

Exemple :

- Table des cases ayant des cases partagées en lecture/écriture :

N°de cadre/frame number	N°de page	Processus propri étaire
Cadre 1	1	Processus 1
Cadre 2	3	Processus 1
Cadre 3	2	Processus 1, Processus 2
Cadre 4	5	Processus 2

- Table des pages du processus 1 :

N°de page	Droit	N°de cadre
1	`--x`	1
2	`rw-`	3
3	`rw-`	2

- Table des pages du processus 2 :

N°de page	Droit	N°de cadre
2	`rw-`	3
5	`--x`	4

13.6 La pagination multiniveaux

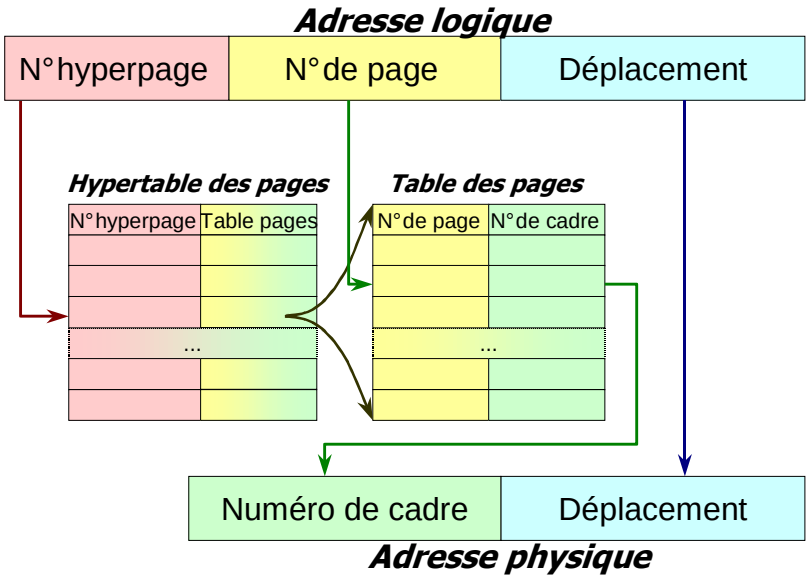
Les architectures 32 bits permettent d'adresser 2^{32} octets (4'294'967'296). Les architectures 64 bits permettent théoriquement l'adressage de 2^{64} octets (18'446'744'073'709'551'616). La baisse des coûts de production de la mémoire RAM fait que les machines en ont de plus en plus, et il devient inconcevable de charger de manière continue la table des pages d'un processus.

Ainsi est née la notion d'***hypertable*** : la table des tables des pages d'un processus. L'adresse virtuelle ne contient plus deux sous ensembles (la page et le décalage), mais 3 : l'***hyperpage***, la page, et le décalage.

L'hypertable est aussi située en mémoire centrale, et se trouve pointée elle aussi par un registre matériel particulier du MMU.

L'accès à la mémoire physique se fait alors avec un accès supplémentaire :

- la lecture de l'hyperpage est utilisée comme entrée dans l'hypertable, et nous permet d'accéder à une table des pages ;
- la page permet de trouver le cadre dans la table des pages ainsi obtenue ;
- la lecture de la donnée dans la mémoire physique se fait à l'aide de ce cadre et du décalage.



Traduction d'adresse (hyperpagination)

13.7 Mémoire virtuelle – swap

13.7.1 Principe

Le mécanisme d'allocation mémoire commence par allouer en premier tous les cadres disponibles dans la mémoire physique. Lorsqu'il n'y a plus de cadre disponible, il libère certains cadres en plaçant leur contenu sur disque.

Lorsqu'il est élu, le processus ayant des pages *swapées* sur le disque devra être capable de connaître cette information. Aussi, chaque entrée de la table des pages possède un ***bit de validation*** (Appelé généralement « ***bit V*** »). Ce bit est à `1` si la page fait référence à un cadre en mémoire centrale, et vaut `0` si le contenu de la page a été stocké sur disque.

Les pages contenant du code (programme) ne sont jamais *swapées*, mais tout simplement réassignées. En effet, il est toujours possible d'en retrouver le contenu dans le fichier correspondant au programme (le fichier de l'exécutable).

Voici ce qu'il se passe lorsque le MMU lit une entrée dans la table des pages, et que le *bit V* de validation est à zéro :

- une trappe appelée « *défaut de page* » est levée, qui suspend l'exécution du processus ;
- le handler de cette exception vérifie s'il reste des cases de libres. S'il n'y a plus de place dans la mémoire vive, le système choisit des cases à libérer. Il stocke le contenu des cases choisies dans la zone de swap sur le disque, et dans les entrées des tables de pages concernées, il positionne les bits V à 1, et change le pointeur de case pour y placer la référence sur le disque dur où la case a été swapée ;
- puis, il initialise une opération d'entrées-sorties afin de charger la page manquante en mémoire centrale (dans une case libre) depuis la zone de swap du disque ;
- l'entée dans la table des pages du processus (et dans la table des cases) qui vient d'être chargée est mise à jour (le bit V est positionné à 1, et le pointeur est changé pour pointer vers la case qui vient d'être chargée, au lieu de pointer vers le disque ;
- le processus qui a levé l'exception reprend son cours.

Comme nous venons de le voir, lorsqu'il n'y a plus de place en mémoire centrale, une case est choisie pour être swapée. Or, cette case a déjà pu être swapée par le passé. Dans ce cas, il est tout à fait intéressant de savoir si le contenu de la case a été modifié en mémoire centrale depuis son dernier chargement. Ainsi, si le contenu n'a pas été modifié (i.e. que les accès par le processus ont été réalisés en lecture seule), et si la zone de swap qui contenait cette case n'a pas été ré-attribuée, il est inutile de stocker son contenu sur disque. C'est pour cette raison qu'en plus du bit V de validation, la table des pages contient aussi un « ***bit M*** », dit ***bit de modification***. Il est appelé le « ***bit D*** » (ou ***bit Dirty***) dans la littérature anglaise. Ce bit vaut 1 si la page est nouvellement allouée (elle n'a ainsi jamais été swapée par le passé), ou lorsque le contenu de la case a été modifié depuis son dernier chargement en provenance de la zone de swap.

Il existe plusieurs philosophies pour choisir quelles sont les pages qui doivent être « *swapées* » quand il n'y a plus de place en mémoire centrale. L'objectif de ces algorithmes est de choisir des pages qui ont la probabilité la plus faible d'être utilisée à court terme, réduisant ainsi la probabilité d'avoir rapidement de nouveaux défauts de page.

Tout d'abord, il est assez rare que les développeurs de système d'exploitation fassent le choix de développer un algorithme recherchant une unique page à *swaper*. En général, les systèmes d'exploitation font le choix d'un processus, et il *swape* toutes les pages du processus. En agissant ainsi, le système libère d'un coup plusieurs pages (toutes celles d'un processus en réalité), réduisant ainsi la probabilité d'avoir à nouveau à faire tourner l'algorithme « élection d'une page à *swaper* » au défaut de page suivant.

Afin d'évaluer ces différentes philosophies, les ingénieurs simulent une suite d'accès mémoire, et comptent le nombre de défaut de page engendré. Parmi les algorithmes connus pour choisir quelles sont les pages qui doivent être « *swapées* », citons :

- l'algoritme ***optimal***,
- l'algorithme du ***remplacement aléatoire*** (où le *cadre victime* est choisi au hasard),
- l'algorithme ***FIFO*** (*First In, First Out*/premier entré, premier sorti),
- l'algorithme ***LRU*** (*Least Recently Used*/la moins récemment utilisée),
- l'algorithme ***de la seconde chance***,
- l'algorithme ***LFU*** (*Least Frequently Used*, soit « la moins souvent utilisée »).

13.7.2 L'algorithme optimal

Quelle que soit la suite d'accès aux pages, il existe toujours une solution optimale engendrant le moins de défaut de page. Il suffit de choisir de swaper la page qui ne sera pas utilisée durant la période la plus longue.

Exemple (source : Wikipedia) : supposons qu'une machine ayant 3 cases de mémoire vive ait besoin d'accéder à 8 pages (numérotées de 0 à 7) dans l'ordre suivant : « 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 » :

Page demandée	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Case 1	7	7	**7**	2	2	2	2	2	2	2	2	2	2	2	2	2	**2**	7	7	7
Case 2		0	0	0	0	0	**0**	4	4	**4**	0	0	0	0	0	0	0	0	0	0
Case 3			1	1	**1**	3	3	3	3	3	3	3	**3**	1	1	1	1	1	1	1
Défaut de page	O	O	O	O	N	O	N	O	N	N	O	N	N	O	N	N	N	O	N	N

Cet algorithme n'aura engendré que 9 défauts de page. Toute autre solution aurait engendré plus de défaut de page. Malheureusement, il est impossible de mettre en œuvre cet algorithme. Il faudrait connaître à l'avance la chaîne d'accès aux pages, ce qui reviendrait à connaître l'avenir... Les autres algorithmes cherchent à s'approcher de cette solution optimale.

13.7.3 L'algorithme FIFO

Pour cet algorithme, nous faisons l'hypothèse que c'est la page la plus anciennement chargée en mémoire qui doit être swapée. Pour implémenter cet algorithme, il convient d'ajouter dans la table des pages la date du dernier chargement de chaque case (la case choisie se faisant après la recherche de celle dont cette date de dernier chargement est la plus ancienne), ou alors, de gérer une structure de liste où l'on amènera en première position le cadre chargé depuis le plus longtemps (liste FIFO).

Si la programmation de cet algorithme est assez simple, son efficacité n'est pas optimale. Dans notre exemple, il engendrerait 15 défauts de page :

Page demandée	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Case 1	7	7	**7**	2	2	2	**2**	4	4	**4**	0	0	0	0	0	0	**0**	7	7	7
Case 2		0	0	0	**0**	3	3	**3**	2	2	2	2	**2**	1	1	1	1	**1**	0	0
Case 3			1	1	1	**1**	0	0	**0**	3	3	3	3	**3**	2	2	2	2	**2**	1
Défaut de page	O	O	O	O	N	O	O	O	O	O	O	N	N	O	O	N	N	O	O	O

13.7.4 L'algorithme LRU

Cet algorithme consiste à choisir comme victime le cadre qui n'a pas été référencé depuis le plus longtemps. On peut l'implémenter soit en rajoutant chaque entrée de la table des

pages une date qui indique quand a eu lieu la dernière référence à cette entrée (on choisira alors, la page dont la date est la plus ancienne), soit via une structure de liste où l'on amènera en première position le cadre le plus anciennement référencé. Dans notre exemple, cet algorithme donne lieu à 12 remplacements :

Page demandée	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Case 1	7	7	**7**	2	2	2	**2**	4	4	**4**	0	0	**0**	1	1	1	1	1	1	1
Case 2		0	0	0	0	0	0	0	**0**	3	3	3	3	3	**3**	0	0	0	0	0
Case 3			1	1	**1**	3	3	**3**	2	2	2	2	2	2	2	2	**2**	7	7	7
Défaut de page	O	O	O	O	N	O	N	O	O	O	O	N	N	O	N	O	N	O	N	N

13.7.5 L'algorithme de la seconde chance

Un bit de référence est associé à chaque page dans la table des pages des processus. Ce bit de référence est mis à 1 a chaque référence à la page. Le choix de la page victime (qui va être swapée) se fait suivant ce mécanisme :

- la page la plus anciennement chargée est sélectionnée comme étant à priori la page victime (même début que pour l'algorithme LRU) ;
- si la valeur du bit de référence est à 0, alors la page est effectivement remplacée ;
- au contraire, si la valeur du bit de référence est à 1, alors une seconde chance est donnée à la page. Son bit de référence est remis à 0, et la page la plus anciennement chargée suivante (dont le bit de référence est à 0) est sélectionnée.

Cet algorithme constitue une bonne optimisation de l'algorithme LRU. Il n'est pas beaucoup plus coûteux à implémenter car de nombreux matériels offrent directement un bit de référence associé à chaque case de la mémoire centrale (la gestion de ce bit est faite par le MMU, et non par du logiciel).

13.7.6 L'algorithme LFU

Dans cet algorithme, le système garde un compteur qui est incrémenté à chaque fois que le cadre est référencé. La victime sera le cadre dont le compteur est le plus bas.

Cet algorithme, bien qu'intuitivement correct, souffre de deux inconvénients, qui le rendent peu utilisé :

- au démarrage des programmes, quelques pages peuvent être intensément utilisées, puis plus jamais par la suite. La valeur du compteur sera si élevée qu'ils ne seront remplacés que trop tardivement ;
- il faut aussi gérer le cas de dépassement de capacité du compteur, ce qui est coûteux en CPU.

13.7.7 Anomalie de Belady

Intuitivement, nous serions en mesure de penser qu'augmenter le nombre de cadres de pages (c'est-à-dire agrandir la taille de la mémoire vive), doit réduire le nombre de défauts de page.

L'*anomalie de Belady* (du nom de la personne ayant découvert ce phénomène en 1970) est un contre-exemple, qui montre que ce n'est pas toujours vrai avec l'algorithme FIFO. Il existe des contres-exemples où le nombre de défaut de page est plus important avec plus de cadres. Néanmoins, il ne faut pas exagérer la portée de cette curiosité. Si des contres exemples à l'efficacité de l'algorithme FIFO existent, ils restent marginaux au regard de son efficacité « en moyenne ».

Par ailleurs, d'autres algorithmes de remplacement de pages (LRU par exemple) ne sont pas sujets à ce type d'anomalie.

Exercice : simuler l'algorithme LRU avec une mémoire de 3 cadres avec séquence de références aux pages suivantes : 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4. Refaire le même exercice avec la même séquence, mais avec une mémoire vive de 4 cadres.

13.8 Application : la gestion de mémoire sous Linux

Le système d'exploitation Linux étant destiné à s'exécuter sur de nombreuses machines d'architectures différentes, le modèle de gestion mémoire mis en œuvre a été défini pour dépendre le moins possible d 'un équipement électronique particulier (il utilise une sorte de plus petit dénominateur commun). La programmation de ce modèle s'est faite au cas par cas, en fonction du support matériel offert par la machine (par curiosité, dans les sources du noyau Linux, il existe un répertoire « `arch` » qui contient les sous-répertoires suivants : « `alpha` », « `cris` », « `i386` », « `m68k` », « `parisc` », « `s390` », « `sparc` », « `v850` », « `arm` », « `frv` », « `ia64` », « `m68knommu` », « `ppc` », « `sh` », « `sparc64` », « `x86_64` », « `arm26` », « `h8300` », « `m32r` », « `mips` », « `ppc64` », « `sh64` », « `um` », « `xtensa` ». Chacun de ces sous-répertoire contient du code spécifique à une architecture matérielle particulière (et à un CPU particulier). Dans ce sous-répertoire ce trouve un dossier « `mm` » qui contient les fonctions de gestion de mémoire (souvent programmées en assembleur) spécifique à l'architecture.

13.8.1 Les régions

L'espace d'adressage d'un processus Linux est compose de cinq parties principales (on parle de cinq régions) qui ont des adresses continues de l'espace d'adressage :

- le code,
- les données initialisées (exemple, une variable globale « `char *s="123";` »),
- les données non initialisées (exemple, une variable globale « `int compteur;` »),
- le tas (qui est alloué par exemple avec la fonction `malloc()`),
- et la pile à l'exécution (utilisées par les opérations `push` et `pop` du CPU).

Il comporte en plus :

- une zone contenant les arguments du programme exécutable et les variables d'environnements de celui-ci,
- des régions allouées pour contenir les données des bibliothèques partagées utilisées par le processus,
- et enfin, tout processus peut ajouter de nouvelles régions à son espace d'adressage, dans lesquelles il peut projeter le contenu d'un fichier. Cette opération de projection permet alors au processus d'accéder aux données du fichier par le biais d'opération de lecture et d'écriture en mémoire centrale plutôt que par le biais des opérations d'entrées-sorties (qui sont alors effectuées par le système).

Avec un CPU de type x86, 3 Go sont affectés à l'espace d'adressage d'un processus. Les régions sont paginées avec des pages dont la taille est de 4 Ko. Pour éviter la fragmentation, les régions ont une taille de taille multiple de 4'096.

13.8.2 La gestion de la pagination

Pour chaque entrée (chaque ligne) de la table des pages, le noyau linux indique :

- Présent (bit V de validation),
- Accessed (bit de référence dans l'algorithme de la seconde chance),
- Dirty (bit M de modification),
- Read/write (âge du dernier accès),

- Le numéro de case dans la mémoire vive.

De plus, une structure système permet de stocker les propriétés de chaque page :

- adresses des cases libres précédentes et suivantes,
- le nombre de processus partageant la case,
- état de la page (verrouillée, accédée, *dirty*, etc.),
- un champ âge.

Le code du noyau qui gère quelles sont les pages à swaper s'appelle le thread « *kswapd* ». Il utilise l'algorithme de la deuxième chance :

- le thread noyau « *kswapd* » dit le « *dérobeur* de page » est réveillé toutes les 10 secondes,
- il libère des pages si le nombre de pages vides descend en dessous d'un certain seuil (il n'attend pas les défauts de page),
- une page est swapée lorsqu'elle a atteint un certain temps sans être référencée (cet âge est un paramètre système).

A chaque référence à une page, l'âge de la page est mis à 0, et le bit *accessed* est mis à 1. A chacun de ses passages, le thread kswapd :

- met à 0 le bit accessed s'il est à 1,
- incrémente l'âge de la page.

Une page est élue pour être swapée :

- si le bit *accessed* est à 0,
- l'âge limite (paramètre système) est atteint.

13.8.3 La gestion de l'espace d'adressage

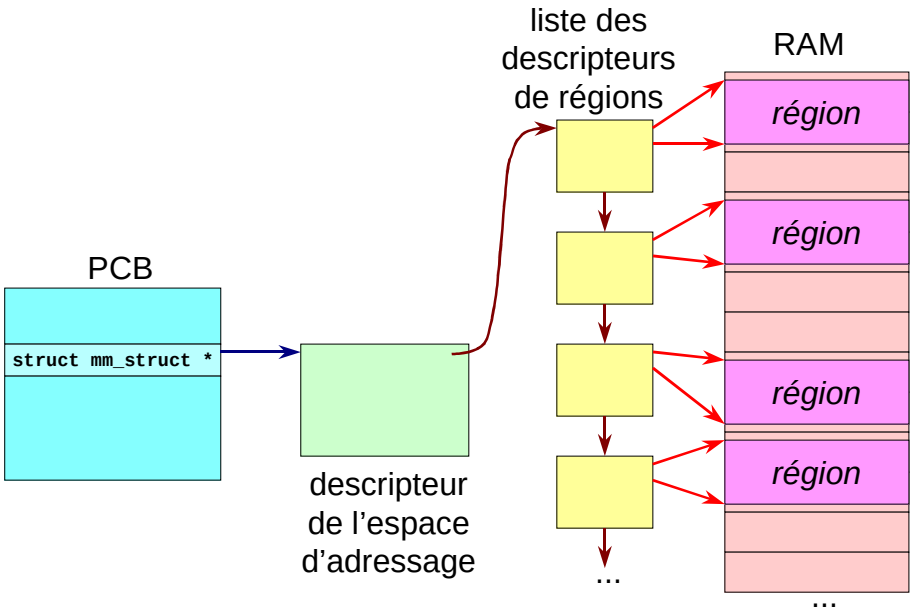
Chaque processus possède, dans son du ***PCB*** (révision : ***Process Control Block***/Bloc de Contrôle de Processus) un pointeur de type « `struct mm_struct *` ». Une telle structure contient :

- la liste des régions composant l'espace d'adressage ;
- le nombre de processus partageant ce descripteur (ce nombre pouvant être supérieur à 1 dans le cas de processus légers évoluant au sein du même espace d'adressage) ;
- les adresses de début et de fin des sections de code, données non initialisées, données initialisées, tas et pile ;
- les adresses de début et de fin des sections contenant les arguments du programme exécutable ;
- les adresses de début et de fin des sections contenant les variables d'environnements du programme exécutable;
- l'adresse de la table globale des pages du processus ;
- la taille en octets de l'espace d'adressage du processus.

De plus, le système gère une liste des régions, ordonnée suivant un ordre croissant des adresses mémoire. Cette liste chaîne des éléments de type « `struct vm_area_struct` » définie dans les sources du noyaux dans le fichier « `linux/mm.h` ». Cette structure contient :

- les adresses de début et de fin de la région ;
- les propriétés associées à la région, notamment :
    - `VM_READ` : la région est accessible en lecture,
    - `VM_WRITE` : la région est accessible en écriture,

- VM_EXEC : la région est accessible en exécution,
- VM_GROWSDOWN, VM_GROWSUP : la région peut être étendue soit vers le bas, soit vers le haut,
- VM_LOCKED : la région est verrouillée et ne peut pas être retirée de mémoire centrale (swap interdit),
- VM_SHARED : la région est partagée par plusieurs processus (thread par exemple),
- VM_SHM, la région est une région de mémoire partagée (Cf. les IPC *shared memory*).



Descripteurs de régions d'un processus

Un accès non conforme à une région provoque La levée d'une trappe et l'envoi au processus fautif du signal SIGSEGV qui entraîne la terminaison du processus fautif.

Les appels systèmes qui invitent à créer/détruire des espaces d'adressage sont :

- les fonctions fork() et exec() :
  - la création d'un nouveau processus via l'appel système fork() engendre une duplication de l'espace d'adressage du processus père pour créer l'espace d'adressage du processus fils par le mécanisme de copie sur écriture (*copy on write*) déjà abordé :
    - les pages de l'espace d'adressage du fils pointent en lecture seule vers les mêmes cases que le père,
    - en cas d'accès en lecture vers ces cases, rien ne change. En cas d'accès en écriture, une exception est levée. Le système alloue alors une nouvelle région, en y allouant de nouvelles cases, copie dedans les données des cases du père, et place les droits d'accès cette fois-ci en lecture/écriture ;
  - lors d'un appel à une fonction de la famille execXXX(), il y a recouvrement du code et des données d'un processus par un nouveau code et de nouvelles données. Les anciennes régions héritées au moment du fork() sont libérées et de nouvelles régions sont allouées pour contenir le nouveau code et les nouvelles données du processus ;
- la projection d'un fichier en mémoire centrale. Le système alloue alors une nouvelle région pour contenir le fichier par les fonctions mmap() et munmap() :

```
#include   <unistd.h>
#include   <sys/mman.h>
void *mmap(void *debut, size_t longueur, int prot, \
           int flags, int desc, off_t offset);
int munmap(void *debut, size_t longueur);
```

- la création d'une nouvelle région de mémoire partagée (Cf. IPC *shared memory*).

La modification de la taille d'une région peut se faire pour la région du « tas », lors des appels aux fonctions `malloc()`, `calloc()`, `realloc()`, et `free()`.

```
<<
SYNOPSIS
       #include <stdlib.h>
       void *calloc(size_t nmemb, size_t size);
       void *malloc(size_t size);
       void free(void *ptr);
       void *realloc(void *ptr, size_t size);

DESCRIPTION
       calloc() alloue la mémoire nécessaire pour un tableau de
       nmemb éléments, chacun d'eux représentant size octets,
       et renvoie un pointeur vers la mémoire allouée. Cette
       zone est remplie avec des zéros.

       malloc() alloue size octets, et renvoie  un  pointeur  sur
       la  mémoire allouée. Le contenu de la zone de mémoire n'est
       pas initialisé.

       free() libère l'espace mémoire pointé par ptr, qui a été
       obtenu lors d'un appel antérieur à malloc(), calloc() ou
       realloc(). Si le  pointeur ptr  n'a  pas  été  obtenu  par
       l'un de ces appels, ou s'il a déjà été libéré avec free(),
       le comportement est indéterminé. Si ptr est égal à NULL,
       aucune tentative de libération n'a lieu.

       realloc()  modifie  la  taille  du  bloc de mémoire pointé
       par ptr pour l'amener à une taille de size octets. realloc()
       conserve le contenu  de la zone mémoire minimum entre la
       nouvelle et l'ancienne taille. Le contenu de la zone de
       mémoire nouvellement allouée n'est  pas  initialisé. Si ptr
       est NULL, l'appel de realloc() est équivalent à
       malloc(size). Si size vaut zéro, l'appel est équivalent à
       free(ptr). Si  ptr  n'est  pas NULL,  il doit avoir été
       obtenu par un appel antérieur à malloc(), calloc() ou
       realloc(). Si la zone pointée était déplacée, un free(ptr)
       est effectué.

VALEUR RENVOYÉE
       Pour  calloc()  et  malloc(), la valeur renvoyée est un
       pointeur sur la mémoire allouée, qui est correctement
       alignée pour n'importe quel type de variable, ou NULL si la
       demande échoue.

       free() ne renvoie pas de valeur.

       realloc()  renvoie un pointeur sur la mémoire nouvellement
       allouée, qui est correctement alignée pour n'importe quel
       type de variable,  et  qui peut  être différent de ptr, ou
       NULL si la demande échoue. Si size vaut zéro, realloc
```

```
       renvoie NULL ou un pointeur acceptable  pour  free().  Si
       realloc()  échoue,  le  bloc mémoire original reste intact,
       il n'est ni libéré ni déplacé.

>>
```

13.9 Les algorithmes d'allocation mémoire

13.9.1 Principes généraux

Avant tout, il faut distinguer deux types d'allocation :

- l'allocation statique (celle qui est faite au lancement d'un processus, pour réserver de la place pour le code, pour la pile, les variables de taille fixe, etc). L'avantage de l'allocation statique se situe essentiellement au niveau des performances, puisqu'on évite les coûts de l'allocation dynamique à l'exécution : la mémoire statique est immédiatement utilisable ;
- l'allocation dynamique (qui est faite pour gérer des objets de tailles variables). Beaucoup plus souple, cette méthode oblige à allouer avant d'utiliser (puis libérer une fois devenu inutile) l'espace nécessaire pour stocker chaque objet.

Un algorithme simple de gestion de la mémoire consiste à regarder la mémoire libre dans son ensemble, et de l'allouer au fur et à mesure. Ainsi, lorsqu'un processus demande N octets, le noyau cherche la première zone d'espace continu ayant N octets de mémoire contiguë, et l'alloue. Cet algorithme, nommé « ***allocation first fit*** ».

Le problème avec cet algorithme simpliste, est qu'à force d'allocation/désallocation, la mémoire se fragmente. Une optimisation de cet algorithme consiste à chercher le plus petit espace contiguë de mémoire libre contenant N octets (algorithme « ***allocation best fit*** »). Le système recherche le plus petit espace libre de M octets, avec $M \geq N$. L'inconvénient de cet algorithme est qu'il génère beaucoup de petits espaces libres de « M-N » octets, qui sont inutilisés car trop petits.

Dans tous les cas, pour parcourir la mémoire, le noyau construit des listes chaînées (voire doublement chaînées) entre elles.

13.9.2 Implémentation sous Linux

Le noyau gère l'ensemble des cases libres de la mémoire centrale par l'intermédiaire d'une table nommée « `free_area` » comportant 10 entrées. Chacune des entrées comprend deux éléments :

- une liste doublement chaînée de blocs de pages libres, l'entrée n¶ gérant des blocs de 2^i pages ;
- un tableau binaire nommé « `bitmap` », qui reflète l'allocation des pages allouées. Pour une entrée i du tableau `free_area` ($0 \leq i \leq 9$), chaque $n^{ème}$ bit du tableau reflète l'état d'un ensemble de deux groupes de 2^i blocs de pages. Si le bit est à 0, alors les deux groupes de blocs de pages correspondant sont soit tous les deux totalement occupés, soit contiennent tous les deux au moins une page libre. Si le bit vaut 1, alors au moins l'un des deux blocs est totalement occupé (et l'autre contient au moins une page libre).

Le noyau gère ainsi 10 listes de blocs de pages dont la taille – en page – est égal à 2^0 (1 page), 2^1 (2 pages), 2^2 (4 pages), .. . 2^9 (512 pages). Il effectue des demandes d'allocation pour des blocs de pages contiguës de taille supérieure à 1, notamment pour satisfaire les requêtes mémoires de composants ne connaissant pas la pagination, comme c'est le cas par exemple pour certains pilotes d'entrées-sorties.

Lorsque le noyau traite une demande d'allocation pour un nombre de i pages consécutives, il prend dans la liste $\log_2(i)$ de la table `free_area` le premier bloc libre. Si aucun bloc n'a été trouvé dans cette liste, le noyau explore les listes de taille supérieure j (j>i). Dans ce dernier cas, i cases sont allouées dans le bloc de j cases et les j-i cases restantes sont replacées dans les listes de taille inférieure.

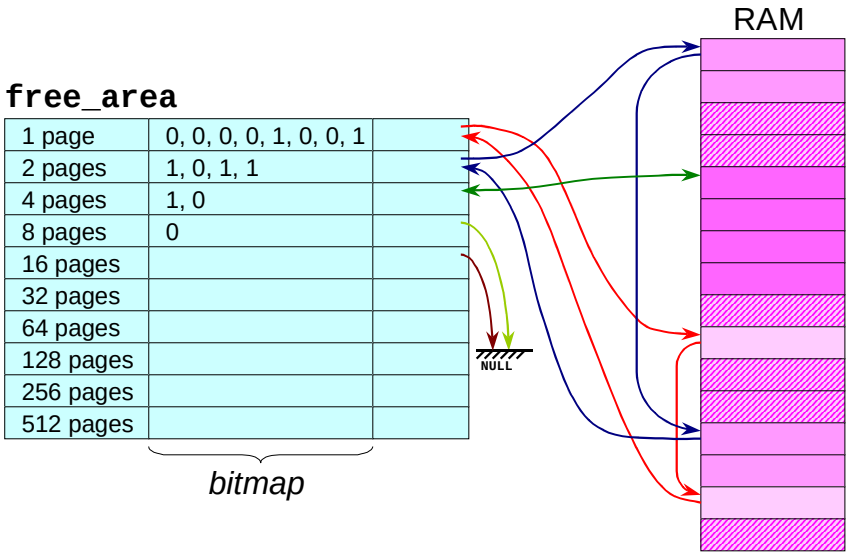


Table des blocs de pages libres `free_area`

Lorsque le noyau libère un bloc de i pages libres, il replace ce bloc dans une des listes de la table `free_area`, en tentant de fusionner ce bloc avec ses voisins si ceux-ci sont de même taille et si ils sont libres également. Ainsi, un bloc de 2 pages libres ayant un voisin comportant également 2 pages libres, forme un groupe de 4 pages libres. Ce groupe de 4 pages libres est inséré dans la liste de la troisième entrée de la table `free_area`, à moins qu'il ne comporte lui aussi un voisin de 4 pages libres, auquel cas il est fusionné avec lui pour former un bloc de 8 pages libres... et ainsi de suite jusqu'à former des groupes de 512 pages libres.

14 Systèmes de fichiers et implémentation

14.1 Introduction

La mémoire vive souffre de deux handicaps :

- elle est (relativement) coûteuse,
- elle perd ses données dès qu'elle n'est plus alimentée en électricité.

Les disques durs (et autres périphériques de masse) ne souffrent pas de ces lacunes, mais ils ont des temps d'accès qui les rendent inutilisable comme mémoire pour stoker des programmes exécutés par le CPU, ou pour stocker les variables de ces programmes.

Le compromis utilisé dans tous les systèmes d'exploitation consiste à utiliser la mémoire vive pour l'exécution des programmes (codes et données). Les exécutables (ceux qui sont chargés en RAM), les bibliothèques dynamiques, les données sources, et les résultats des traitements sont stockés sur les périphériques de masse.

Chaque programme (exécutable), chaque ensemble de données, chaque document, etc. est souvent stocké dans un « *container* » appelé ***fichier***. Classiquement, les fichiers sont identifiés par un nom.

Sous Linux, il existe différents types de fichiers (nous rentrerons dans le détail par la suite) :

- les containers de données, documents, programmes, bibliothèques, etc.
- les fichiers « *périphériques* » (souvent sous `/dev`),
- les fichiers de communication entre les processus (*pipes*),
- les fichiers d'interface entre l'utilisateur et le noyau (sous `/proc`),
- les liens symboliques.

Souvent, l'utilisateur a une vue logique du fichier : il le voit comme un container contenant ses données sous forme continue. En pratique, physiquement, les données sont découpées, et stockées sur le périphérique de masse, suivant un mécanisme qui permet de les retrouver, les modifier, et les effacer si nécessaire.

Parce que la gestion des fichiers deviendrait vite très difficile, tous les fichiers ne sont pas stockés « en vrac », mais sont classiquement rangés dans des « ***répertoires*** » ou « ***dossiers*** » (*directory* en Anglais). Dans les systèmes d'exploitation modernes, ces répertoires peuvent contenir des sous répertoires, qui eux même peuvent contenir des sous-répertoires, et ainsi de suite. L'ensemble de tous les répertoires, sous-répertoires, et fichiers forment une arborescence, comme par exemple (vue simplifiée) :

- `/` le répertoire racine
    - `/bin` : les fichiers exécutables (en binaire) pour l'initialisation du système, et les commandes "essentielles ;
    - `/boot` : le noyau vmlinuz et les fichiers de démarrage (grub ou lilo…) ;
    - `/dev` : répertoire de fichiers spéciaux, qui servent de canaux de communication avec les périphériques (disques, adaptateur réseau, etc...) ;
    - `/etc` : les fichiers de configuration du système et les principaux scripts de paramétrage ;
        - `/etc/rc.d` : scripts de démarrage du système
        - `/etc/X11` : scripts de configuration du serveur X
        - `/etc/sysconfig` : configuration des périphériques
        - `/etc/skel` : fichiers recopiés dans le répertoire personnel d'un nouvel utilisateur lorsque son compte est créé ;
    - `/home` : la répertoire contenant les répertoires personnels des utilisateurs ;

- `/lib` : les bibliothèques et les modules du noyau
- `/mnt` : la répertoire où sont classiquement « *montés* » les systèmes de fichiers des périphériques amovibles (CD-Rom, disquette, clé USB, système de fichier réseau, etc.). Nous reviendrons sur cette notion de « *montage* » ;
- `/opt` : lieu d'installation d'applications supplémentaires ;
- `/root` : répertoire personnel du super-utilisateur « root » ;
- `/sbin` : les fichiers exécutables pour l'administration du système ;
- `/tmp` : stockage des fichiers temporaires ;
- `/usr` : programmes accessibles à tout utilisateur. Sa structure reproduit celle de la racine `/` ;
- `/var` : données variables liées à la machine (fichiers d'impression, logs, traces de connexions http, etc.) ;
- `/proc` : pseudo-répertoire contient une « image » du système (par exemple, `/proc/kcore` est une image virtuelle de la RAM).

Le sous-ensemble du système d'exploitation qui assure la gestion cette arborescence, les fichiers qu'ils contiennent, etc. s'appelle le ***système de gestion de fichiers*** (*SGF*), ou tout simplement ***système de fichiers*** (*SF*), ***file system*** (*FS*) en Anglais. Voici une liste non exhaustive de systèmes de gestion de fichiers célèbres :

- `ext2`, `ext3` : Extented FS version 2 et 3 (Linux, BSD) ;
- `FAT` : File Allocation Table (DOS/Windows, Linux, BSD, OS/2, Mac OS X). Se décompose en plusieurs catégories :
  - `FAT12`,
  - `FAT16`,
  - `FAT32`,
  - `VFAT` ;
- `FFS` : Fast File System (BSD, Linux expérimental) ;
- `HFS` et `HFS+` : Hierarchical File System (Mac OS, Mac OS X, Linux) ;
- `HPFS` : High Performance FileSystem (OS/2, Linux) ;
- `minix fs` (minix, Linux) ;
- `NTFS` : New Technology FileSystem (Windows NT/2000/XP, Linux – écriture expérimentale –, Mac OS X en lecture seule) ;
- `ReiserFS` (Linux, BSD en lecture seule) ;
- `CFS` : Cryptographic File System - FS chiffré (BSD, Linux) ;
- `cramfs` : FS compressé (Linux en lecture seule) ;
- `EFS` : Encrypting File System : FS chiffré au-dessus de `NTFS` (Windows) ;
- `ISO 9660` : en lecture seule sur tous les systèmes lisant les CDROM/DVDROM ;
- `QNX4fs` : FS utilisé pour le temps réel (QNX, Linux en lecture seule) ;
- `UDF` : format de disque universel (système de fichiers des DVD-ROM et des disques optiques réinscriptibles tels les CD-RW, DVD±RW, etc.).

Certains *file systems* permettent d'accéder à des périphériques via un réseau :

- `AFS` : Andrew File System : (Aix, Linux expérimental) ;
- `Coda` : Systèmes de fichiers informatique (Linux) ;
- `NFS` : tous les Unix, Linux, Mac OS X (Windows pour la 4) ;
- `NCP` : NetWare Core Protocol (Novell NetWare, Linux en client seul) ;
- `SMB` : Server message block (Windows, Linux, BSD et Mac OS X via Samba) ;
- `CIFS` : Evolution de SMB, supporté par Samba ainsi que par Windows 2000 et XP.

14.2 Le fichier logique

Comme nous l'avons vu, l'utilisateur (le programmeur) voit un fichier de données comme un *container* qui stocke celles ci de façon linéaire. Les opérations classiques réalisées sur un fichier sont :

- ouvrir le fichier (soit en lecture seule, soit en écriture, ou en lecture+écriture),
- lire dans le fichier à partir d'une « *tête de lecture virtuelle* »,
- écrire dans le fichier à partir d'une « *tête de lecture virtuelle* »,
- dans certains cas, déplacer cette « *tête de lecture virtuelle* » dans le fichier,
- puis enfin, fermer le fichier.

Il est aussi possible de :

- renommer un fichier,
- le déplacer (le changer de répertoire),
- et dans certains systèmes, de créer un lien (physique ou logique) vers un fichier (nous verrons cette notion plus tard).

A noter que les « supports de masse » peuvent permettre des accès selon différents modes :

- les bandes magnétiques, lecteurs DAT ou streamer permettent des ***accès séquentiels*** (la lecture se fait « *en continu* »),
- les disques durs, CD-Rom, clés USB, etc. permettent des ***accès directs*** aux données (il est possible de déplacer la « *tête de lecture virtuelle* » à n'importe quel endroit du fichier),
- enfin, pour faciliter les recherches, il est possible d'utiliser une notion de clé, clé qui permet de déterminer la position d'une donnée sur un disque. On parlera alors d'***accès indexé*** (ou ***accès aléatoire***).

14.3 Généralités sur la résolution de nom

Comme nous l'avons vu, un fichier est identifié par un « *nom* ». Suivant les systèmes d'exploitation, les noms peuvent avoir des formes très différentes. Nous pouvons toutefois reconnaître certaines propriétés constantes :

- pour chaque disque, ou pour chaque partition (nous reviendrons sur ce concept de partition, mais retenez qu'il s'agit souvent de découper un disque physique en plusieurs disques virtuels, appelés alors « *partition* »), il existe un dossier qui contient tous les fichiers et sous-dossiers, appelé parfois « *répertoire racine* » ou tout simplement « *racine* » ;
- le « nom de fichier » contient le nom lui-même, mais peu aussi être préfixé de la suite de répertoire et sous-répertoires dans lequel il se trouve (on parle alors de « *chemin* » ;
- chaque programme qui s'exécute possède dans la plupart des OS un « ***répertoire courant*** », appelé aussi « ***répertoire de travail*** ». Très souvent, un processus hérite du répertoire courant de son père. Mais il arrive aussi (c'est souvent le cas par défaut dans les OS graphiques) que le répertoire courant soit le répertoire qui contient l'exécutable, ou soit une des propriétés de l'icône qui le représente. Des primitives permettent à un programme de changer son répertoire courant ;
- aussi, le « chemin » qui référence un fichier peut être relatif à ce répertoire courant (on parle alors de ***chemin relatif***), ou peut contenir toutes les informations qui permettent de le localiser (ordinateur, disque, partition, suite de répertoires/sous-répertoires, etc.). On parle alors de ***chemin absolu*** ;

- il existe un caractère séparateur permettant de différencier les différents répertoires/sous-répertoires dans un chemin (par exemple, il s'agit du slash « **/** » sous Unix/Linux, et de l'anti-slash « **** » sous Windows ;
- il existe un ou plusieurs caractère(s) qui traduisent le répertoire courant (il s'agit du point « **.** » pour Unix/Linux et Windows), et un ou plusieurs caractère(s) qui traduisent le répertoire parent (il s'agit de deux points à la suite « **..** » pour Unix/Linux et Windows). A noter que le répertoire parent de la racine est la racine elle-même.

L'ensemble théorique de tous les noms possibles s'appelle l'***espace de nom***. Dans les systèmes d'exploitations modernes, cet espace de nom peut contenir différents sous-ensembles, chacun étant géré par un *file system* différent. Certains de ces systèmes de gestions de fichiers permettent d'accéder à des périphériques locaux (par exemple, `FAT16` pour un lecteur de disquette, `ext3` pour un disque dur, `NTFS` pour un disque dur externe USB, etc.), et d'autres, à des périphériques accessibles via le réseau (par exemple, `NFS` pour accéder à un partage Unix, `SMB/CIF` pour accéder à un partage Windows/Samba, etc.).

A noter que chaque gestionnaire de gestion de fichiers possède ses propres propriétés en terme de :

- profondeur maximum d'imbrication des répertoires/sous-répertoires,
- nombre maximum de lettres dans un nom de répertoire ou de fichier,
- nombre maximum de lettres au total dans le nom,
- de différentiation majuscules/minuscules (FS *case sensitive* ou pas),
- la prise en charge (ou pas) des « localisations » (caractères accentués, support de plusieurs pages de codes, etc.),
- niveau de sécurité (gestion ou pas de notion d'utilisateur propriétaire, de groupes, d'*access list* – ACLs –, etc),
- propriétés (gestions de notions de fichiers cachés, de fichiers exécutables, de droits en terme de lecture/écriture, date de création, date de dernier accès, date de dernière modification, etc.) ;
- de chiffrement (certain FS permettent de crypter les données contenues),
- de compression (certains FS permettent de compresser des données pour gagner de la place),
- de *journalisation* (chaque opération d'écriture est journalisée dans un fichier spécial ; ainsi, en cas d'arrêt violent du système, il est possible de réparer rapidement la structure du *file system* à l'aide de ce journal, sans avoir à parcourir tout le disque) ;
- etc.

A noter qu'un informaticien, *Tim Berners-Lee*, a proposé dans le RFC 1630, mis à jour par le RFC 2396, puis par le 3986, une norme standard de nomenclature : l'***URI*** (*Uniform Resource Identifier*). Il est probable que les futures versions des OS sauront lire directement les URIs.

14.4 Le fichier physique

14.4.1 Le disque dur

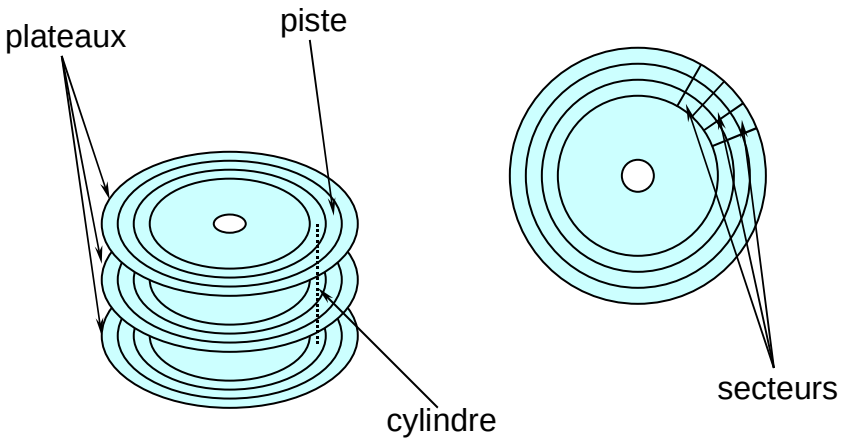
Chaque disque dur est constitué :

- d'un ensemble de plateaux (souvent, jusqu'à 20), ce qui forme une pile de disques ;
- chaque plateau possède 2 faces ;

- chaque face est découpée en « pistes » (cercles concentriques). Suivant les modèles, chaque plateau peut être découpé de 10 à plus de 1000 pistes. A noter que chaque ensemble de pistes situées les unes au dessus des autres sur les différents plateaux s'appellent des cylindres ;
- enfin, chaque piste est découpée en secteurs (souvent, de 4 à 32 secteurs par piste). Selon les modèles, toutes les pistes peuvent contenir le même nombre de secteurs, ou avoir un nombre de secteurs plus petit pour les pistes proches de l'axe, et plus grand pour les pistes qui en sont éloignées. Quoi qu'il en soit, chaque secteur contient en général de 32 à 4096 octets (couramment : 512 octets).

L'opération qui consiste à créer une telle structure logique à partir du disque physique s'appelle le ***formatage***.

Les têtes de lecture/écriture sont fixées à un bras, faisant ressembler l'ensemble à un peigne. Un disque peut donc lire en parallèle tous les secteurs situés les uns au dessus des autres.



Structure physique d'un disque dur

Pour optimiser les accès aux secteurs, ceux-ce sont regroupés en « blocs ». Le bloc constitue la plus petite unité d'échange entre le disque et la mémoire centrale. Vous l'aurez compris, la taille d'un bloc dépend fortement de la structure physique du disque.

14.4.2 Les méthodes d'allocation des mémoires secondaires

Physiquement, les fichiers logiques sont découpés en morceaux, eux même stockés dans des blocs. Or les fichiers se constituent, grandissent, rapetissent, s'effacent, d'autres sont créés... l'espace disque libre subit le même phénomène que la mémoire centrale : il se fragmente.

Tout l'art des gestionnaires de fichiers consiste à :

- proposer une structure permettant de stocker sur le disque l'espace de noms (comment l'utilisateur aura découpé l'espace de nom en répertoire, sous-répertoire, etc.),
- allouer les blocs libres aux fichiers qui se constituent et grandissent afin d'éviter la fragmentation.

On retrouve principalement quatre méthodes d'allocation des blocs :

- la **méthode l'allocation contiguë** : cette méthode suppose que le fichier sera stocké sur des blocs qui se suivent physiquement. Cette méthode est simple et efficace, mais peu utilisée : elle impose au programmeur de connaître à l'avance la taille finale du fichier. Comme pour la mémoire centrale, il existe deux algorithmes permettant l'allocation contiguë :
    - l'allocation « ***first fit*** » : la première zone de blocs contiguë de taille supérieure ou égale à celle du fichier est allouée,
    - l'allocation « ***best fit*** » : le système recherche l'espace libre de a taille la plus petite possible pouvant contenir le fichier ;
- la **méthode d'allocation par zones** : il s'agit d'une extension de la méthode précédente. Ici, un fichier peut-être constitué de zones. Chaque zone est constituée de blocs contiguës, mais les zones ne sont pas nécessairement contiguës. Le début du fichier est stocké dans une « *zone primaire* », le reste étant stocké dans des « *zones secondaires* », qui sont allouées au fur et à mesure que le fichier grandit. Souvent, le nombre de zones secondaires est limité (limitant ainsi la possibilité d'extension d'un fichier). Les temps d'accès peuvent être moins bons (si les zones ne sont pas physiquement contiguës, le contrôleur de disque devra déplacer le bras pour changer les têtes de place), mais cette méthode permet à un fichier de grandir ;
- la **méthode d'allocation par blocs chaînés** : dans cette méthode, le fichier est disposés dans une suite de blocs chaînés entre eux, qui peuvent être disposés physiquement n'importe où sur le disque. Pour étendre un fichier, il suffit de prendre un bloc libre, et de créer un lien entre le dernier bloc alloué précédemment et ce nouveau. Cette méthode possède deux inconvénients majeurs : la seule méthode de lecture est la lecture par accès séquentiel ; de plus, elle occupe de la place sur le disque pour stocker le chaînage ;
- la **méthode d'allocation indexée** : dans cette méthode, toutes les adresses des blocs physiques constituant un fichier sont rangés dans une table appelée index, elle même contenue dans un bloc du disque. A la création d'un fichier, un index est créé, avec toutes ses entrées initialisées à 0. Ces entrées sont ensuite mises à jour au fur et à mesure que le fichier grandit. Ce regroupement de toutes les adresses des blocs d'un fichier dans une même table permet un accès direct. De plus, les blocs alloués au fichier ne contiennent que des données du fichier (et plus de liens vers d'autres blocs). Apparaissent néanmoins deux problèmes :
    - si la taille du fichier est petite, et que la taille d'un blocs est grand, nous perdons beaucoup de place ;
    - inversement, si le fichier est grand, un seul bloc peut ne pas suffire à stocker tous les blocs composant un fichier. Dans ce cas, la solution est de réaliser un index multiniveaux : le premier bloc d'index ne contient pas une liste du blocs contenant les données du fichier, mais une liste de blocs d'index. Ainsi, la taille maximale potentielle d'un fichier est agrandie exponentiellement. Par contre, il faudra lire deux blocs (et non plus un seul) avant de pouvoir accéder aux données d'un fichier. Evidemment, plus le nombre de niveaux est important, plus le temps passé à suivre les indexes est important, réduisant ainsi les performances.

14.4.3 La gestion de l'espace libre

Le système conserve une liste d'espace libre, qui mémorise tous les blocs libres du disque. Lors de la création ou de l'agrandissement d'un fichier, le système cherche dans cet espace des blocs pour les alloués au fichier. Ces derniers sont alors ôtés de la liste. Lors de la destruction d'un fichier, les blocs le constituant sont alors remis dans cette liste des blocs libres.

Parmi les méthodes utilisées pour recenser l'espace libre, deux sont principalement utilisées :

- la **gestion de l'espace libre par vecteur de bits** : dans cette méthode, un tableau binaire (ne contenant que des 0 et des 1) indique la disponibilité d'un blocs. Si le $i^{ème}$ bit du tableau est à 0, le blocs numéro i est libre. Sinon, il est occupé. La longueur de ce vecteur binaire est donc égal au nombre de blocs dans le disque ;
- la **gestion de l'espace libre par liste chaînée** : ici, les blocs libres sont chaînés entre eux. L'inconvénient ce cette méthode est que si nous avons besoin d'allouer N blocs consécutifs, nous seront peut-être obligé de parcourir une longue liste de blocs. Une variante de cette méthode consiste à regrouper les blocs libres dans une zone, et d'indiquer dans chaque premier bloc d'une zone libre le nombre de blocs contigus, puis le lien vers le premiers blocs de la zone suivante (constituée elle aussi de blocs contigus).

14.4.4 Les partitions

Comme nous l'avons déjà vu, le nombre de fichiers pouvant être stockés dans un disque peut être très important. Pour éviter que tous les fichiers soient stockés uniformément dans le même disque physique, ce dernier peut être découpé en un ou plusieurs disques logiques : on parle alors de ***partitions***, ou de ***volumes***.

Chaque partition peut être repérée par un nom, appelé ***label***.

En ce qui concerne les PC, les informations liées aux partitions sont stockés dans une table, appelée ***table des partitions***. Cette table des partitions était stockée dans le premier bloc du disque (appelé MBR, pour *master boot record*). Comme ce bloc est de taille réduite, le système ne pouvais gérer historiquement que quatre partitions. Or, ce chiffre s'étant par la suite révélé insuffisant, une astuce a consisté à typer les partitions. Le système doit posséder au minimum une ***partition primaire*** (qui sont les seules à pouvoir contenir un secteur d'amorçage, qui contient le chargeur du noyau), et au maximum une ***partition secondaire*** (ou ***partition étendue***), qui doit toujours être la dernière. Celle-ci peut alors contenir autant de lecteurs logiques que souhaités, qui sont autant de partitions (mais qui ne peuvent être amorçables).

14.4.5 Le montage d'une partition

Sous Windows, chaque disque, ou chaque partition est identifié avec une lettre de lecteur (« A: » et « B: » sont généralement réservés pour les lecteurs de disquettes, « C: » pour la partition amorçables, etc.

Sous Unix/Linux, une partition doit être montée sous un répertoire. Ainsi, il est nécessaire d'avoir au minimum une partition : la partition sous laquelle sera montée la racine de l'espace de noms (on parle aussi de ***partition root*** dans la littérature anglaise).

La création d'un lien « répertoire → partition » s'appelle le montage. Cette opération se fait automatiquement à l'initialisation du système (la partition « `root` » à monter au démarrage est un paramètre du noyau, souvent fourni par le « *boot loader* » (fréquemment lilo ou grub pour Linux) ; puis, les scripts d'initialisation regardent alors les différents montages qu'ils doivent réaliser dans le fichier de configuration « `/etc/fstab` »). Cette opération de montage/démontage peut ensuite être effectuée à tout moment par le super utilisateur, à l'aide des commandes « `mount` » et « `umount` » (qui, elles-même, font appel à des primitives système « `mount()` » et « `umount()` »).

A noter que les différentes partitions peuvent toutes être gérées par un « *file system* » différent. Par exemple, la racine peut être une partition de type `ext3`. Le dossier « `/home` » peut se voir monter une partition de type « `reiserfs` », le répertoire « `/mnt/floppy` » se voit monter une partition de type « `vfat` », alors que le répertoire « `/mnt/cdrom` » se voit monter une partition de type « iso9660 ».

14.5 Exemple de système de gestion de fichier : ext2

Historiquement, Linux était un projet dont le but avoué était de pouvoir lire des partitions de type `minix` (gestionnaire de partition très limité : adressage des blocs sur 16 bits – ce qui limite la taille des fichiers à 64 Ko –, un nombre de 16 entrées maximum dans chaque répertoire, et des noms de fichiers ne pouvant excéder 14 caractères).

Depuis, Linux a été profondément remodeler pour acquérir une couche virtualisant les *file systems* (Cf. prochain chapitre), ainsi que des systèmes de fichiers plus évolués et ayant moins de contraintes. Nous allons étudier dans ce chapitre l'un des plus connu d'entre eux : le gestionnaire de fichiers `ext2`.

14.5.1 La structure d'i-node

Un fichier physique Linux est identité par un nom et toutes les informations le concernant sont stockées dans un descripteur appelé ***i-node*** ou ***i-nœud***. Ensuite, le fichier n'a pas de structure logique et est simplement constitué comme une suite d'octets.

Les blocs sont alloués au fichier selon une organisation indexée. Chaque bloc est identité par un numéro logique qui correspond à son index dans la partition du disque et est codé sur 4 octets. La taille d'un bloc devant être une puissance de 2, multiples de la taille d'un secteur (généralement 512 octets), elle varie selon les systèmes entre 512, 1'024, 2'048 ou 4'096 octets.

L'inode du fichier est une structure stockée sur le disque, allouée à la création du fichier, et est repérée par un numéro. Un i-node contient les informations suivantes:

- le nom du fichier ;
- le type du fichier et les droits d'accès associés ;
- les identificateurs du propriétaire du fichier ainsi que du groupe ;
- diverses heures telles que l'heure de dernier accès au fichier, l'heure de dernière modification du contenu du fichier, l'heure de dernière modification de l'inode, l'heure de suppression du fichier ;
- le nombre de liens vers d'autres fichiers ;
- la taille du fichier (en octets) ;
- le nombre de blocs de données alloués au fichier ;
- deux listes de contrôle d'accès respectivement pour le fichier et pour le répertoire ;
- une table d'adresses des blocs de données.

Les types de fichier sous linux sont :

- les fichiers réguliers (ou normaux) : « *regular files* » sont les fichiers les plus courants. Ils sont destinés à stocker les données des utilisateurs quel que soit le type de ces données. Ces fichiers sont sans organisation et ne constituent qu'une suite d'octets, caractérisée par sa longueur ;
- les répertoires ;
- les liens symboliques. Il s'agit de pointeurs (alias) vers un autre fichier. Toute action effectuée sur le contenu d'un lien équivaut a une action sur le fichier pointé ;
- les fichiers de périphériques. Se sont des fichiers liés a un pilote d'entrées-sorties au sein du noyau. Une opération de lecture ou d'écriture réalisée sur le fichier équivaut à une action sur le périphérique. On distingue à ce niveau :
    - o les fichiers spéciaux en mode bloc (ex : disques), pour lesquels les échanges s'effectuent bloc après bloc,
    - o et les fichiers spéciaux en mode caractère (ex : les terminaux) pour lesquels les échanges s'effectuent caractère après caractère.

Ces fichiers spéciaux, ainsi que le mécanisme des entrées-sorties feront l'objet d'un prochain chapitre ;

- les tubes nommés (pipes) et les sockets.

A noter que les liens symboliques, les tubes nommés, les sockets et les fichiers de périphériques sont des fichiers dont la structure physique se résume a l'allocation d'un i-node. Aucun bloc de données ne leur est associé.

14.5.2 Les droits classiques sur les fichiers sous Unix/Linux

Les droits des fichiers sont appliqués à trois groupes de comptes utilisateurs :

- le propriétaire du fichier,
- le groupe qui contient le propriétaire du fichier,
- tous les autres comptes, qui ne sont ni le propriétaire, ni un membre du groupe auquel il appartient.

Trois types de permissions peuvent être donnés à des fichiers :

- la permission en lecture,
- la permission en écriture,
- la permission en exécution.

En interne, une constante en octale est liée à chacun de ces droits :

- 4 pour la permission en lecture,
- 2 pour la permission en écriture,
- 1 pour la permission en exécution.

Par exemple, un droit en lecture+exécution vaudra 4+1 = 5. Traditionnellement, les droits sont représentés en octal sous la forme : 0XYZ, avec X le droit pour le propriétaire, Y le droit pour le groupe auquel il appartient, et Z le droit pour les autres utilisateurs. Par exemple, un droit de « 0750 » signifie que le propriétaire peut tout faire (lire, écrire, et exécuter), le groupe auquel il appartient peut l'exécuter et le lire, alors que les autres utilisateurs ne peuvent accéder au fichier. Pour un répertoire, le droit « x » signifie « peut accéder au contenu du répertoire ».

Enfin, 3 autres bits ont une signification particulière :

- le bit « *setuid* » : un exécutable pour lequel ce bit est positionné s'exécute avec l'identité du propriétaire du fichier et non avec celle de l'utilisateur qui a demandé l'exécution ;
- le bit « *setgid* » : un exécutable pour lequel ce bit est positionne s'exécute avec l'identité du groupe du propriétaire du fichier, et non avec celle de l'utilisateur qui en a demandé l'exécution ;
- enfin, le bit « *sticky* » : les fichiers appartenant a un répertoire pour lequel ce bit est positionné ne peuvent être supprimés que par leurs propriétaires.

La commande shell « `ls -al` » permet de lister tous ces attributs pour les fichiers d'un répertoire :

- le premier champ code le type du fichier et les droits associes au fichier. Le type (1 caractère) est codé par une lettre qui prend la valeur suivante :
  - `d` pour un répertoire (*directory*),
  - `p` pour un tube (*pipe*),
  - `l` pour un lien symbolique,
  - `b` pour un périphérique « bloc »,
  - `c` pour un périphérique « caractère »,
  - `-` pour un fichier ordinaire ;

Les droits d'accès sont codés par groupe de trois caractères, correspondant a chacune des permissions possibles pour chacune des trois entités citées au paragraphe précédent. Ainsi « `rwxr--r-x` » indique que l'utilisateur peut lire, écrire, et exécuter le fichier (premier groupe « `rwx` »), le groupe peut seulement lire le fichier (deuxième groupe « `r--` »), tandis que les autres peuvent lire et exécuter le fichier (troisième groupe « `r-x` ») ;

- le deuxième champ indique le nombre de liens pour le fichier ;
- le troisième champ indique le nom de l'utilisateur propriétaire du fichier ;
- le quatrième champ indique le nom du groupe ;
- le cinquième champ indique le nombre d'octets composant le fichier ;
- le sixième champ indique la date de dernière modification du fichier ;
- le dernier champ indique le nom du fichier.

La commande shell « `chmod droit fichier` » permet de changer les droits du fichier « `fichier` », afin de les positionner à « `droit` ». Cf. « `man chmod` » pour plus de détails.

14.5.3 Liens physiques et liens symboliques

Le système de gestion de fichiers `ext2` manipule deux types de liens : les liens physiques, et les liens symboliques.

Le lien physique correspond à l'association d'un nom a un fichier (et donc, a un i-node). Un même fichier peut recevoir plusieurs noms différents, dans un même répertoire, ou dans des répertoires différents d'une même partition. Créer un lien physique sur un fichier équivaut a créer une nouvelle entrée dans un répertoire pour mémoriser l'association « nom de fichier <-> i-node ». Un lien physique ne peut pas être créé sur un répertoire.

Le lien symbolique correspond à la création d'un fichier de type particulier (de type « `l` »), qui contient comme donnée un pointeur vers un autre fichier (+/- le raccourcis sous Windows).

La commande « `ln nomfichier1 nomfichier2` » crée un lien physique sur le fichier `nomfichier1`, en lui attribuant un nouveau nom `nomfichier2`.

La commande « `ln -s nomfichier1 nomfichier2` » crée un lien symbolique sur le fichier `nomfichier1`, en créant le fichier `nomfichier2` qui contient comme donnée le chemin d'accès au fichier `nomfichier1`.

14.5.4 L'allocation des blocs de données

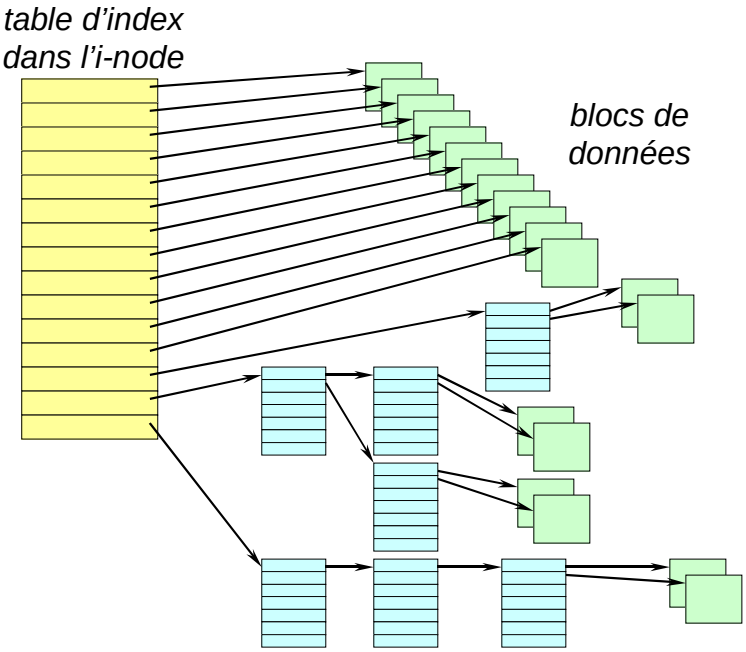
L'i-node du fichier contient un tableau de `EXT2_NBLOCKS` entrées, contenant chacune un numéro de bloc logique. Par défaut la valeur de `EXT2_NBLOCKS` est égale à 15.

L'organisation de cette table suit l'organisation indexée s'inspirant de celle que nous avons vus précédemment, à une différence près, qui réalise une optimisation.

Ainsi :

- les 12 premières entrées du tableau contiennent un numéro de bloc logique correspondant directement à un bloc de données du fichier ;
- la treizième entrée sert de premier niveau d'index. Elle contient un numéro de bloc logique contenant lui-même des numéros de blocs logiques qui correspondent à des blocs de données ;
- la quatorzième entrée sert de deuxième niveau d'index. Elle contient un numéro de bloc logique contenant lui-même des numéros de blocs logiques qui contiennent à leur tour les numéros de blocs logiques correspondant a des blocs de données ;

- enfin, la quinzième entrée introduit un niveau d'indirection supplémentaire.



Index multiniveaux dans les blocs

Adressage des blocs constituant un fichier ext2

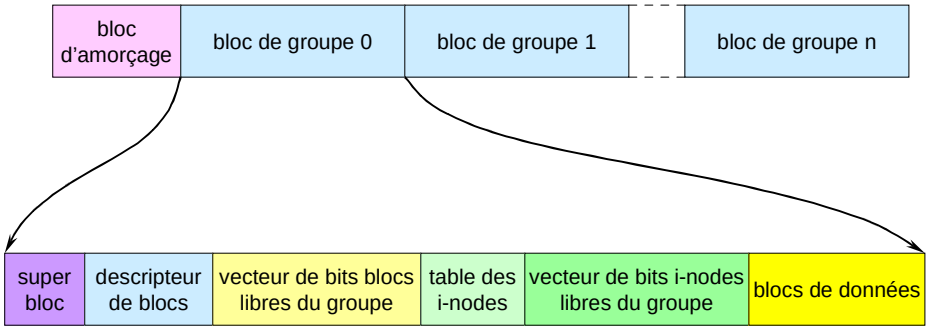
Lors de la création d'un fichier, aucun bloc de données ne lui est alloué. Les blocs sont attribués au fichier au fur et à mesure de son extension, en commençant par allouer les 12 premières entrées de la table, puis en emplissant ensuite le premier niveau d'indirection, puis le second et enfin le troisième si besoin est.

Pour des raisons d'efficacité, et de façon à réduire la fragmentation du fichier, le système pré-alloue des blocs de données. Plus précisément, lorsque le système alloue un nouveau bloc pour le fichier, il pré-alloue en même temps jusqu'à 8 blocs adjacents. Ces blocs pré-alloués sont libérés lorsque le fichier est fermé, s'ils n'ont pas été utilisés.

14.5.5 Structure des partitions

Une partition Linux/`ext2` est composée d'un bloc d'amorçage, utilisé lors du démarrage du système, puis d'un ensemble de groupe de blocs, chaque groupe contenant des blocs de données et des i-nodes enregistrés dans les pistes adjacentes. En effet, chaque groupe de blocs contient les informations suivantes :

- une copie du super bloc (Cf. ci-dessous) du système de gestion de fichier,
- une copie des descripteurs de groupe de blocs (Cf. ci-dessous),
- un vecteur de bits pour gérer les blocs libres du groupe,
- un groupe d'i-nodes, souvent appelé « table des i-nodes ». Le numéro de l'i-node est égal à son rang dans cette table,
- un vecteur de bits pour gérer les i-nodes libres,
- des blocs de données.



Structure d'une partition ext2

Le ***super bloc*** contient des informations générales sur la partition, tel que :

- le nom de la partition ;
- des statistiques comme l'heure de la dernière opération de montage, et le nombre d'opérations de montage réalisées sur la partition ;
- la taille de la partition (en bloc) ;
- le nombre total d'i-nodes dans la partition ;
- la taille d'un bloc dans la partition ;
- le nombre de blocs libres et d'i-nodes libres dans la partition ;
- le nombre de blocs et d'i-nodes par groupe ;
- l'heure du dernier contrôle de cohérence et l'intervalle de temps entre chaque contrôle de cohérence.

Ce super-bloc est dupliqué dans tous les groupes de blocs. Le système utilise couramment la copie placée dans le groupe de blocs 0. La mise à jour des copies placées dans les autres groupes de blocs a lieu lors des contrôles de cohérence du système de gestion de fichiers (commande « `/sbin/e2fsck` »). Si une corruption est constatée sur le super-bloc du groupe 0, alors les copies redondantes sont utilisées pour ramener la partition à un état cohérent.

Chaque groupe de blocs est associé à un ***descripteur de groupe de blocs*** qui contient les informations suivantes :

- les numéros des blocs contenant les vecteurs de bits gérant la liste des blocs libres et la liste des i-nodes libres ;
- le nombre de blocs libres et le nombre d'i-nodes libres dans le groupe ;
- le nombre de répertoires dans le groupe ;
- le numéro du premier bloc contenant les i-nodes libres.

D'une façon similaire au super-bloc, les descripteurs des groupes de blocs sont dupliqués dans tous les groupes de blocs. Les mises à jour sont effectuées lors des contrôles de cohérences de la partition.

Afin de réduire la fragmentation, le système Linux tente d'allouer un nouveau bloc à un fichier dans le groupe de blocs contenant le dernier bloc alloué pour ce même fichier. Sinon, il cherche un bloc libre dans le groupe de blocs contenant l'inode du fichier.

La liste des blocs libres et la liste des i-nodes libres sont gérées par l'intermédiaire de deux vecteurs de bits. Un bit égal à 0 signifie que le bloc de donnée ou l'i-node correspondant est libre, et la valeur 1 signifie au contraire que le bloc de donnée ou l'i-node correspondant est occupé. Chacun des vecteurs de bits est entièrement compris dans un seul bloc du disque.

14.6 Le système de gestion de fichiers virtuel VFS

14.6.1 Introduction

Comme nous l'avons déjà vu, Linux est capable d'avoir, dans son arborescence globale (dans son espace de noms) plusieurs *file systems*. Comme il est inconcevable que l'utilisateur (le programmeur) ait à utiliser plusieurs primitives selon le *file system* qui gère le fichier auquel il accède, Linux utilise une couche virtuelle entre le file système réel et l'utilisateur. Cette couche virtuelle se présente comme un système de gestion de fichiers virtuel, appelé ***VFS*** (***Virtual File System***).

L'utilisateur appelle les primitives de VFS sur un fichier. Selon le *file system* qui gère le fichier en question, le système traduira ces appels aux routines de VFS en leur équivalent pour le *file system* réel en effectuant les opérations suivantes :

- vérification des paramètres ;
- conversion du nom de fichier en numéro de périphérique et numéro d'i-node ;
- vérification des permissions ;
- appel a la fonction spécifique au système de gestion de fichiers concerné.

En plus de cette normalisation des appels aux primitives de gestion de fichier, VFS gère deux systèmes de cache :

- un cache de noms qui conserve les conversions les plus récentes des nom de fichiers en numéro de périphérique et numéro d'i-node ;
- un cache de tampons disque appelé *buffer cache*, qui contient un ensemble de blocs de données lus depuis le disque.

14.6.2 Structures et fonctionnement de VFS

Les structures du VFS créent un modèle de fichier, pour tout système de gestion de fichiers, qui est très proche du modèle de fichier natif de Linux `ext2`. Ce choix permet au système de travailler sur son propre système de gestion de fichiers avec un minimum de surcharge.

La structure du VFS est organisée autour d'objets auxquels sont associés des traitements. Les objets sont :

- l'objet « ***système de gestion de fichiers*** » (super-bloc), représenté par une structure « super-bloc » (`struct super_block`, définie dans `<linux/fs.h>`), qui contient des informations sur les caractéristiques du système de gestion de fichiers, telles que :
    - la taille des blocs,
    - les droits d'accès,
    - la date de dernière modification,
    - le point de montage sur l'arborescence de fichiers.

  Cette structure offre un ensemble d'opérations permettant de manipuler le système de gestion de fichiers associé (champ `struct super_operations *s_ops`), notamment lire ou écrire un i-node, écrire le super-bloc, et obtenir des informations

sur le système de gestion de fichiers. L'ensemble des structures `super_block` est regroupé dans une liste doublement chaînée ;

- l'objet ***« i-node »*** : décrit par un i-node (structure `struct inode` définie dans `<linux/fs.h>`). Cette structure contient des informations sur le fichier associé à l'i-node de même nature que celles trouvées dans un i-node de type `ext2`. Elle contient par ailleurs des informations spécifiques liées au type de système de gestion de fichiers, ainsi que des méthodes permettant de manipuler la structure (`struct inode_operations *i_op`). L'ensemble des i-nodes est géré de deux façons différentes :
  - une liste circulaire doublement chaînée regroupe l'ensemble des i-nodes ;
  - et pour la recherche rapide d'un i-node particulier, l'accès s'effectue par le biais d'une table de hachage, avec pour clé de recherche, l'identifiant du super-bloc et le numéro de l'inode dans ce super-bloc.
- l'objet « ***fichier*** » (file) : chaque fichier ouvert est décrit par une structure `struct file` (définie dans `<linux/fs.h>`). Cette structure contient un ensemble d'informations utiles pour la réalisation des opérations de lecture et d'écriture sur le fichier, telles que le type d'accès autorisé, la position courante dans le fichier, le nombre de processus ayant ouvert le fichier, etc. Comme pour les structures précédentes, la structure `struct file` inclut un ensemble de méthodes (`struct file_operations *f_op`) permettant de manipuler la structure, qui sont les opérations pour la lecture, l'écriture, l'ouverture, ou la fermeture d'un fichier, etc. ;
- l'objet « ***nom de fichier*** » (dentry) : les objets nom de fichier (ou *dentry*) sont créés pour mémoriser les entrées de répertoire lues lors de la recherche d'un fichier. Ainsi la recherche du fichier « `/home/manu/test.c` » aboutit à la création de trois objets *dentry* (un pour `home`, le suivant pour `manu`, le denier pour `test.c`), chaque objet *dentry* effectuant le lien entre le nom et le numéro d'inode associé. Ces objets sont par ailleurs gérés par le cache des noms.

14.6.3 Accès à VFS par un processus

Chaque processus accède à un fichier ouvert en utilisant un « ***descripteur de fichier*** ». Ce descripteur de fichier est en fait un numéro d'index dans la ***table des fichiers ouverts***.

En pratique, chaque entrée de cette table des fichiers ouverts, appelé *fd* (descripteur de fichier), pointe vers un objet de type *fichier*. Lui-même pointe à son tour vers un objet de type *dentry*, afin d'effectuer la liaison entre le nom du fichier et son i-node. Cet i-node pointe à son tour ver l'objet « *super-blocs* » décrivant le système de gestion de fichiers auquel appartient le fichier.

Par ailleurs, le champ *fs* du bloc de contrôle du processus (PCB) pointe sur un objet *fichier*, qui correspond au répertoire courant du processus.

A noter que qu'à la création de chaque processus, le fils hérite des fichiers ouverts du père (la table des fichiers ouverts est dupliquées). De plus, 3 fichiers sont toujours déjà ouverts :

- l'entrée numéro 0 dans la table des descripteurs de fichiers correspond à l'entrée standard, appelée ***stdin***, correspondant par défaut au clavier ;
- l'entrée numéro 1 dans la table des descripteurs de fichiers correspond à la sortie standard, appelée ***stdout***, correspondant par défaut à l'écran ;
- enfin, l'entrée numéro 2 dans la table des descripteurs de fichiers correspond à la sortie des messages d'erreur, appelée ***stderr***, correspondant par défaut à l'écran.

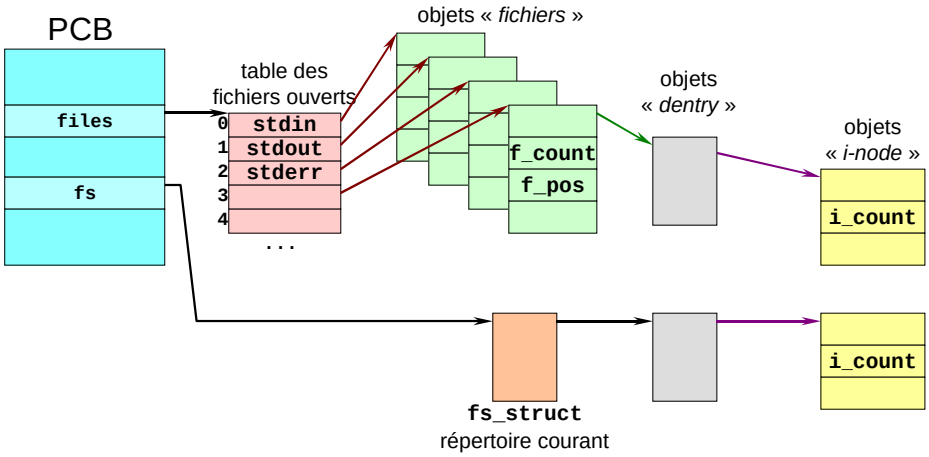


Table des fichiers ouverts, et relations entre les objets de VFS

Un champ « `f_count` » permet de savoir par combien de processus un fichier est ouvert (appelé « *nombre de référence* »). Le fichier est libéré (et la structure est effacée) lorsque que ce compteur est à zéro. De plus, le champ « `f_pos` » indique la position courante de la tête de lecture virtuelle dans le fichier (appelé « *pointeur du fichier* »).

Comme exercice, regardons les actions effectuées par le VFS lors de l'ouverture d'un fichier (appel à la primitive `desc = open(nom_fichier, mode_ouventure)`) :

- la routine d'enveloppe correspondant a la fonction `open()` lève une trappe, bascule le processus appelant en mode superviseur, puis appelle la routine système `sys_open()` après avoir passé les paramètres d'exécution (nom du fichier et mode d'ouverture) ;
- une nouvelle entrée est allouée dans la table des fichiers ouverts et un nouvel objet *fichier* pointé par cette entrée est créé ;
- le nom de fichier est converti en numéro de périphérique et i-node, et l'objet *dentry* correspondant est associé au fichier ;
- l'i-node associée au fichier est obtenue (méthode `look_up` de l'objet i-node représentant le répertoire contenant le fichier) ;
- les opérations concernant le fichier (champ `*f_ops` de l'objet *fichier*) sont initialisées depuis le champ `default_file_ops` de la structure `i_op` de l'i-node correspondant au fichier ;
- la méthode `open` de l'objet *fichier* est appelée pour réaliser l'ouverture physique du fichier en fonction du type du système de gestion de fichiers ;
- l'index de l'entrée de la table des fichiers ouverts pointant sur l'objet fichier est retourné a l'utilisateur. Elle constitue le descripteur `desc` du fichier.

14.6.4 Le fonctionnement du cache des *dentry* (*dcache*)

Lorsque le VFS doit lire un fichier depuis le support de masse, il doit au préalable convertir le nom logique du fichier vers son équivalent physique, à savoir le numéro du périphérique contenant le fichier, et le numéro de l'inode descriptive du fichier. Prenons par exemple la recherche du fichier dont le chemin est « `manu/tp/test.c` » pour le compte de l'utilisateur `manu`. Le traitement de ce chemin d'accès suit les étapes suivantes :

- le chemin ne commençant pas par « / », la recherche s'effectue depuis le répertoire courant du processus, dont l'i-node est repérée depuis le bloc de contrôle du processus. Si le chemin du fichier avait commencé par « / », le chemin absolu aurait été analysé à partir de l'i-node de la racine du système de gestion de fichiers, qui est également conservée en mémoire. En utilisant l'i-node décrivant le répertoire courant, le VFS recherche dans les blocs de données de ce fichier répertoire l'entrée concernant « `manu` ». L'entrée trouvée, le VFS connaît le numéro d'i-node associé à « `manu` ». Un objet *dentry* est créé pour ce couple ;
- l'inode décrivant le répertoire « `manu` » est lue depuis le disque, et l'entrée concernant « `tp` » est recherchée dans les blocs de données de ce fichier répertoire. L'entrée trouvée, le VFS connaît le numéro d'i-node associé à « tp ». Un objet *dentry* est créé pour ce couple ;
- l'inode décrivant le répertoire « `tp` » est lue depuis le disque et l'entrée concernant « `test.c` » est recherchée dans les blocs de données de ce fichier répertoire. L'entrée trouvée, le VFS connaît le numéro d'i-node associé a « `test.c` ». Un objet *dentry* est créé pour ce couple.

Cet exemple montre que la recherche d'un i-node associée à un fichier est un traitement long et coûteux. Aussi, le VFS maintient-il un cache de tous les objets *dentry* créés au cours de cette recherche. Ainsi, lorsque le VFS cherche le numéro d'i-node associé à un nom de répertoire, il regarde tout d'abord dans le cache si un objet *dentry* concernant ce nom existe. Si oui, il récupère le numéro d'i-node associé sans lecture disque. Sinon, il lit les blocs de données associés au répertoire depuis le disque jusqu'à trouver l'entrée recherchée.

14.6.5 Le cache des blocs disque (*buffer cache*)

Le VFS maintient également un cache contenant les blocs disque les plus récemment accédés. Lors de la lecture d'un bloc, celui-ci est place dans une zone mémoire appelée tampon (*buffer*). Ainsi, lorsque le VFS cherche a accéder a un bloc du disque, il consulte d'abord l'ensemble des tampons présents dans le cache. Si le tampon correspondant est présent, alors l'opération de lecture ou d'écriture est réalisée dans le cache. Sinon, le bloc est lu depuis le disque et place dans un tampon libre s'il en existe un. Sinon, le VFS récupère le tampon le moins récemment utilisé pour y placer le nouveau bloc dont il a besoin.

Remarque : dans le cas d'une écriture, le bloc est modifié en mémoire centrale sans être recopié immédiatement sur le disque. L'écriture du bloc modifié sur le disque n'est effectuée que lorsque le tampon contenant ce bloc est libéré pour y placer un nouveau bloc. Il s'ensuit qu'un arrêt intempestif du système peut entraîner une perte de données pour tous les tampons dont le contenu modifié n'a pas été recopié sur le disque.

C'est pourquoi il existe deux threads noyau, `bdflush` et `kupdate`, responsables de la gestion des tampons dans le *buffer cache* et de leur écriture sur le disque si leur contenu a été modifié :

- `bdflush` est réveillé lorsque trop de tampons modifiés existent dans le cache, ou lorsque la taille du cache devint insuffisante par rapport au nombre de tampons requis ;
- `kupdate` effectue une sauvegarde régulière des tampons modifiés.

Par ailleurs, une application utilisateur dispose de trois primitives pour forcer la sauvegarde des tampons modifiés la concernant :

- `sync()` sauvegarde tous les tampons modifiés sur le disque ;
- `fsync()` permet à un processus de sauvegarder sur le disque tous les blocs appartenant à un fichier qu'il est ouvert ;

- `fdatasync()` réalise la même opération que `fsync()`, sans sauvegarder l'i-node du fichier.

14.7 Les opérations sur les fichiers

14.7.1 L'ouverture d'un fichier

L'ouverture d'un fichier se fait à l'aide de la primitive `open()` :

```
<<
SYNOPSIS
       #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);
       int creat(const char *pathname, mode_t mode);

DESCRIPTION
       Étant donné le chemin pathname d'un fichier, open() renvoie
       un descripteur de fichier (petit entier positif ou nul)
       qui pourra ensuite être utilisé dans d'autres appels
       système (read(2), write(2), lseek(2), fcntl(2), etc.).
       Le descripteur de fichier renvoyé par un appel réussi
       sera le plus petit descripteur de fichier non actuellement
       ouvert par le processus.

       Le nouveau descripteur de fichier est configuré pour
       rester ouvert après un appel à execve(2) (son attribut
       FD_CLOEXEC décrit dans fcntl(2) est initialement
       désactivé). La position dans le fichier est fixée au début
       du fichier (voir lseek(2)).

       Un appel à open() crée une nouvelle description de fichier
       ouvert, une entrée dans la table de fichiers ouverts
       du système. Cette entrée enregistre la position dans le
       fichier et les attributs d'état du fichier
       (modifiables par l'opération F_SETFL de fcntl()). Un
       descripteur de fichier est une référence à l'une de ces
       entrées ; cette référence n'est pas modifiée si pathname
       est ensuite supprimé ou modifié pour correspondre à un autre
       fichier. La nouvelle description de fichier ouvert n'est
       initialement partagée avec aucun autre processus, mais ce
       partage peut apparaître après un fork(2).

       Le paramètre flags est l'un des éléments O_RDONLY, O_WRONLY
       ou O_RDWR qui réclament respectivement l'ouverture du
       fichier en lecture seule, écriture seule, ou lecture-
       écriture.

       De plus, zéro ou plus d'attributs de création de fichier et
       d'attributs d'état de fichier peuvent être spécifiés dans
       flags avec un OU binaire. Les attributs de création de
       fichier sont O_CREAT, O_EXCL, O_NOCTTY et O_TRUNC. Les
       attributs d'état de fichier sont tous les autres attributs
       listés ci-dessous. La distinction entre ces deux groupes
```

est que les attributs d'état de fichier peuvent être lus et (dans certains cas) modifiés avec fcntl(2). La liste complète des attributs de création et d'état de fichier est la suivante :

O_APPEND :
Le fichier est ouvert en mode « ajout ». Initialement, et avant chaque write(), la tête de lecture/écriture est placée à la fin du fichier comme avec lseek(). Il y a un risque d'endommager le fichier lorsque **O_APPEND** est utilisé, sur un système de fichiers NFS, si plusieurs processus tentent d'ajouter des données simultanément au même fichier. Ceci est dû au fait que NFS ne supporte pas l'opération d'ajout de données dans un fichier, aussi le noyau du client est obligé de la simuler, avec un risque de concurrence des tâches.

O_ASYNC :
Déclencher un signal (SIGIO par défaut, mais peut être changé via fcntl(2)) lorsque la lecture ou l'écriture deviennent possibles sur ce descripteur. Ceci n'est possible que pour les terminaux, pseudo-terminaux, sockets et (depuis Linux 2.6) tubes et FIFO. Voir fcntl(2) pour plus de détails.

O_CREAT :
Créer le fichier s'il n'existe pas. Le possesseur (UID) du fichier est renseigné avec l'UID effectif du processus. Le groupe propriétaire (GID) du fichier est le GID effectif du processus ou le GID du répertoire parent (ceci dépend du système de fichiers, des options de montage, du mode du répertoire parent, etc.) Voir par exemple les options de montage bsdgroups et sysv groups du système de fichiers ext2, décrites dans la page mount(8)).

O_DIRECT :
Essayer de minimiser les effets du cache d'entrée-sortie sur ce fichier. Ceci dégradera en général les performances, mais est utilisé dans des situations spéciales, lorsque les applications ont leur propres caches. Les entrées-sorties dans le fichier se font directement depuis l'espace utilisateur, elles sont synchrones (à la fin de read(2) ou write(2), les données ont obligatoirement été transférées). Sous Linux 2.4, la taille des transferts, l'alignement du tampon et la position dans le fichier doivent être des multiples de la taille de blocs logiques du système de fichiers. Sous Linux 2.6, un alignement sur des multiples de 512 octets est suffisant.

Une interface à la sémantique similaire (mais obsolète) pour les périphériques de type bloc est

décrite à la page raw(8).

O_DIRECTORY :
Si **pathname** n'est pas un répertoire, l'ouverture échoue. Cet attribut est spécifique à Linux et fut ajouté dans la version 2.1.126 du noyau, pour éviter des problèmes de dysfonctionnement si opendir(3) est invoqué sur une FIFO ou un périphérique de bande. Cet attribut ne devrait jamais être utilisé ailleurs que dans l'implémentation de opendir.

O_EXCL :
En conjonction avec **O_CREAT**, déclenchera une erreur si le fichier existe, et open() échouera. On considère qu'un lien symbolique existe, quel que soit l'endroit où il pointe. **O_EXCL** ne fonctionne pas sur les systèmes de fichiers NFS. Les programmes qui ont besoin de cette fonctionnalité pour verrouiller des tâches risquent de rencontrer une concurrence critique (race condition). La solution consiste à créer un fichier unique sur le même système de fichiers (par exemple avec le PID et le nom de l'hôte), utiliser link(2) pour créer un lien sur un fichier de verrouillage. Si link() renvoie 0, le verrouillage est réussi. Sinon, utiliser stat(2) sur ce fichier unique pour vérifier si le nombre de liens a augmenté jusqu'à 2, auquel cas le verrouillage est également réussi.

O_LARGEFILE :
(LFS) (Ndt : Large Files System) Autoriser l'ouverture des fichiers dont la taille ne peut pas être représentée avec un off_t (mais qui peut être représentée avec un off64_t).

O_NOATIME :
(Depuis Linux 2.6.8) Ne pas mettre à jour l'heure de dernier accès au fichier (champ st_atime de l'i-noeud) quand le fichier est lu avec read(2). Ce attribut est seulement conçu pour les programmes d'indexation et d'archivage, pour lesquels il peut réduire significativement l'activité du disque. L'attribut peut ne pas être effectif sur tous les systèmes de fichiers. Par exemple, avec NFS, l'heure d'accès est mise à jour par le serveur.

O_NOCTTY :
Si **pathname** correspond à un périphérique de terminal - voir tty(4) -, il ne deviendra pas le terminal contrôlant le processus même si celui-ci n'est attaché à aucun autre terminal.

O_NOFOLLOW :
Si **pathname** est un lien symbolique, l'ouverture échoue. Ceci est une extension FreeBSD, qui fut ajoutée à Linux dans la version 2.1.126. Les

liens symboliques se trouvant dans le chemin d'accès proprement dit seront suivis normalement.

O_NONBLOCK ou **O_NDELAY :**
Le fichier est ouvert en mode « non-bloquant ». Ni la fonction open() ni aucune autre opération ultérieure sur ce fichier ne laissera le processus appelant en attente. Pour la manipulation des FIFO (tubes nommés), voir également fifo(7). Pour une explication de l'effet de **O_NONBLOCK** en conjonction avec les verrouillages impératifs et les bauxs de fichiers, voir fcntl(2).

O_SYNC :
Le fichier est ouvert en écriture synchronisée. Chaque appel à write() sur le fichier bloquera le processus appelant jusqu'à ce que les données aient été écrites physiquement sur le support matériel (voir la section RESTRICTIONS plus bas).

O_TRUNC :
Si le fichier existe, est un fichier régulier, et est ouvert en écriture (**O_RDWR** ou **O_WRONLY**), il sera tronqué à une longueur nulle. Si le fichier est une FIFO ou un périphérique terminal, l'attribut **O_TRUNC** est ignoré. Sinon, le comportement de **O_TRUNC** n'est pas précisé. Sur de nombreuses versions de Linux, il sera ignoré ; sur d'autres versions il déclenchera une erreur).

Certains de ces attributs optionnels peuvent être modifiés par la suite avec la fonction fcntl().

L'argument **mode** indique les permissions à utiliser si un nouveau fichier est créé. Cette valeur est modifiée par le umask du processus : la véritable valeur utilisée est (**mode** & ~umask). Notez que ce mode ne s'applique qu'aux accès ultérieurs au fichier nouvellement créé. L'appel open() qui crée un fichier dont le mode est en lecture seule fournira quand même un descripteur de fichier en lecture et écriture.

Les constantes symboliques suivantes sont disponibles pour **mode** :

S_IRWXU : 00700 L'utilisateur (propriétaire du fichier) a les autorisations de lecture, écriture, exécution.
S_IRUSR : 00400 L'utilisateur a l'autorisation de lecture.
S_IWUSR : 00200 L'utilisateur a l'autorisation d'écriture.
S_IXUSR : 00100 L'utilisateur a l'autorisation d'exécution.
S_IRWXG : 00070 Le groupe a les autorisations de lecture, écriture, exécution.
S_IRGRP : 00040 Le groupe a l'autorisation de lecture.
S_IWGRP : 00020 Le groupe a l'autorisation d'écriture.
S_IXGRP : 00010 Le groupe a l'autorisation d'exécution.
S_IRWXO : 00007 Tout le monde a les autorisations de lecture, écriture, exécution.

S_IROTH : 00004 Tout le monde a l'autorisation de lecture.
S_IWOTH : 00002 Tout le monde a l'autorisation d'écriture.
S_IXOTH : 00001 Tout le monde a l'autorisation d'exécution.

Le mode devrait toujours être indiqué quand **O_CREAT** est dans les attributs **flags**, (il est ignoré dans les autres cas).

creat() est équivalent à open() avec l'attribut **flags** égal à **O_CREAT** | **O_WRONLY** | **O_TRUNC**.

VALEUR RENVOYÉE

open() et creat() renvoient le nouveau descripteur de fichier s'ils réussissent, ou -1 s'ils échouent, auquel cas **errno** contient le code d'erreur.

NOTES

Notez que open() peut ouvrir des fichiers spéciaux mais creat() ne peut pas en créer, il faut utiliser mknod(2) à la place.

Sur les systèmes de fichiers NFS, où la correspondance d'UID est activée, open() peut renvoyer un descripteur de fichier alors qu'une requête read(2) par exemple sera refusée avec le code d'erreur EACCES.
En effet, c'est parce que le client a effectué open() en vérifiant les autorisations d'accès, mais la correspondance d'UID est calculée par le serveur au moment des requêtes de lecture ou d'écriture.

Si un fichier est créé, ses horodatages st_atime, st_ctime, st_mtime (respectivement heure de dernier accès, de dernière modification d'état, et de dernière modification ; voir stat(2)) sont fixés à l'heure actuelle, ainsi que les champs st_ctime et st_mtime du répertoire parent. Sinon, si le fichier est modifié à cause de l'attribut **O_TRUNC**, ses champs st_ctime et st_mtime sont remplis avec l'heure actuelle.

ERREURS

EACCES : L'accès demandé au fichier est interdit, ou la permission de parcours pour l'un des répertoires du chemin **pathname** est refusée, ou le fichier n'existe pas encore et le répertoire parent ne permet pas l'écriture. (Voir aussi path_resolution(2).)
EEXIST : **pathname** existe déjà et **O_CREAT** et **O_EXCL** ont été indiqués.
EFAULT : **pathname** pointe en dehors de l'espace d'adressage accessible
EISDIR : On a demandé une écriture alors que **pathname** correspond à un répertoire (en fait, **O_WRONLY** ou **O_RDWR** ont été demandés).
ELOOP : **pathname** contient une référence circulaire (à travers un lien symbolique), ou l'attribut **O_NOFOLLOW** est indiqué et **pathname** est un lien symbolique.

EMFILE : Le processus a déjà ouvert le nombre maximal de fichiers.
ENAMETOOLONG : **pathname** est trop long.
ENFILE : La limite du nombre total de fichiers ouverts sur le système a été atteinte.
ENODEV : **pathname** correspond à un fichier spécial et il n'y a pas de périphérique correspondant. (Ceci est un bogue du noyau Linux ; dans cette situation, ENXIO devrait être renvoyé.)
ENOENT : **O_CREAT** est absent et le fichier n'existe pas. Ou un répertoire du chemin d'accès **pathname** n'existe pas, ou est un lien symbolique pointant nulle part.
ENOMEM : Pas assez de mémoire pour le noyau.
ENOSPC : **pathname** devrait être créé mais le périphérique concerné n'a plus assez de place pour un nouveau fichier.
ENOTDIR: Un élément du chemin d'accès **pathname** n'est pas un répertoire, ou l'attribut **O_DIRECTORY** est utilisé et **pathname** n'est pas un répertoire.
ENXIO : **O_NONBLOCK** | **O_WRONLY** est indiqué, le fichier est une FIFO et le processus n'a pas de fichier ouvert en lecture. Ou le fichier est un noeud spécial et il n'y a pas de périphérique correspondant.
EOVERFLOW : **pathname** fait référence à un fichier régulier, trop grand pour pouvoir être ouvert – voir **O_LARGEFILE** plus haut.
EPERM : L'attribut **O_NOATIME** est indiqué, mais l'UID effectif de l'appelant n'est pas le propriétaire du fichier, et l'appelant n'est pas privilégié (CAP_FOWNER).
EROFS : Un accès en écriture est demandé alors que **pathname** réside sur un système de fichiers en lecture seule.
ETXTBSY : On a demandé une écriture alors que **pathname** correspond à un fichier exécutable actuellement utilisé.
EWOULDBLOCK : L'attribut **O_NONBLOCK** est indiqué, et un bail incompatible est détenu sur le fichier (voir fcntl(2)).

RESTRICTIONS
Plusieurs problèmes se posent avec le protocole NFS, concernant entre autres **O_SYNC**, et **O_NDELAY**.
POSIX fournit trois variantes différentes des entrées-sorties synchronisées correspondant aux attributs **O_SYNC**, **O_DSYNC** et **O_RSYNC**. Actuellement (2.1.130) elles sont toutes équivalentes sous Linux.

>>

14.7.2 La fermeture d'un fichier

La fermeture d'un fichier se fait via la primitive `close()` :

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

```
       int close(int fd);

DESCRIPTION
       close() ferme le descripteur fd, de manière à ce qu'il ne
       référence plus aucun fichier, et puisse être réutilisé.

       Si fd est la dernière copie d'un descripteur de fichier
       donné, les ressources qui lui sont associées sont libérées.
       Par exemple tous les verrouillages sont supprimés et si le
       descripteur etait la dernière référence sur un fichier
       supprimé avec unlink(2), le fichier est effectivement
       effacé.

VALEUR RENVOYEE
       close renvoie 0 s'il réussit, ou -1 en cas d'échec, auquel
       cas errno contient le code d'erreur.

ERREURS
       EBADF : Le descripteur de fichier fd est invalide

>>
```

14.7.3 La lecture et l'écriture dans un fichier

Lecture et écriture dans un fichier se font à l'aide des primitives `read()` et `write()` :

```
<<
SYNOPSIS
       #include <unistd.h>
       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read()  lit  jusqu'à  count  octets depuis le descripteur de
       fichier fd dans le tampon pointé par buf.
       Si count vaut zéro, read() renvoie zéro et n'a pas d'autres
       effets.  Si count est supérieur à SSIZE_MAX, le résultat est
       indéfini.

VALEUR RENVOYÉE
       read()  renvoie  -1  s'il  échoue,  auquel  cas  errno
       contient le code d'erreur, et la position de la tête de
       lecture  est  indéfinie.  Sinon, read() renvoie le nombre
       d'octets lus (0 en fin de fichier), et avance la tête de
       lecture de ce nombre. Le fait que le nombre renvoyé  soit
       plus petit  que le nombre demandé n'est pas une erreur.
       Ceci se produit à la fin du fichier, ou si on lit depuis
       un tube ou un terminal,  ou  encore si read() a été
       interrompu par un signal.

ERREURS
       EAGAIN : On  utilise  une  lecture  non bloquante (attribut
                O_NONBLOCK du descripteur de fichier) et aucune
                donnée n'était disponible.
       EBADF  : fd n'est pas un descripteur  de  fichier  valide
                ou  n'est  pas ouvert en lecture.
       EFAULT : buf pointe en dehors de l'espace d'adressage
                accessible.
       EINTR  : read()  a été interrompu par un signal avant
                d'avoir eu le temps de lire quoi que ce soit.
```

EINVAL : Le descripteur **fd** correspond à un objet sur lequel il est impossible de lire. Ou bien le fichier a été ouvert avec le drapeau O_DIRECT, et l'adresse de **buf**, la valeur de count ou la position actuelle de la tête de lecture ne sont pas alignées correctement.

EIO : Erreur d'entrée-sortie. Ceci arrive si un processus est dans un groupe en arrière-plan et tente de lire depuis le terminal. Il reçoit un signal SIGTTIN mais il l'ignore ou le bloque. Ceci se produit également si une erreur d'entrée-sortie bas-niveau s'est produite pendant la lecture d'un disque ou d'une bande.

EISDIR : fd est un répertoire.

D'autres erreurs peuvent se produire, suivant le type d'objet associé à **fd**. POSIX permet à un read() interrompu par un signal de renvoyer soit le nombre d'octets lus à ce point, soit -1, et de placer errno à EINTR.

SYNOPSIS

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

DESCRIPTION

write() écrit jusqu'à **count** octets dans le fichier associé au descripteur **fd** depuis le tampon pointé par **buf**. La norme POSIX réclame qu'une lecture avec read() effectuée après le retour d'une écriture avec write(), renvoie les nouvelles données. Notez que tous les systèmes de fichiers ne sont pas compatibles avec POSIX.

VALEUR RENVOYÉE

write() renvoie le nombre d'octets écrits (0 signifiant aucune écriture), ou -1 s'il échoue, auquel cas **errno** contient le code d'erreur. Si **count** vaut zéro, et si le descripteur est associé à un fichier normal, 0 sera renvoyé sans effets de bord. Pour un fichier spécial, les résultats ne sont pas portables.

ERREURS

EAGAIN : L'écriture est non-bloquante (attribut O_NONBLOCK du descripteur), et l'opération devrait bloquer.

EBADF : **fd** n'est pas un descripteur de fichier valide, ou n'est pas ouvert en écriture.

EFAULT : **buf** pointe en dehors de l'espace d'adressage accessible.

EFBIG : Tentative d'écrire sur un fichier dont la taille dépasse un maximum dépendant de l'implémentation ou du processus, ou d'écrire à une position qui dépasse le maximum autorisé.

EINTR : L'appel système a été interrompu par un signal avant d'avoir pu écrire quoi que ce soit.

EINVAL : **fd** correspond à un objet sur lequel il est impossible d'écrire. Ou bien le fichier a été ouvert avec l'attribut O_DIRECT, et soit l'adresse

de buf, soit la valeur de count, soit la position actuelle dans le fichier ne sont pas alignées correctement.

EIO : Une erreur d'entrée-sortie bas niveau s'est produite durant la modification de l'inoeud.

ENOSPC : Le périphérique correspondant à **fd** n'a plus de place disponible.

EPIPE : **fd** est connecté à un tube (pipe) ou une socket dont l'autre extrémité est fermée. Quand ceci se produit, le processus écrivain reçoit un signal SIGPIPE. Ainsi la valeur de retour de write n'est vue que si le programme intercepte, bloque ou ignore ce signal.

D'autres erreurs peuvent se produire suivant le type d'objet associé à **fd**.

>>

14.7.4 Se positionner dans un fichier

La position courant du pointeur de fichier se fait à l'aide de la primitive `lseek()` :

<<
SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

DESCRIPTION

La fonction lseek() place la tête de lecture/écriture à la position **offset** dans le fichier associé au descripteur **fildes** en suivant la directive **whence** ainsi :

SEEK_SET
La tête est placée à **offset** octets depuis le début du fichier.

SEEK_CUR
La tête de lecture/écriture est avancée de **offset** octets.

SEEK_END
La tête est placée à la fin du fichier plus **offset** octets.

La fonction lseek() permet de placer la tête au-delà de la fin actuelle du fichier (mais cela ne modifie pas la taille du fichier). Si des données sont écrites à cet emplacement, une lecture ultérieure de l'espace intermédiaire (un « trou ») retournera des zéros (« \0 ») jusqu'à ce que d'autres données y soient écrites.

VALEUR RENVOYÉE
lseek(), s'il réussit, renvoie le nouvel emplacement, mesuré en octets depuis le début du fichier. En cas d'échec, la valeur (off_t)-1 est renvoyée, et **errno** contient le code d'erreur.

ERREURS

EBADF **fildes** n'est pas un descripteur de fichier ouvert.

EINVAL **whence** n'est ni **SEEK_SET**, ni **SEEK_CUR**, ni **SEEK_END**, ou bien la position demandée serait négative, ou après la fin d'un périphérique.

EOVERFLOW
La position résultante dans le fichier ne peut être représentée dans un off_t

ESPIPE **fildes** est associé à un tube (pipe), une socket, ou une file FIFO.

>>

14.7.5 Manipuler les attributs des fichiers

Les attributs d'un fichier peuvent être manipulés à l'aide des primitives `stat()` et `fstat()` :

<<
SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *path, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

DESCRIPTION

Ces fonctions renvoient des informations à propos du fichier indiqué. Vous n'avez besoin d'aucun droit d'accès au fichier pour obtenir les informations, mais vous devez — dans le cas de stat() et lstat() — avoir le droit de parcours de tous les répertoires mentionnés dans le chemin menant au fichier.

stat() récupère l'état du fichier pointé par **path** et remplit le tampon **buf**.

lstat() est identique à stat(), sauf que dans le cas où **path** est un lien symbolique, il donne l'état du lien lui-même plutôt que celui du fichier visé.

fstat() est identique à stat(), sauf que le fichier dont l'état est donné est celui référencé par le descripteur de fichier **filedes**.

Les trois fonctions retournent une structure stat contenant les champs suivants :

```
struct stat {
    dev_t     st_dev;      /* Périphérique            */
    ino_t     st_ino;      /* Numéro i-nœud           */
    mode_t    st_mode;     /* Protection              */
```

```
        nlink_t   st_nlink;    /* Nb liens matériels       */
        uid_t     st_uid;      /* UID propriétaire         */
        gid_t     st_gid;      /* GID propriétaire         */
        dev_t     st_rdev;     /* Type périphérique        */
        off_t     st_size;     /* Taille totale en octets  */
        blksize_t st_blksize;  /* Taille de bloc pour E/S  */
        blkcnt_t  st_blocks;   /* Nombre de blocs alloués  */
        time_t    st_atime;    /* Heure dernier accès      */
        time_t    st_mtime;    /* Heure dernière modif.    */
        time_t    st_ctime;    /* Heure dernier chmt état  */
    };
```

Le champ st_dev indique le périphérique sur lequel ce fichier se trouve.

Le champ st_rdev indique le périphérique que ce fichier (inœud) représente.

Le champ st_size indique la taille du fichier (s'il s'agit d'un fichier régulier ou d'un lien symbolique) en octets. La taille d'un lien symbolique est la longueur de la chaîne représentant le chemin d'accès qu'il vise, sans le caractère NUL final.

Le champ st_blocks indique le nombre de blocs de 512 octets alloués au fichier. Cette valeur peut être inférieure à st_size/512, par exemple si le fichier a des trous.

Le champ st_blksize donne la taille de bloc « préférée » pour des entrées-sorties efficaces. Des écritures par blocs plus petits peuvent entraîner un cycle lecture/modification/réécriture inefficace.

Tous les systèmes de fichiers de Linux n'implémentent pas tous les champs liés à la date. Certains systèmes de fichiers autorisent le montage de telle manière que les accès ne modifient pas le champ st_atime (voir l'option « noatime » de mount(8)).

Le champ st_atime est modifié par les accès au fichier, par exemple avec execve(2), mknod(2), pipe(2), utime(2) et read(2) (d'au moins un octet). D'autres routines, comme mmap(2), peuvent ou non mettre à jour ce champ st_atime.

Le champ st_mtime est modifié par des changements sur le fichier lui-même, c'est-à-dire mknod(2), truncate(2), utime(2) et write(2) (d'au moins un octet). D'autre part, le champ st_mtime d'un répertoire est modifié lors de la création ou la suppression de fichiers en son sein. Le champ st_mtime n'est pas mis à jour lors de modification de propriétaire, groupe, mode ou nombre de liens physiques.

Le champ st_ctime est modifié lors d'une écriture ou une modification de données concernant l'inœud (propriétaire, groupe, mode, etc.).

Les macros POSIX suivantes sont fournies pour vérifier le type de fichier (dans le champ st_mode) :

S_ISREG(m) un fichier ordinaire ?
S_ISDIR(m) un répertoire ?
S_ISCHR(m) un périphérique en mode caractère ?
S_ISBLK(m) un périphérique en mode bloc ?
S_ISFIFO(m) FIFO (tube nommé) ?
S_ISLNK(m) un lien symbolique ?
S_ISSOCK(m) une socket ?

Les attributs suivants correspondent au champ st_mode :

S_IFMT	0170000	masque du type de fichier
S_IFSOCK	0140000	socket
S_IFLNK	0120000	lien symbolique
S_IFREG	0100000	fichier ordinaire
S_IFBLK	0060000	périphérique blocs
S_IFDIR	0040000	répertoire
S_IFCHR	0020000	périphérique caractères
S_IFIFO	0010000	fifo
S_ISUID	0004000	bit Set-UID
S_ISGID	0002000	bit Set-Gid (voir ci-dessous)
S_ISVTX	0001000	bit « sticky » (voir ci-dessous)
S_IRWXU	00700	lecture/écriture/exécution du prop.
S_IRUSR	00400	le propriétaire a le droit de lecture
S_IWUSR	00200	le propriétaire a le droit d'écriture
S_IXUSR	00100	le propriétaire a le droit d'exécution
S_IRWXG	00070	lecture/écriture/exécution du groupe
S_IRGRP	00040	le groupe a le droit de lecture
S_IWGRP	00020	le groupe a le droit d'écriture
S_IXGRP	00010	le groupe a le droit d'exécution
S_IRWXO	00007	lecture/écriture/exécution des autres
S_IROTH	00004	les autres ont le droit de lecture
S_IWOTH	00002	les autres ont le droit d'écriture
S_IXOTH	00001	les autres ont le droit d'exécution

Le bit Set-GID (**S_ISGID**) a plusieurs utilisations particulières : pour un répertoire, il indique que la sémantique BSD doit être appliquée en son sein, c'est-à-dire que les fichiers qui y sont créés héritent leur GID du répertoire et non pas du GID effectif du processus créateur, et les sous-répertoires auront automatiquement le bit **S_ISGID** actif. Pour les fichiers qui n'ont pas d'autorisation d'exécution pour le groupe (**S_IXGRP** non actif), ce bit indique qu'un verrouillage strict est en vigueur sur ce fichier.

Le bit « sticky » (**S_ISVTX**) sur un répertoire indique que les fichiers qui s'y trouvent ne peuvent être renommés ou effacés que par leur propriétaire, par le propriétaire du répertoire ou par un processus privilégié.

VALEUR RENVOYÉE
En cas de réussite, zéro est renvoyé, sinon -1 est renvoyé et **errno** contient le code d'erreur.

ERREURS
EACCES La permission de parcours est refusée pour un des répertoires contenu dans le chemin **path**. (Voir aussi

```
              path_resolution(2).)

       EBADF  filedes est un mauvais descripteur.

       EFAULT Un pointeur se trouve en dehors de l'espace
              d'adressage.

       ELOOP  Trop de liens symboliques rencontrés dans le chemin
              d'accès.

       ENAMETOOLONG
              Nom de fichier trop long.

       ENOENT Un composant du chemin path n'existe pas,  ou  il
              s'agit  d'une chaîne vide.

       ENOMEM Pas assez de mémoire pour le noyau.

       ENOTDIR
              Un composant du chemin d'accès n'est pas un
              répertoire.
```

>>

14.7.6 Réaliser des opérations sur des fichiers

La fonction « `fcntl()` » permet de se livrer à diverses opérations sur un descripteur de fichier. Le synopsis de cette fonction est :

<<

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

>>

L'utilisation de cette fonction dépasse le programme du cours, mais notez que la commande « **cmd** » qui peut être appliquée au descripteur « **fd** » peut être de dupliquer un descripteur de fichier (Cf. fonction « `dup()` » déjà rencontrée), changer les attributs du descripteur de fichier, connaître l'attribut d'état d'un fichier, verrouiller un fichier, gérer les signaux (par exemple, pour rendre les appels à `read()` et `write()` bloquants/non bloquants), être notifié d'une modification de fichier et de répertoire, etc.

14.7.7 Création/suppression de liens

La création et la suppression de liens se fait à l'aide des primitives `link()` et `unlink()` :

<<

```
SYNOPSIS

       #include <unistd.h>
       int link(const char *oldpath, const char *newpath);

DESCRIPTION
       link()  crée  un nouveau lien (aussi appelé lien matériel ou
       hard link) sur un fichier existant.
```

```
       Si newpath existe, il ne sera pas écrasé.

       Ce nouveau nom pourra être utilisé exactement comme l'ancien
       quelle que soit  l'opération.  Les deux noms réfèrent au
       même fichier (et ont donc les mêmes permissions et
       propriétaire) et il est impossible  de  déterminer quel nom
       était l'original.

VALEUR RENVOYÉE
       L'appel renvoie 0 quand il réussit, ou -1 s'il échoue,
       auquel cas errno contient le code d'erreur.
-----------------------------------------------------------------
SYNOPSIS
       #include <unistd.h>
       int unlink(const char *pathname);

DESCRIPTION
       unlink() détruit un nom dans le système de fichiers. Si ce
       nom était le dernier lien sur un fichier,  et  si  aucun
       processus  n'a  ouvert  ce fichier,  ce  dernier est effacé,
       et l'espace qu'il utilisait est rendu disponible.

       Si le nom était le dernier lien sur un fichier,  mais  qu'un
       processus conserve  encore le fichier ouvert, celui continue
       d'exister jusqu'à ce que le dernier descripteur le
       référençant soit fermé.

       Si le nom correspond à un lien symbolique, le lien est
       effacé.

       Si le nom correspond à une socket, une FIFO, ou un
       périphérique, le nom est  effacé mais les processus qui ont
       ouvert l'objet peuvent continuer à l'utiliser.

VALEUR RENVOYÉE
       En cas de réussite, zéro est renvoyé, sinon -1  est  renvoyé
       et  errno contient le code d'erreur.

>>
```

14.7.8 Modification et tests des droits d'un fichier

Les droits d'accès sont positionnés à la création de ce dernier. Ils peuvent être modifiés par la suite à l'aide des primitives `chmod()` et `fchmod()` :

```
<<
SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       int chmod(const char *path, mode_t mode);
       int fchmod(int fildes, mode_t mode);

DESCRIPTION
       chmod  change  le  mode d'accès du fichier donné par path ou
       associé au descripteur fildes.

       Le mode est spécifié par un OU binaire « | » entre les
       éléments  suivants (les nombres sont en octal) :
```

S_ISUID 04000 modification du n° d'utilisateur (UID) à l'exécution.

S_ISGID 02000 modification du numéro de groupe (GID) à l'exécution.

S_ISVTX 01000 positionner le sticky bit pour conserver le code du programme en mémoire après exécution.

S_IRUSR 00400 accès en lecture pour le propriétaire

S_IWUSR 00200 accès en écriture pour le propriétaire

S_IXUSR 00100 accès en exécution/parcours par le propriétaire

S_IRGRP 00040 accès en lecture pour le groupe

S_IWGRP 00020 accès en écriture pour le groupe

S_IXGRP 00010 accès en exécution/parcours pour le groupe

S_IROTH 00004 accès en lecture pour les autres

S_IWOTH 00002 accès en écriture pour les autres

S_IXOTH 00001 accès en exécution/parcours pour les autres

L'UID effectif du processus appelant doit correspondre à celui du propriétaire du fichier, ou le processus doit être privilégié (sous Linux : il doit avoir la capacité CAP_FOWNER).

Si le processus appelant n'est pas privilégié (sous Linux : n'a pas la capacité CAP_FSETID), et si le groupe du fichier ne correspond ni au GID effectif du processus, ni à l'un de ses éventuels groupes supplémentaires, le bit S_ISGID sera désactivé, mais cela ne créera pas d'erreur.

Par mesure de sécurité, suivant le type de système de fichiers, les bits Set-UID et Set-GID peuvent être effacés si un fichier est écrit. (Sous Linux, cela arrive si le processus qui écrit n'a pas la capacité CAP_FSETID. Sur certains systèmes de fichiers, seul le superutilisateur peut positionner le Sticky-Bit, lequel peut avoir une signification spécifique. Pour la signification du Sticky-Bit et du bit Set-GID sur les répertoires, voir stat(2).

Sur les systèmes de fichiers NFS, une restriction des autorisations d'accès aura un effet immédiat y compris sur les fichiers déjà ouverts, car les contrôles d'accès sont effectués sur le serveur, mais les fichiers sont maintenus ouverts sur le client. Par contre, un

élargissement des autorisations peut ne pas être immédiat pour les autres clients, s'ils disposent d'un cache.

VALEUR RENVOYÉE
L'appel renvoie 0 quand il réussit, ou -1 s'il échoue, auquel cas **errno** contient le code d'erreur.

>>

Inversement, il est possible de connaître les droits d'accès à un fichier via la primitive access() :

<<
SYNOPSIS
#include <unistd.h>
int access(const char ***pathname**, int **mode**);

DESCRIPTION
access() vérifie si le processus serait autorisé à lire, écrire, exécuter, ou tester l'existence d'un fichier (ou d'un autre objet appartenant au système de fichiers), dont le nom est **pathname**. Si pathname est un lien symbolique, ce sont les permissions du fichier pointé par celui-ci qui sont testées.

mode est un masque constitué des arguments liés par un OU binaire « | » : **R_OK**, **W_OK**, **X_OK** et **F_OK**.

Les requêtes **R_OK**, **W_OK** et **X_OK** servent respectivement à tester la lecture, l'écriture, et l'exécution du fichier. La requête **F_OK** teste l'existence du fichier.

Les tests dépendent des permissions des répertoires apparaissant dans le chemin pathname et éventuellement des cibles des liens symboliques rencontrés sur ce chemin.

Le test est effectué avec les UID et GID réels du processus, plutôt qu'avec les IDs effectifs qui sont utilisés lorsque l'on tente l'opération. Ceci permet aux programmes Set-UID de déterminer les autorisations de l'utilisateur ayant invoqué le programme.

Seuls les bits d'accès sont vérifiés, et non pas le contenu du fichier. Ainsi, l'autorisation d'écriture dans un répertoire indique la possibilité d'y créer des fichiers et non d'y écrire comme dans un fichier. De même, un fichier DOS peut être considéré comme exécutable, alors que l'appel execve(2) échouera évidemment.

Si le processus a les privilèges suffisants, une implémentation peut indiquer un succès pour **X_OK** même si le fichier n'a aucun bit d'exécution positionné.

VALEUR RENVOYÉE
L'appel renvoie 0 s'il réussit (toutes les requêtes sont autorisées), ou -1 s'il échoue (au moins une requête du mode est interdite), auquel cas **errno** contient le code d'erreur.

>>

14.7.9 Modification du propriétaire d'un fichier

Le propriétaire et le groupe propriétaire sont fixés à la création du fichier. Ils peuvent être modifiés par la suite, en utilisant les primitives `chown()` et `fchown()` :

```
<<
SYNOPSIS

       #include <sys/types.h>
       #include <unistd.h>
       int chown(const char *path, uid_t owner, gid_t group);
       int fchown(int fd, uid_t owner, gid_t group);
       int lchown(const char *path, uid_t owner, gid_t group);

DESCRIPTION
       Ces  appels  système  modifient le propriétaire et le groupe
       du fichier désigné par le chemin path ou par le descripteur
       de fichier fd. Seul un processus  privilégié  (sous  Linux :
       un  processus  qui a la capacité CAP_CHOWN) peut modifier le
       propriétaire d'un fichier. Le  propriétaire peut modifier le
       groupe du fichier pour n'importe quel groupe auquel il
       appartient. Un processus privilégié  (sous  Linux :  avec
       la  capacité CAP_CHOWN) peut modifier le groupe
       arbitrairement.

       Si l'argument owner ou group vaut -1, l'élément
       correspondant n'est pas changé.

       Quand le propriétaire, ou le groupe d'un fichier exécutable
       sont  modifiés  par  un  utilisateur ordinaire, les bits
       S_ISUID et S_ISGID sont effacés. POSIX ne précise pas s'il
       faut agir de même lorsque  c'est  le superutilisateur  qui
       invoque chown(). Le comportement de Linux dans ce cas dépend
       de la version du noyau. Si le fichier n'est  pas  exécutable
       par  les  membres  de son groupe, (son bit S_IXGRP étant à
       zéro) le bit S_ISGID indique la présence d'un verrou
       obligatoire sur le fichier,  et n'est donc pas effacé par un
       chown().

VALEUR RENVOYÉE
       L'appel renvoie 0 quand il réussit, ou -1 s'il échoue,
       auquel cas errno contient le code d'erreur.

>>
```

14.8 Les opérations sur les répertoires

14.8.1 Changer de répertoire courant

Le répertoire courant d'un processus est celui à partir duquel il a été lancé (s'il est lancé depuis un shell), ou il est hérité du père après un fork(). Ce répertoire courant peut être changé à l'aide des primitives `chroot()` et `fchroot()` :

```
<<
SYNOPSIS

       #include <unistd.h>
```

```
       int chdir(const char *path);
       int fchdir(int fd);

DESCRIPTION
       chdir()  remplace  le  répertoire  de travail courant par
       celui indiqué dans le chemin path.

       fchdir() est identique à chdir(), sauf  que  le  répertoire
       cible  est fourni sous forme de descripteur de fichier.

VALEUR RENVOYÉE
       L'appel renvoie 0 quand il réussit, ou -1 s'il échoue,
       auquel cas errno contient le code d'erreur.

>>
```

14.8.2 Changer de répertoire racine

Pour un processus, la racine « / » est héritée de son père, et il s'agit en général de celle qui est montée par le noyau au démarrage. Néanmoins, un processus peut voir sa racine changer de place via la primitive `chroot()` :

```
<<
SYNOPSIS

       #include <unistd.h>
       int chroot (const char *path);

DESCRIPTION
       chroot()  remplace le répertoire racine du processus en
       cours par celui spécifié par le chemin path. Ce répertoire
       sera utilisé  comme  origine des  chemins  commençant par
       « / ». Le répertoire racine est hérité par tous les enfants
       du processus ayant fait le changement.

       Seul un processus  privilégié  (sous  Linux :  un  processus
       ayant  la capacité CAP_SYS_CHROOT) peut appeler chroot(2).

       Cette  fonction change un des ingrédients de l'algorithme de
       Résolution des chemins, et ne modifie rien d'autre.

       Notez que cet appel système ne modifie pas le  répertoire
       de  travail, aussi  « . »  peut se retrouver en dehors de
       l'arbre dont la racine est « / ». En particulier, le
       superutilisateur peut s'évader  d'un  « piège chroot » en
       faisant « mkdir foo; chroot foo; cd .. ».

       Cet  appel  ne  ferme aucun descripteur de fichier, et de
       tels descripteurs peuvent permettre un accès à des fichiers
       hors de l'arbre dont la racine est le nouveau « / ».

VALEUR RENVOYÉE
       L'appel renvoie 0 quand il réussit, ou -1 s'il échoue,
       auquel cas errno contient le code d'erreur.

>>
```

14.8.3 Création d'un répertoire

La création d'un répertoire se fait via la primitive `mkdir()` :

```
<<
SYNOPSIS

       #include <sys/stat.h>
       #include <sys/types.h>
       int mkdir(const char *pathname, mode_t mode);

DESCRIPTION
       mkdir() crée un nouveau répertoire nommé pathname.
       Le  paramètre  mode spécifie les permissions à appliquer au
       répertoire. Cette valeur peut être modifiée par le umask du
       processus : les permissions du répertoire effectivement créé
       vaudront (mode & ~umask & 0777).

       Le  répertoire  nouvellement créé aura pour propriétaire
       l'UID effectif du processus. Si le répertoire au-dessus du
       nouveau  répertoire  a  son bit Set-GID à 1, ou si le
       système de fichiers est monté avec une sémantique de groupe
       BSD, le nouveau répertoire héritera  de  l'appartenance
       au  groupe de son parent. Sinon il appartiendra au groupe
       correspondant au GID effectif du processus.

       Si le répertoire parent a son bit Set-GID à 1,  le  nouveau
       répertoire aura aussi son bit Set-GID à 1.

VALEUR RENVOYÉE
       mkdir()  renvoie  0  s'il  réussit, ou -1 s'il échoue,
       auquel cas errno contient le code d'erreur.

>>
```

14.8.4 Destruction d'un répertoire

Un répertoire peut être détruit à l'aide de la primitive `rmdir()` :

```
<<
SYNOPSIS

       #include <unistd.h>
       int rmdir(const char *pathname);

DESCRIPTION
       rmdir() supprime un répertoire, lequel doit être vide.

VALEUR RENVOYÉE
       En  cas  de  réussite,  zéro est renvoyé, sinon -1 est
       renvoyé et errno contient le code d'erreur.

>>
```

14.8.5 Exploration d'un répertoire

Trois primitives permettent de lister le contenu d'un répertoire : opendir(), readdir(), et closedir() :

<<
SYNOPSIS
```
       #include <sys/types.h>
       #include <dirent.h>
       DIR *opendir(const char *nom);
```

DESCRIPTION

La fonction opendir() ouvre un flux répertoire correspondant au répertoire **nom**, et renvoie un pointeur sur ce flux. Le flux est positionné sur la première entrée du répertoire.

VALEUR RENVOYÉE

La fonction opendir() renvoie un pointeur sur le flux répertoire. Si une erreur se produit, NULL est renvoyé, et **errno** contient le code d'erreur.

ERREURS

EACCES Accès interdit.
EMFILE Trop de descripteurs de fichiers pour le processus en cours.
ENFILE Trop de fichiers ouverts simultanément sur le système.
ENOENT Le répertoire n'existe pas, ou **nom** est une chaîne vide.
ENOMEM Pas assez de mémoire pour terminer l'opération.
ENOTDIR **nom** n'est pas un répertoire

SYNOPSIS
```
       #include <sys/types.h>
       #include <dirent.h>
       struct dirent *readdir(DIR *dir);
```

DESCRIPTION

La fonction readdir() renvoie un pointeur sur une structure dirent représentant l'entrée suivante du flux répertoire pointé par **dir**. Elle renvoie NULL a la fin du répertoire, ou en cas d'erreur.

Avec Linux, la structure dirent est définie comme suit :

```
   struct dirent {
       ino_t          d_ino;       /* numéro de l'inode */
       off_t          d_off;       /* décalage vers le
                                      prochain dirent */
       unsigned short d_reclen;    /* longueur de cet
                                      enregistrement */
       unsigned char  d_type;      /* type du fichier */
       char           d_name[256]; /* nom du fichier */
   };
```

Les données renvoyées par readdir() sont écrasées lors de l'appel suivant à readdir() sur le même flux répertoire.

VALEUR RENVOYÉE

La fonction readdir() renvoie un pointeur sur une structure dirent, ou NULL lorsqu'une erreur se produit, ou lorsque la fin du répertoire est atteinte. En cas d'erreur, **errno** contient le code d'erreur.

```
ERREURS
       EBADF  Le flux répertoire dir est invalide.
 ----------------------------------------------------------------
SYNOPSIS
       #include <sys/types.h>
       #include <dirent.h>
       int closedir(DIR *dir);

DESCRIPTION
       La fonction closedir() ferme le flux du répertoire associé à
       dir. Après cet appel, le descripteur dir du flux du
       répertoire n'est plus disponible.

VALEUR RENVOYÉE
       En cas de succès, la fonction closedir() renvoie 0. En cas
       d'erreur, -1 est renvoyé et errno est définie en
       conséquence.

ERREURS
       EBADF  Le descripteur de flux du répertoire dir est
              invalide.

>>
```

14.9 Les opérations diverses

14.9.1 Les opération sur les liens symboliques

Les primitives `symlink()` et `readlink()` permettent respectivement de créer un lien symbolique, et de lire le contenu du fichier sur lequel il pointe :

```
<<
SYNOPSIS
       #include <unistd.h>
       int symlink(const char *oldpath, const char *newpath);

DESCRIPTION
       symlink()  crée  un lien symbolique avec le nom newpath
       indiqué, et qui pointe sur oldpath.

       Les liens sont interprétés à l'exécution, comme si le
       contenu du lien était remplacé par le chemin d'accès
       pour trouver un fichier ou un répertoire.

       Les liens symboliques peuvent contenir .. pour le chemin,
       qui (s'il est utilisé au début du lien) se réfère aux
       répertoires parents du lien.

       Un  lien  symbolique  (aussi  nommé « soft link ») peut
       pointer vers un fichier existant ou sur un fichier non-
       existant.

       Les permissions d'accès à un lien symbolique sont sans
       importance,  le propriétaire est ignoré lorsque l'on suit le
       lien, il n'est vérifié que pour supprimer ou renommer le
       lien si celui-ci se trouve dans un répertoire avec le sticky
```

```
       bit (S_ISVTX) positionné.

       Si newpath existe il ne sera pas écrasé.

VALEUR RENVOYÉE
       En  cas  de  réussite,  zéro est renvoyé, sinon -1 est
       renvoyé et errno contient le code d'erreur.
 ----------------------------------------------------------------
SYNOPSIS
       #include <unistd.h>
       ssize_t readlink(const char *path, char *buf, size_t
                        bufsiz);

DESCRIPTION
       readlink() place le contenu du lien symbolique path dans le
       tampon buf, dont la taille est bufsiz. readlink() n'ajoute
       pas  de  caractère  NUL dans  le  tampon buf. Il tronquera
       le contenu (à la longueur bufsiz) si le tampon est trop
       petit pour recevoir les données.

VALEUR RENVOYÉE
       readlink() renvoie le nombre de caractères écrits dans le
       tampon, ou -1 s'il échoue, auquel cas errno contient le code
       d'erreur.

>>
```

14.9.2 Les opérations sur les partitions

Les primitives mount() et umount() permettent de monter et de démonter une partition :

```
<<
SYNOPSIS

       #include <sys/mount.h>
       int mount(const char *source, const char *target, \
                 const char *filesystemtype, \
                 unsigned long mountflags, const void *data);
       int umount(const char *target);
       int umount2(const char *target, int flags);

DESCRIPTION
       mount()  attache  le  système  de fichiers spécifié par
       source (qui est généralement un nom de périphérique, mais
       peut aussi être un répertoire ou un objet fictif) au
       répertoire indiqué par target.

       umount()  et  umount2() suppriment l'attachement du système
       de fichiers (le plus récemment) monté sur le répertoire
       target.

       Des privilèges appropriés (sous Linux : la capacité
       CAP_SYS_ADMIN) sont nécessaires pour monter ou démonter des
       systèmes de fichiers.

       Depuis  Linux  2.4  un  même  système  de fichiers peut être
       visible en différents points, et plusieurs montages peuvent
       être empilés  au  même point.
```

L'argument **filesystemtype** prend une des valeurs listées dans **/proc/filesystems** (par exemple « ext2 », « minix », « msdos », « proc », « nfs », « iso9660 » etc). Des types supplémentaires peuvent être disponibles lorsque les modules appropriés sont chargés.

L'argument **mountflags** peut avoir le nombre magique 0xC0ED (**MS_MGC_VAL**) dans ses 16 bits de poids forts (ceci était obligatoire dans les versions antérieures à la 2.4, mais ne l'est plus), et certains attributs de montage (définis dans <linux/fs.h> pour libc4 ou libc5 et dans <sys/mount.h> pour glibc2) comme bits de poids faibles :

MS_BIND
(Depuis Linux 2.4) Effectuer un montage lié, rendant un fichier ou une arborescence visibles en un autre point du système de fichiers. Les montages liés peuvent traverser les frontières entre systèmes de fichiers et ouvrir une porte dans une prison chroot(2). Les arguments **filesystemtype**, **mountflags**, et **data** sont ignorés.

MS_DIRSYNC (depuis Linux 2.5.19)
Rendre synchrones les modifications sur les répertoires du système de fichiers. (Cette propriété peut être obtenue pour les répertoires individuels ou les sous-arborescences en utilisant chattr(8).)

MS_MANDLOCK
Autoriser les verrouillages impératifs sur le système de fichiers. (Le verrouillage impératif devra toutefois être validé fichier par fichier, comme décrit dans fcntl(2).)

MS_MOVE
Déplacer une sous-arborescence. **source** indique un point de montage existant et **target** indique son nouvel emplacement. Le déplacement est atomique, à aucun moment la sous-arborescence n'est démontée. Les arguments **filesystemtype**, **mountflags**, et **data** sont ignorés.

MS_NOATIME
Ne pas mettre à jour les dates d'accès pour (tous) les fichiers du système de fichiers.

MS_NODEV
Ne pas autoriser la présence de fichiers spéciaux de périphérique sur le système de fichiers.

MS_NODIRATIME
Ne pas mettre à jour les dates d'accès pour les répertoires du système de fichiers.

MS_NOEXEC
Ne pas permettre l'exécution de programme depuis le système de fichiers.

MS_NOSUID
Ne pas tenir compte des bits Set-UID et Set-GID lors de l'exécution de programmes sur le système de fichiers.

MS_RDONLY
Monter le système de fichiers en lecture seule.

MS_REMOUNT
Reinitialiser un montage existant. Ceci permet de modifier les attributs **mountflags** et **data** d'un montage existant sans avoir besoin de démonter et remonter le système de fichiers. **source** et **target** doivent avoir les mêmes valeurs que durant l'appel mount() initial ; **filesystemtype** est ignoré.

Les attributs **mountflags** suivants peuvent être modifiés : **MS_RDONLY**, **MS_SYNCHRONOUS**, **MS_MANDLOCK** ; avant le noyau 2.6.16, **MS_NOATIME** et **MS_NODIRATIME** pouvaient également être modifiés. Enfin, avant le noyau 2.4, les attributs **MS_NOSUID**, **MS_NODEV** et **MS_NOEXEC** pouvaient être modifiés.

MS_SYNCHRONOUS
Rendre synchrones les écritures sur le système de fichiers (comme si l'option **O_SYNC** de open(2) était indiquée à chaque appel sur ce système de fichiers).

Depuis Linux 2.4, les attributs **MS_NODEV**, **MS_NOEXEC**, et **MS_NOSUID** sont configurables de manière variable sur chaque point de montage. À partir du noyau 2.6.16, **MS_NOATIME** et **MS_NODIRATIME** peuvent aussi être configurés pour chaque point de montage.

L'argument **data** est interprété différemment suivant le type de système de fichiers. Typiquement, c'est une chaîne d'options comprises par le système de fichiers, séparées par des virgules. Voir mount(8) pour des détails sur les options disponibles pour chaque type de système.

Depuis Linux 2.1.116 l'appel système umount2() fonctionne comme umount(), mais dispose d'options supplémentaires (**flags**) configurant le comportement de l'opération :

MNT_FORCE (depuis Linux 2.1.116)
Forcer le démontage, même si le système de fichiers est occupé (Seulement pour les montages NFS).

MNT_DETACH (depuis Linux 2.4.11)
Faire un détachement paresseux : rendre le point de montage invalide pour les nouveaux accès, et réaliser le démontage complet lorsque le point ne sera plus occupé.

MNT_EXPIRE (depuis Linux 2.6.8)
Marquer le point de montage comme ayant expiré. Si un

point de montage n'est pas utilisé, un premier appel à umount2() avec ce paramètre échoue avec l'erreur **EAGAIN**, mais marque le point de montage comme expiré. Il reste dans cet état tant qu'aucun processus n'y accède. Un second appel à umount2() avec **MNT_EXPIRE** détache le point de montage expiré. Ce paramètre ne peut être combiné avec **MNT_FORCE** ou **MNT_DETACH**.

VALEUR RENVOYÉE

L'appel renvoie 0 quand il réussit, ou -1 s'il échoue, auquel cas **errno** contient le code d'erreur.

ERREURS

Les erreurs détaillées ici sont indépendantes du type de système de fichiers. Chaque type de système peut avoir des codes d'erreurs spécifiques, et un comportement particulier. Voir les sources du noyau pour plus de détails.

EACCES Un élément du chemin d'accès n'est pas consultable (voir aussi path_resolution(2)), ou on tente de monter un système de fichiers en lecture seule sans préciser l'attribut **MS_RDONLY**, ou bien le périphérique de bloc source est situé sur un système de fichiers monté avec l'attribut **MS_NODEV**.

EAGAIN Un appel à umount2() avec l'attribut **MNT_EXPIRE** a marqué correctement un système de fichiers non utilisé comme expiré.

EBUSY **source** est déjà monté, ou ne peut pas être remonté en lecture seule car il y a des fichiers ouverts en écriture, ou ne peut pas être monté sur **target** car **target** est occupé (c'est le répertoire de travail d'un processus, le point de montage d'un autre périphérique, des fichiers y sont ouverts, etc.). Ou le démontage est impossible car le point est occupé.

EFAULT L'un des arguments pointe en dehors de l'espace d'adressage accessible.

EINVAL **source** a un superbloc invalide, ou on tente un remontage (**MS_REMOUNT**) alors que source n'était pas encore monté sur **target**. Ou un déplacement (**MS_MOVE**) est demandé alors que **source** n'est pas un point de montage ou est « / ». Ou un démontage est demandé sur **target** qui n'est pas un point de montage. Ou bien umount2() a été appelé avec l'option **MNT_EXPIRE**, en même temps que soit **MNT_DETACH** soit **MNT_FORCE**.

ELOOP Trop de liens symboliques rencontrés dans un chemin, ou un déplacement a été tenté dans lequel target est un descendant de source.

EMFILE (Dans le cas où un périphérique de bloc n'est pas nécessaire :) Table de montage factice pleine.

```
       ENAMETOOLONG
              Un des arguments est plus long que MAXPATHLEN.

       ENODEV filesystemtype n'est pas configuré dans le noyau.

       ENOENT Un des chemins est vide ou a un composant inexistant.

       ENOMEM Le noyau n'a pas pu allouer suffisamment de mémoire.

       ENOTBLK
              source n'est pas un fichier spécial en mode bloc.

       ENOTDIR
              Le second argument, ou un préfixe du premier
              argument, n'est pas un répertoire.

       ENXIO  Le nombre majeur du périphérique source est invalide.

       EPERM  L'appelant n'a pas les privilèges appropriés.
>>
```

14.10 Le système de fichier `/proc`

Le système de gestion de fichiers `/proc`, supporté par le VFS, est un système de gestion de fichiers particulier qui ne contient pas de données stockées sur le disque. Son rôle est de fournir a l'utilisateur un ensemble d'informations sur le noyau et les processus, à travers une interface de fichiers virtuels accessibles par l'interface du VFS.

On y trouve un ensemble de fichiers qui sont essentiellement de trois types:

- des fichiers d'informations sur le noyau et la machine tels que `cmdline` (arguments passés au noyau lors du démarrage), `cpuinfo` (description du ou des processeurs de la machine), `devices` (liste des gestionnaires de périphériques), `filesystems` (liste des systèmes de gestion de fichiers supportés par le noyau), `interrupts` (liste des interruptions matérielles utilisées par les gestionnaires de périphériques), `meminfo` (état de la mémoire centrale), `modules` (liste des modules chargés dans le noyau), etc. ;
- des répertoires tels que `net` (fichiers concernant les protocoles réseau), `scsi` (fichiers concernant les gestionnaires de périphériques scsi), etc. ;
- un répertoire par processus actifs sur la machine, dont le nom est égal au PID du processus. Ce répertoire comprend un ensemble de fichiers tels que `cmdline` (liste des arguments du processus), `cwd` (lien sur le répertoire courant du processus), `environ` (les variables d'environnements du processus), `exe` (lien sur le fichier exécutable du processus), `fd` (répertoire contenant des liens sur les fichiers ouverts par le processus), `maps` (liste des zones mémoire contenues dans l'espace d'adressage du processus), `mem` (contenu de l'espace d'adressage du processus), `root` (lien sur le répertoire racine du processus), `stat`, `statm` et `status` (état du processus).

15 Les entrées-sorties

Les processus ne cessent de communiquer avec les différents périphériques (clavier, écran, souris, disques durs, lecteurs de CD-Rom, clés USB, scanners, réseau, etc. Nous allons voir comment, sous Linux, cette communication se fait à travers des « fichiers spéciaux » (déjà rencontrés rapidement dans le précédent chapitre).

15.1 Principes

Les caractéristiques techniques de tous les périphériques d'entrées sont très différentes. Et même pour un périphérique donné (les disques durs par exemple), les caractéristiques peuvent être notamment différentes (dans notre exemple, les disques SCSI ont des propriétés temps réel que n'ont pas officiellement les disques IDE). C'est pourquoi a communication processus/périphérique se fait à travers deux couches :

- le ***pilote de périphérique*** (ou ***device driver*** en Anglais), qui propose au programmeur (ou au noyau) une API normalisée,
- et le ***contrôleur d'entrées-sorties*** ou ***dispositif d'unité d'échange***, formé de l'électronique qui gère le périphérique (contrôleur IDE ou SCSI par exemple).

15.1.1 Le contrôleur d'entrées-sorties

Le contrôleur d'entrées-sorties permet au CPU de dialoguer avec le périphérique, et ce à travers le bus. Ainsi, le contrôleur traduit les ordres du CPU (données au travers du bus de communication) en signaux électroniques utiles au périphérique. Inversement, par les interruptions matérielles, le périphérique peut demander d'engager un dialogue sur le bus.

La communication unité d'échange/CPU se fait à travers des « ***registres adressables*** ». Si leur nombre diffère selon les équipements, on retrouve certains registres de façon constante :

- le registre d'état, qui permet de connaître l'état du périphérique (l'état est codé sous forme numérique, pouvant signifier par exemple : en attente, en cours de lecture, en cours d'écriture, résultat disponible, plus de papier, initialisation, etc.) ;
- le registre de commande : permet au CPU d'envoyer des ordres (opérations d'entrées-sorties) ;
- un registre de données : c'est au travers de ce registre que se font les échanges mémoire/unité d'échange.

15.1.2 Le pilote

Le pilote offre aux programmes un certains nombre de fonctions standards acceptant des paramètres normalisés (API), parmi lesquelles :

- `open` : ouverture du périphérique (initialisation au niveau électronique et des structures) ;
- `read`, `write` : fonctions permettant de lire et d'écrire sur le périphérique ;
- `schedule` : permet d'ordonnancer une suite de plusieurs lectures/écritures, de façon à optimiser la durée des différentes opérations ;
- `interrupt` : fonction exécutée lors de la levée d'une interruption par le périphérique ;
- `close` : ferme le périphérique.

En pratique, le pilote dialogue avec le périphérique selon 3 modes :

- le polling (ou attente active) ;

- le mode avec interruption ;
- le mode avec DMA (*Direct Memory access*).

Le mode avec scrutation (ou attente active) consiste à intéroger régulièrement le registre d'état du périphérique afin de voir si des données ont été délivrées (ou prêtes à recevoir). Ce mécanisme est inutilement consommateur de CPU si le périphérique est inactif.

Le mode de fonctionnement « *avec interruption* » utilise le mécanisme d'interruption matérielle. Une fonction `interrupt()` est associée au périphérique. Dès qu'un événement concernant le périphérique survient, une interruption est levée, et cette fonction est appelée. Elle va lire le registre d'état afin de gérer l'événement avec le code ad hoc. Cette méthode ne consomme pas de CPU en dehors des moments où le périphérique doit communiquer.

Enfin, le mode avec DMA utilise un composant matériel permettant d'effectuer des échanges entre le contrôleur d'entrées-sorties et la mémoire, sans l'aide du CPU. Le DMA comprend :

- un registre d'adresse qui reçoit l'adresse du premier caractère à transférer,
- un registre de comptage qui reçoit le nombre de caractères à transférer,
- un registre de commande qui reçoit le type d'opération à effectuer (lecture ou écriture),
- une zone tampon permettant le stockage de données,
- un composant électronique (sorte de processeur minimaliste), qui exécute un transfert sans utilisation du processeur central.

Les fonctions du pilote deviennent alors très simples : elles effectuent simplement les opérations d'initialisation du DMA : initialisation des registres, et envoi d'un ordre au « processeur DMA ». Dès cet ordre reçu, l'opération d'entrée-sortie s'effectue sans l'utilisation du CPU central, qui peut vaquer à d'autres opérations. A la fin du transfert, une interruption est levée pour prévenir le pilote.

15.1.3 Ordonnancement des requêtes des pilotes

Comme nous l'avons vu, le pilote peut comporter une fonction `schedule` dont le rôle est d'ordonnancer les requêtes à destination du périphérique qu'il contrôle.

Par exemple, dans le cas des accès aux disques durs, l'accès a un secteur du disque se décompose en deux parties :

- le temps de positionnement du bras correspond au temps nécessaire pour que le bras portant la tête de lecture vienne se positionner sur la piste contenant le secteur ;
- le temps de latence correspond au temps nécessaire pour qu'une fois la tête placée sur la bonne piste, le secteur passe sous la tête de lecture. A noter que ce temps de latence dépend de la vitesse de rotation du disque ;
- le temps de lecture/transfert des données.

Les disques ayant des vitesses de rotation très rapides (de 4'500 à plus de 10'000 tours/minute), et les bus sachant communiquer avec des vitesses de l'ordre du Mbits/s, il s'avère que le temps de positionnement est l'opération la plus pénalisante lors d'une entrée-sortie sur le disque.

Une des première méthode pour accéder aux données consiste à utiliser l'algorithme « FCFS » ou « First Come, First Served ». Il s'agit de notre classique FIFO, où chaque demande est traitée immédiatement, dans l'ordre où elle arrive. Avec cette technique, la plus simple à mettre en oeuvre, le mouvement du bras est aléatoire, et n'est pas optimisé pour minimiser son déplacement (et par conséquent, le temps d'accès).

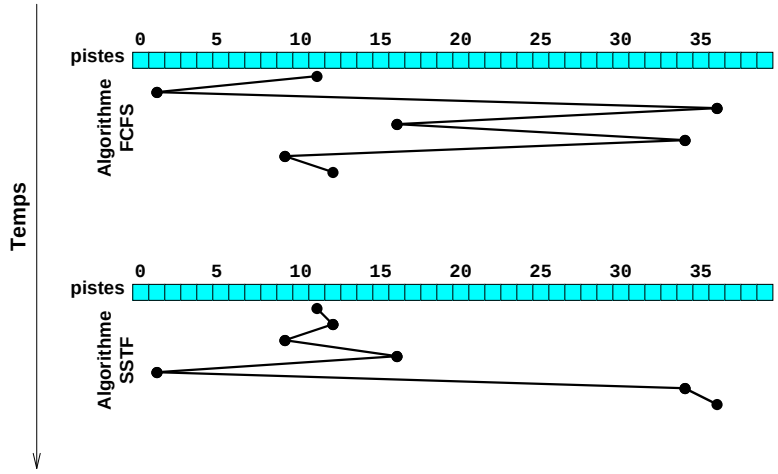
Aussi, afin de réduire la pénalité induite par les mouvements du bras, le pilote ordonnance les requêtes de la file d'attente du contrôleur de manière à réduire ces mouvements. Pour ce, il applique des heuristiques (algorithmes d'optimisation), dont les principales sont :

- SSTF (Shortest Seek Time First) ;
- SCAN (ascenseur ou balayage) ;
- C-SCAN.

Ces algorithmes sont efficaces lorsque le pilote doit demander de lire plusieurs secteurs entraînant ainsi une salve de déplacements de la tête de lecture/enregistrement sur plusieurs pistes.

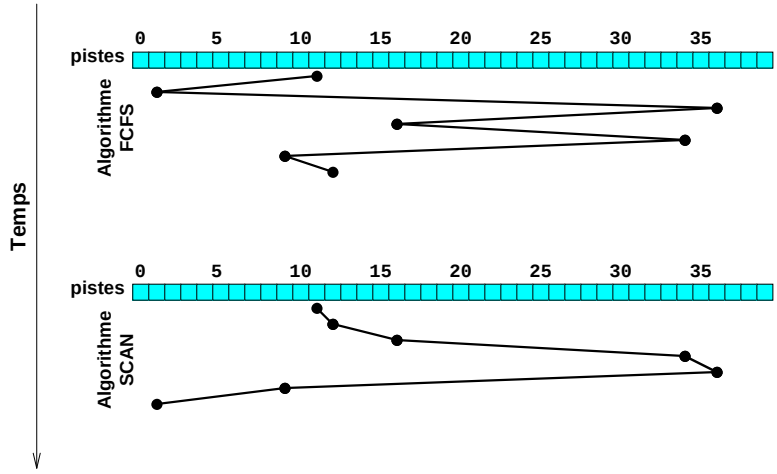
L'algorithme SSTF (*Shortest Seek Time First*) consiste à choisir, parmi tous les déplacements de tête possibles, celui induisant le moins de déplacement du bras.

Exemple : supposons que la tête soit positionnée sur la piste n°11, et que le pilote reçoive des ordres de lecture de secteurs situés sur les pistes 1, 36, 16, 34, 9, et 12. Le schéma suivant présente le résultat des déplacements avec l'algorithme FCFS, et ceux engendrés par l'algorithme SSTF.



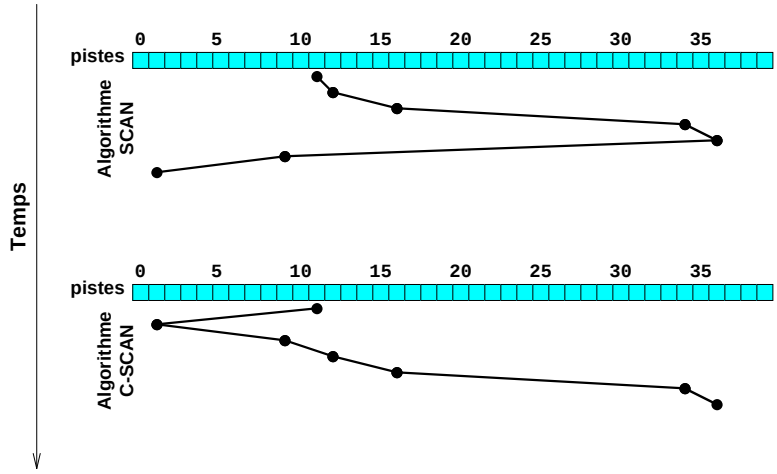
Déplacement des têtes sur un disque selon les algorithmes FCFS et SSTF

Le problème, avec cette politique, est le risque de famine que l'on rencontre pour les pistes excentrées par rapport à la position courante. Aussi, pour éviter ce phénomène, un autre algorithme consiste à faire balayer la tête de lecture le long des pistes, dans un sens, puis dans un autre. Les requêtes sont alors servies au fur et à mesure que les pistes sont balayées. Dans l'exemple précédent, nous allons supposer que la tête de lecture se trouve au dessus de la piste 11, et quelle se déplace dans le sens croissant des pistes. L'algorithme nous donne :



Déplacement des têtes sur un disque selon les algorithmes FCFS et SCAN

Une variante de cette heuristique est la politique C-SCAN. Il reprend le principe de l'algorithme précédent, sauf qu'une fois arrivé à une extrémité, plutôt que de répondre aux requêtes dans l'autre sens, il va positionner directement la tête à l'autre extrémité. Ainsi, le parcourt des pistes se fait toujours dans le même sens. Dans notre exemple :



Déplacement des têtes sur un disque selon les algorithmes SCAN et C-SCAN

15.2 Les entrées-sorties sous Linux

15.2.1 Fichiers spéciaux

Linux gère les entrées-sorties au travers des fichiers spéciaux (Cf. chapitre sur les fichiers). Ainsi un même appel système de type `write()` peut être utilisé pour écrire dans un fichier régulier, ou pour lancer une impression, en écrivant sur l'écran d'un terminal au travers de

son fichier spécial « /dev/tty0 », attaché au pilote de gestion des écrans. Classiquement, les fichiers spéciaux sont de deux types sous Unix/Linux :

- les fichiers de type bloc (b) correspondent aux périphériques structurés en blocs, pour lesquels l'accès direct est possible. L'accès à ces périphériques s'effectue au travers du cache de tampons (*buffer cache*) présenté au chapitre traitant des fichiers. Les périphériques de type « disques durs », « disquette », « CD-ROM/DVD-ROM », « clés USB », etc. appartiennent à cette catégorie ;
- les fichiers de type caractères (c) correspondent aux périphériques sans structure, sur lesquels les données ne sont accessibles que de façon séquentielle, octet par octet. Les périphériques comme l'imprimante, le terminal, le scanner, la carte son, ou encore souris et joystick font partie de cette catégorie.

Chaque fichier spécial est identifié par 3 attributs :

- son type (« c » ou « b ») ;
- un numéro sur un octet (compris entre 1 et 255), appelé numéro majeur, qui identifie le pilote gérant le périphérique ;
- un numéro appelé numéro mineur, qui identifie, pour un pilote donné, un des périphériques physiques qui lui est assigné.

Par exemple, dans un noyau 2.6 sur architecture PC (avec un chip UART 16550), les ports série sont gérés par un pilote identité par le numéro majeur 4. Pour un PC ayant 4 ports séries (identifiés sous DOS COM1, COM2, COM3, et COM4), les numéros mineurs correspondant seront les numéros de 64 à 67. Ainsi, un fichier spécial de type caractère ayant comme numéro majeur 4 et comme numéro mineur 64 fera référence au port RS-232 COM1.

A l'instar des systèmes de gestion de fichiers, les pilotes de périphériques s'enregistrent auprès du noyau (soit lors de sa compilation, soit lors du chargement d'un module).

Le noyau maintient deux tables de 256 entrées chacune, `blkdevs[]` et `chrdevs[]`, qui contiennent respectivement des descripteurs des périphériques blocs ou caractères qui lui sont connus. Chaque entrée donne le nom du pilote et les opérations liées au fichier spécial associé.

L'enregistrement des pilotes s'effectue au moyen des deux opérations suivantes :

- ```
  int register_chrdev(unsigned int major, const char *name, \
                      struct file_operations *fops);
  ```
- ```
  int register_blkdev(unsigned int major, const char *name, \
                      struct file_operations *fops);
  ```

Ainsi, un pilote pour un périphérique de type imprimante effectue l'appel suivant pour s'enregistrer auprès du noyau :

```
register_chrdev (LP_MAJOR, "lp", &lp_fops);
```

« `LP_MAJOR` » correspond au numéro majeur affecté à la classe des périphériques de type « lp » (les imprimantes parallèles), tandis que « `lp_fops` » est un pointeur sur une table des opérations associées au périphérique. Ainsi, lorsque l'utilisateur fait une opération « `write()` » sur de fichier « `/dev/lp0` » par exemple, le VFS accède à l'entrée de la table « `chrdevs[]` » concernant les périphériques de type « `lp` », et il en déduit l'opération « `write()` » via le pointeur « `lp_fops` » qui correspond au périphérique.

A l'inverse, un pilote de désenregistre du noyau à l'aide des fonctions :

- `int unregister_chrdev(unsigned int major, const char *name);`
- `int unregister_blkdev(unsigned int major, const char *name);`

Par convention, la plupart des fichiers spéciaux sont créés dans le répertoire « `/dev` », à l'aide de la commande « `mknod` » :

```
# mknod [OPTION]... NOM TYPE [MAJEUR MINEUR]
```

Cette commande « mknod » fait appel à une primitive système « mknod() » dont le synopsis est :

<<
```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/sysmacros.h>
#include <fcntl.h>
#include <unistd.h>
int mknod(const char *ref, mode_t mode, dev_t dev);
```
>>

Le paramètre « ref » spécifie le nom du fichier à créer. Le paramètre « mode » donne les permissions et le type de fichier à créer. Ce type est représenté par l'une des constantes suivantes:

- S_IFREG : fichier régulier ;
- S_IFCHR : fichier spécial de type caractère ;
- S_IFBLK : fichier spécial de type bloc ;
- S_IFIFO : tube nommé.

Le paramètre « dev » correspond à l'identificateur du périphérique (couple numéro majeur = 8 bits de poids fort, numéro mineur = 8 bits de poids faible). La macro « major(d) » extrait le numéro majeur d'un identificateur de périphérique, « minor(d) » extrait le numéro mineur d'un identificateur de périphérique, et « makedev(maj,min) » calcule l'identificateur ayant les numéros majeur « maj » et mineur « min ».

Initialement, il fallait maintenir les fichiers spéciaux dans le répertoire « /dev » en fonction des développements du noyau. Un script (souvent appelé MAKEDEV, souvent situé sous « /dev ») avait pour rôle de créer les fichiers spéciaux à l'installation du système d'exploitation. De récents développements dans le noyau Linux (***devfs***, ou plus récemment ***udev***) permettent la création automatique de ces fichiers spéciaux par le noyau lui-même, dès qu'un pilote s'enregistre avec une des fonctions « register_chrdev() » ou « register_blkdev() ».

Un « ls -l /dev » permet de lister les fichiers spéciaux, avec leur nom, leur type, leurs numéros majeur et mineur. Par exemple (sous-ensemble du vrai résultat) :

<<
```
crw-rw----  1 root audio    14,  14 Oct 18  2004 admmidi0
crw-rw----  1 root audio    14,  30 Oct 18  2004 admmidi1
crw-rw----  1 root audio    14,  12 Oct 18  2004 adsp0
crw-rw----  1 root audio    14,  28 Oct 18  2004 adsp1
crw-rw----  1 root video    10, 175 Oct 18  2004 agpgart
brw-rw----  1 root disk      3,   0 Oct 18  2004 hda
brw-rw----  1 root disk      3,   1 Oct 18  2004 hda1
brw-rw----  1 root disk      3,   2 Oct 18  2004 hda2
brw-rw----  1 root disk      3,   3 Oct 18  2004 hda3
brw-rw----  1 root disk      3,   4 Oct 18  2004 hda4
brw-rw----  1 root disk      3,   5 Oct 18  2004 hda5
brw-rw----  1 root disk      3,   6 Oct 18  2004 hda6
brw-rw----  1 root disk      3,   7 Oct 18  2004 hda7
brw-rw----  1 root disk      3,   8 Oct 18  2004 hda8
crw-rw----  1 root lp        6,   0 Oct 18  2004 lp0
crw-rw----  1 root lp        6,   1 Oct 18  2004 lp1
crw-rw-rw-  1 root root      1,   3 Mar 29  2005 null
brw-rw----  1 root disk      1,   0 Mar 29  2005 ram0
brw-rw----  1 root disk      1,   1 Mar 29  2005 ram1
crw-rw-rw-  1 root root      1,   8 Mar 29  2005 random
```

```
crw-------  1 root tty       4,   0 Mar 29  2005 tty0
crw-------  1 root root      4,   1 May 18 14:00 tty1
crw-rw----  1 root dialout   4,  64 Oct 18  2004 ttyS0
crw-rw----  1 root dialout   4,  65 Oct 18  2004 ttyS1
crw-rw----  1 root dialout   4,  66 Oct 18  2004 ttyS2
crw-rw----  1 root dialout   4,  67 Oct 18  2004 ttyS3
crw-rw----  1 root dialout   4,  68 Oct 18  2004 ttyS4
crw-rw-rw-  1 root root      1,   5 Mar 29  2005 zero
```

>>

15.2.2 Opérations de contrôle sur un périphérique

La primitive « `ioctl()` » permet de modifier les paramètres d'un périphérique. Son prototype est :

<<

```
#include <sys/ioctl.h>
int ioctl(int fd, int cmd, char *arg);
```

>>

L'opération désignée par « `cmd` » avec l'argument « `arg` » est réalisée sur le périphérique désigné par le descripteur « `fd` ».

Le type des opérations de contrôle étant fonction des périphériques, celles-ci sont très diverses. Elles ne font pas partie du programme de ce cours. Elles peuvent permettre de modifier les contrôles de flux (xon/xoff ou matériel pour un lien série par exemple), etc.

15.2.3 Multiplexage des entrées-sorties

La primitive « `select()` » permet à un processus de se mettre en attente sur un ensemble de descripteurs, qui peuvent correspondre à un ensemble de périphériques. Le processus s'endort et est réveillé, soit dès qu'au moins un descripteur de l'ensemble est prêt, soit lorsque la temporisation éventuellement associée à « select() » est écoulée :

<<

```
SYNOPSIS
       /* D'après POSIX.1-2001 */
       #include <sys/select.h>

       /* D'après les standards précédents */
       #include <sys/time.h>
       #include <sys/types.h>
       #include <unistd.h>

       int select(int nfds, fd_set *readfds, fd_set *writefds, \
                  fd_set *exceptfds, struct timeval *timeout);

       void FD_CLR(int fd, fd_set *set);
       int FD_ISSET(int fd, fd_set *set);
       void FD_SET(int fd, fd_set *set);
       void FD_ZERO(fd_set *set);

DESCRIPTION
       select() permet à un programme de surveiller plusieurs
       descripteurs de fichiers, en attendant qu'au moins l'un de
       ces descripteurs soit « prêt » pour une certaine classe
       d'opérations d'entrée-sortie. Un descripteur de fichier est
       considéré comme prêt s'il est possible d'effectuer
```

l'opération correspondante (par exemple read(2)) sans bloquer.

Il y a trois ensembles indépendants de descripteurs surveillés simultanément. Ceux de l'ensemble **readfds** seront surveillés pour vérifier si des caractères deviennent disponibles en lecture. Plus précisément, on vérifie si un appel système de lecture ne bloquera pas – en particulier un descripteur en fin de fichier sera considéré comme prêt. Les descripteurs de l'ensemble **writefds** seront surveillés pour vérifier si une écriture ne bloquera pas. Ceux de **exceptfds** seront surveillés pour l'occurrence de conditions exceptionnelles. En sortie, les ensembles sont modifiés pour indiquer les descripteurs qui ont changé de statut. Chacun des trois ensembles de descripteurs peut être NULL si aucun descripteur de fichier ne doit être surveillé pour cette classe d'événements.

Quatre macros sont disponibles pour la manipulation des Ensembles : FD_ZERO() efface un ensemble. FD_SET() et FD_CLR() ajoutent et suppriment un descripteur dans un ensemble. FD_ISSET() vérifie si un descripteur est contenu dans un ensemble, principalement utile après le retour de select().

nfds est le numéro du plus grand descripteur des 3 ensembles, plus 1.

timeout est une limite supérieure au temps passé dans select() avant son retour. Elle peut être nulle, ce qui conduit select() à revenir immédiatement (ce qui sert pour des surveillance en polling). Si le **timeout** est NULL (aucune limite), select() peut bloquer indéfiniment.

La structure temporelle concernée est définie dans <sys/time.h> comme ceci :

```
struct timeval {
    long    tv_sec;         /* secondes      */
    long    tv_usec;        /* microsecondes */
};
```

Certaines applications appellent select() avec trois ensembles de descripteurs vides, **nfds** nul, et un délai **timeout** non nul, afin d'endormir, de manière portable le processus avec une précision plus fine que la seconde.

Sous Linux, la fonction select() modifie **timeout** pour indiquer le temps restant, mais la plupart des autres implémentations ne le font pas (POSIX.1-2001 autorise les deux comportements). Ceci pose des problèmes à la fois pour porter sur d'autres systèmes du code développé sous Linux qui utilise cette valeur de **timeout** modifiée, et pour porter sous Linux du code qui réutilise plusieurs fois la structure **timeval** sans la réinitialiser. La meilleure attitude à adopter est de considérer **timeout** comme indéfini après le retour de select().

VALEUR RENVOYÉE

En cas de réussite select() renvoie le nombre de descripteurs dans les trois ensembles de descripteurs retournés (c'est-à-dire le nombre total de bits à 1 dans **readfds**, **writefds**, **exceptfds**) qui peut être nul si le délai de **timeout** a expiré avant que quoi que ce soit d'intéressant ne se produise. Ils retournent -1 s'ils échouent, auquel cas, **errno** contient le code d'erreur ; les ensembles et **timeout** ne sont plus définis, ne vous fiez plus à leur contenu après une erreur.

ERREURS

EBADF Un descripteur de fichier invalide était dans l'un des ensembles (peut-être un descripteur déjà fermé, ou sur lequel une erreur s'est produite.)

EINTR Un signal a été intercepté.

EINVAL **nfds** est négatif ou la valeur contenue dans **timeout** est invalide.

ENOMEM Pas assez de mémoire pour le noyau.

EXEMPLE

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

int main(void)
{
    fd_set rfds;
    struct timeval tv;
    int retval;
    /* Surveiller stdin (fd 0) en attente d'entrées */
    FD_ZERO(&rfds);
    FD_SET(0, &rfds);
    /* Attends jusqu'à 5 secondes. */
    tv.tv_sec = 5;
    tv.tv_usec = 0;
    retval = select(1, &rfds, NULL, NULL, &tv);
    /* Considérer tv comme indéfini maintenant ! */
    if (retval == -1)
        perror("select()");
    else if (retval)
        printf("Des données sont disponibles maintenant\n");
        /* FD_ISSET(0, &rfds) est alors vrai. */
    else
        printf("Aucune donnée durant les cinq secondes.\n");
    return(0);
}
```

>>