Programmation système

Juliusz Chroboczek

4 mars 2024

Table des matières

1 Systèmes d'exploitation **6**
- 1.1 Accès au matériel . . . 6
  - 1.1.1 Programmation sur matériel nu . . . 6
  - 1.1.2 HAL et moniteur . . . 6
  - 1.1.3 Système d'exploitation . . . 7
  - 1.1.4 Virtualisation et abstraction du matériel . . . 7
- 1.2 Le système Unix . . . 8
  - 1.2.1 Fonctions *stub* . . . 8
  - 1.2.2 La norme POSIX . . . 8

2 Le système de fichiers **10**
- 2.1 Structure du système de fichiers . . . 10
- 2.2 Entrées/sorties de bas niveau . . . 13
- 2.3 Tampons . . . 15
  - 2.3.1 Tampon simple . . . 16
  - 2.3.2 Tampon avec pointeur de début des données . . . 16
  - 2.3.3 Tampon circulaire . . . 17
- 2.4 La bibliothèque `stdio` . . . 18
  - 2.4.1 La structure `FILE` . . . 18
  - 2.4.2 Entrées/sorties de haut niveau . . . 19
  - 2.4.3 Interaction entre `stdio` et le système . . . 20
  - 2.4.4 Entrées/sorties formatées . . . 21
- 2.5 Le pointeur de position courante . . . 22
  - 2.5.1 L'appel système `lseek` . . . 22
  - 2.5.2 Le mode « ajout à la fin » . . . 23
- 2.6 Troncation et extension des fichiers . . . 24
  - 2.6.1 Extension implicite . . . 24
  - 2.6.2 Troncation et extension explicite . . . 24
- 2.7 I-nœuds . . . 25
  - 2.7.1 Lecture des i-nœuds . . . 25
  - 2.7.2 Modification des i-nœuds . . . 27
  - 2.7.3 Liens et renommage de fichiers . . . 29
  - 2.7.4 Comptage des références . . . 30
- 2.8 Manipulation de répertoires . . . 30
  - 2.8.1 Résolution de noms . . . 30
  - 2.8.2 Lecture des répertoires . . . 31
- 2.9 Liens symboliques . . . 32

3 Processus et communication **34**
3.1 Programmes et processus . . . 34
3.1.1 Protection mémoire . . . 34
3.1.2 Structure d'un programme . . . 35
3.1.3 Carte mémoire d'un processus . . . 35
3.1.4 L'ordonnanceur . . . 36
3.1.5 Manipulation des processus . . . 37
3.1.6 Vie et mort des processus . . . 39
3.1.7 Exécution de programme . . . 40
3.1.8 Exécution d'un programme dans un nouveau processus . . . 41
3.2 Communication par fichiers . . . 42
3.2.1 Synchronisation à l'aide de `wait` . . . 42
3.2.2 Cohérence des accès . . . 43
3.2.3 *Locks* . . . 44
3.3 Communication par messages . . . 46
3.3.1 Tubes anonymes . . . 47
3.3.2 Tubes nommés . . . 48
3.3.3 Semantique des tubes . . . 48
3.3.4 *Sockets* de domaine Unix . . . 49
3.4 Redirections . . . 50
3.4.1 Descripteurs standard . . . 50
3.4.2 Redirections . . . 51
3.4.3 Exemple . . . 51
3.5 La mémoire virtuelle . . . 51
3.5.1 Pagination . . . 53
3.5.2 Applications . . . 54
3.6 *Memory mapping* . . . 55
3.6.1 *Mapping* anonyme privé . . . 56
3.6.2 *Mapping* anonyme partagé . . . 56
3.6.3 *Mapping* privé de fichiers . . . 56
3.6.4 *Mapping* de fichiers partagé . . . 57
3.6.5 Problèmes liés à `mmap` . . . 57
3.7 Mémoire partagée et sémaphores POSIX . . . 58
3.7.1 Mémoire partagée POSIX . . . 58
3.7.2 Sémaphores POSIX . . . 59
3.8 Cohérence des accès . . . 60
3.8.1 Le *buffer cache* . . . 60
3.8.2 Les caches du processeur . . . 61
3.8.3 Cohérence de `mmap` . . . 64
3.9 Opérations atomiques . . . 64
3.9.1 Entiers atomiques . . . 65
3.9.2 Implémementation des *locks* . . . 66
3.9.3 Digression : mémoire transactionnelle . . . 68

3.10 Signaux 69
3.10.1 Quelques signaux utiles 69
3.10.2 Envoi de signaux 70
3.10.3 Gestion des signaux 70
3.10.4 Les signaux ne sont pas fiables 71
3.10.5 Les signaux sont asynchrones 71
3.11 Signaux et appels système bloquants 75

4 Entrées-sorties non-bloquantes **76**
4.1 Mode non-bloquant 76
4.2 Attente active 76
4.3 L'appel système `select` 77
4.3.1 Mesure du temps 78
4.3.2 Ensembles de descripteurs 78
4.3.3 L'appel système `select` 79
4.3.4 Exemples 79
4.3.5 Bug 80
4.4 Boucles à événements 80
4.4.1 Boucles à événements et signaux 80

Préface

L'ancienne version de ce document, datée du 29 septembre 2012, était basée sur les notes du cours de Programmation Système de 3e année que j'ai assuré entre 2008 et 2012. La version datée du 4 septembre 2018 a intégré les notes du cours de 4e année que j'ai assuré entre 2014 et 2017.

1 Systèmes d'exploitation

1.1 Accès au matériel

Dans un ordinateur, le processeur communique non seulement avec la mémoire principale, qui contient le code à exécuter et les données sur lesquelles celui-ci opère, mais aussi avec des *périphériques de stockage de masse*, ou mémoire secondaire, et des *périphériques d'entrées sorties*. Le programmeur doit avoir accès à des facilités qui lui permettent d'accéder à ceux-ci.

1.1.1 Programmation sur matériel nu

Il est bien sûr possible d'accéder aux périphériques directement à partir du code du programme (le *code utilisateur*) ; on parle alors de *programmation sur matériel nu*. Cette approche présente plusieurs désavantages :

- le code dépend d'une connaissance intime des détails du matériel, et le changement d'un périphérique force sa récriture ;
- le code n'est pas modulaire : l'accès au matériel est mélangé à la logique du programme elle-même.

La programmation sur matériel nu n'est plus praticable sur les ordinateurs classiques, du fait de l'existence de matériel complexe, hétérogène, et partagé entre différentes applications (c'est notamment le cas des disques). Elle se pratique encore dans les systèmes dits *embarqués*, par exemple votre four à micro-ondes. Cependant, elle est de plus en plus rare : même votre téléphone portable a un système d'exploitation.

1.1.2 HAL et moniteur

Afin d'éviter de rendre nos programmes dépendants des détails du matériel et afin de séparer la logique du programme de l'accès au matériel, il est naturel de mettre le code dit *de bas niveau* (proche du matériel) dans une bibliothèque séparée. Une telle bibliothèque est communément appelée *HAL*, acronyme développé, selon les auteurs, en *Hardware Abstraction Layer* ou en *Hardware Access Library*. (De nos jours, la HAL est elle-même modulaire — elle est constituée de l'ensemble des *pilotes de périphérique (drivers)*.)

Moniteur Si la même HAL est utilisée par tous les programmes, elle est souvent *résidente*, c'est à dire chargée une seule fois lors du lancement *(boot)* du système et présente en mémoire à tout moment. Un *moniteur* est composé d'une HAL résidente et d'une interface utilisateur permettant notamment de charger des programmes en mémoire et de les exécuter, et parfois aussi de contrôler manuellement les périphériques (par exemple de renommer des fichiers sur disque, ou de manipuler l'imprimante). Un exemple bien connu de moniteur est le système *MS-DOS*.

1.1.3 Système d'exploitation

Un moniteur permet d'isoler le code utilisateur des dépendances matérielles ; cependant, l'utilisation du moniteur n'est pas obligatoire, et il reste possible de le contourner et d'accéder directement au matériel. En conséquence, un moniteur n'est pas suffisant pour s'assurer qu'aucune contrainte de sécurité ou de modularité n'est violée par le code utilisateur, ce qui est important notamment sur une machine partagée entre plusieurs utilisateurs.

Un *système d'exploitation*, ou *noyau du système d'exploitation (operating system kernel)*, est une couche située entre le logiciel utilisateur et le matériel dont l'utilisation est obligatoire pour tout processus voulant accéder au matériel.

Protection matérielle et code privilégié Ce qui précède implique que le noyau doit empêcher le code utilisateur d'accéder directement au matériel, ce qui est généralement implémenté en distinguant au niveau matériel entre deux types de code : le code dit *privilégié* ou *code noyau*, qui a le droit d'accéder au matériel, et le code dit *non-privilégié* ou *code utilisateur*, qui n'en a pas le droit. Lorsqu'un programme utilisateur essaie d'accéder (délibérément, ou, comme c'est le plus souvent le cas, par erreur) au matériel ou à des parties de la mémoire qui lui sont interdites, le matériel de *protection* passe la main au noyau qui, typiquement, tue le processus coupable [1].

Appel système L'invocation de code privilégié par du code non-privilégié se fait au moyen d'un *appel système*. Un appel système est semblable à un appel de fonction, mais effectue en plus deux changements de niveau de privilège : une transition vers le mode privilégié lors de l'appel, et une transition vers le mode non-privilégié lors du retour. De ce fait, un appel système est lent, typiquement entre 1 000 et 10 000 fois plus lent qu'un appel de fonction sur du matériel moderne.

1.1.4 Virtualisation et abstraction du matériel

La vision donnée par le système des périphériques n'est pas complètement uniforme. Dans certains cas, le système se contente de vérifier les privilèges du processus avant de lui donner accès au « vrai » matériel ; on parle alors de *médiation*.

Dans d'autres cas, le système présente au processus utilisateur des périphériques *virtuels*, qui se comportent comme de vrais périphériques, mais ne correspondent pas directement à la réalité ; par exemple, il peuvent être plus nombreux que les périphériques réels, c'est souvent le cas de la mémoire et des terminaux. On parle alors de *virtualisation* du matériel.

Enfin, il arrive que le système présente des périphériques *abstraits*, de plus haut niveau que les périphériques réels. C'est généralement le cas des disques (qui sont présentés comme une arborescence de fichiers plutôt qu'une suite de secteurs) et des interfaces réseau. On parle alors d'*abstraction*.

1. Unix, par exemple, envoie un signal `SIGSEGV` ou `SIGBUS` au processus.

1.2 Le système Unix

Unix, une famille de systèmes d'exploitation basée sur le système Unix des Bell Labs d'AT&T, développé dans les années 70, servira de base à ce cours. La famille Unix a aujourd'hui de nombreux représentants, libres (Linux, FreeBSD, NetBSD, Minix) et propriétaires (Mac OS X).

1.2.1 Fonctions *stub*

Le mécanisme exact d'appel système dépend du matériel. Pour cette raison, chaque système Unix inclut dans la bibliothèque C standard (`libc`) un certain nombre de fonctions appelées *stubs* ou *wrappers* dont chacune a le rôle d'invoquer un appel système. Par exemple, l'appel système `time` est invoqué par la fonction

```
int time(int *);
```

déclarée dans le fichier d'entête `<time.h>`.

Le manuel Unix (accessible à l'aide de la commande `man`) distingue entre les *stubs* d'appels système, qu'il documente dans la section 2, et les « vraies » fonctions, qu'il documente dans la section 3.

Convention d'appel et variable `errno` La bibliothèque standard définit une variable globale appelée `errno` qui sert à communiquer les résultats d'erreur entre les fonctions *stub* et le code utilisateur. Lorsqu'un appel système réussit, une fonction *stub* retourne un entier positif ou nul [2] ; lorsqu'il échoue, elle stocke le numéro de l'erreur dans la variable `errno`, et retourne une valeur strictement négative.

La variable `errno` ainsi que des constantes symboliques pour les codes d'erreurs sont déclarées dans le fichier d'entête `<errno.h>`. Pour afficher les messages d'erreurs, on peut utiliser les fonctions `strerror` et `perror` (déclarées dans `<string.h>` et `<stdio.h>` respectivement).

Exécution d'une fonction *stub* Une fonction *stub* effectue les actions suivantes :

- elle stocke les arguments de l'appel dans les bons registres ;
- elle invoque l'appel système ;
- elle interprète la valeur de retour et, si besoin, positionne la variable `errno`.

1.2.2 La norme POSIX

Au début des années 1990, Unix s'était morcelé en un certain nombre de systèmes semblables mais incompatibles. L'écriture d'un programme portable entre les différents dialectes d'Unix était devenue un exercice inutilement excitant. Le norme *IEEE 1003*, communément appelée *POSIX (Portable Operating System Interface)*, a défini un ensemble de fonctions disponibles sur tous les systèmes Unix. Tous les Unix modernes, libres ou propriétaires, implémentent cette norme.

Les interfaces POSIX restent en grande partie compatibles avec les interfaces Unix traditionnelles, mais remplacent les types concrets (tels que `int`) par des types dits « opaques » dont la

2. Il y a quelques exceptions à cette convention.

définition précise n'est pas spécifiée par POSIX, et est donc laissée à la discrétion de l'implémentation. Par exemple, POSIX définit l'appel `time` comme

```
time_t time(time_t *);
```

où le choix du type concret pour lequel `time_t` est un alias est laissé à l'implémenteur. Sur certains systèmes, `time_t` est un alias pour `int` :

```
typedef int time_t;
```

tandis que sur d'autres c'est un alias pour `long int` :

```
typedef long int time_t;
```

ou même `long long int`.

POSIX considère la différence entre une fonction et un appel système comme étant un détail de l'implémentation, et ne différencie donc pas entre les sections 2 et 3 du manuel.

Depuis 2003, chaque version de la norme POSIX est identique à une version de la *Single Unix Specification*, qui est disponible en ligne [3].

3. `https://pubs.opengroup.org/onlinepubs/007908799/index.html`

2 Le système de fichiers

Le disque d'un ordinateur est une ressource partagée : il est utilisé par plusieurs programmes et, sur un système multi-utilisateur, par plusieurs utilisateurs. Les données sur le disque sont stockées dans des *fichiers* (*file* en anglais), des structures de données qui apparaissent aux programmes comme des suites finies d'octets. La structure de données qui organise les fichiers sur le disque s'appelle le *système de fichiers (file system)*.

2.1 Structure du système de fichiers

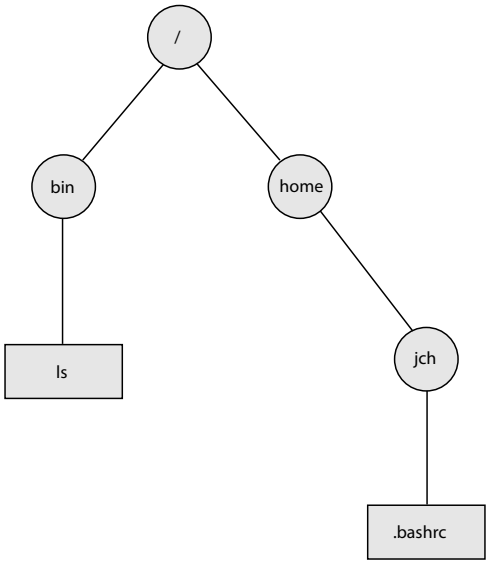


FIGURE 2.1 — Vision abstraite du système de fichiers

Vision abstraite du système de fichiers Du point de vue du programmeur et de l'utilisateur, le système de fichiers apparaît comme un arbre dont les feuilles sont les fichiers (figure 2.1). Un nœud interne de cet arbre s'appelle un *répertoire (directory)*.

Un fichier est identifié par son *chemin d'accès (pathname)*, qui est constitué du chemin depuis la racine dont les composantes sont séparées par des *slashes* « `/` ». Par exemple, le chemin d'accès du fichier en bas à droite de la figure 2.1 est `/home/jch/.bashrc`.

Vision concrète du système de fichiers Concrètement (figure 2.2), le système de fichiers est constitué de deux types de structures de données : les *i-nœuds (i-nodes)* et les *données de fichier*.

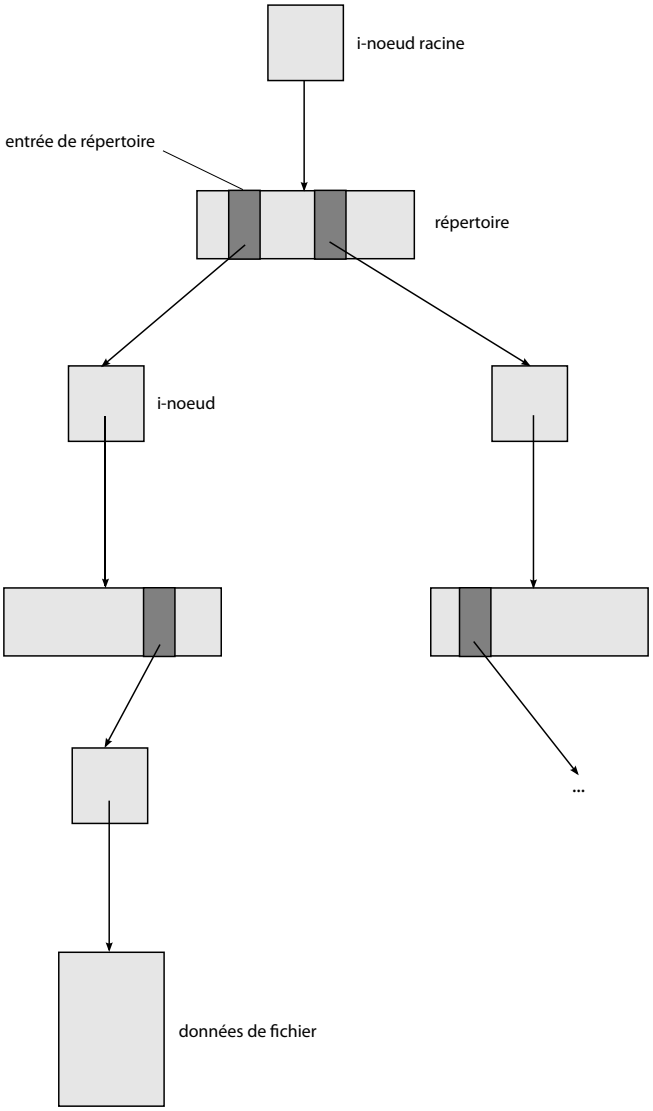


FIGURE 2.2 — Vision concrète du système de fichiers

Ces dernières peuvent être soit des *données de fichier ordinaire*, ou des *répertoires*.

Intuitivement, l'i-nœud « c'est » le fichier. Un i-nœud est une structure de taille fixée (quelques centaines d'octets) qui contient un certain nombre de *méta-données* à propos d'un fichier — son type (fichier ou répertoire), sa taille, sa date d'accès, son propriétaire, etc. — ainsi que suffisamment d'informations pour retrouver le contenu du fichier.

Le contenu du fichier est (une structure de données qui code) une suite d'octets. Dans le cas d'un fichier ordinaire, ce contenu est interprété par l'application, il n'a pas de signification pour le système. Dans le cas d'un répertoire, par contre, c'est le système qui l'interprère comme une suite d'*entrées de répertoire*, dont chacune contient un *nom de fichier* et une *référence à un i-nœud*.

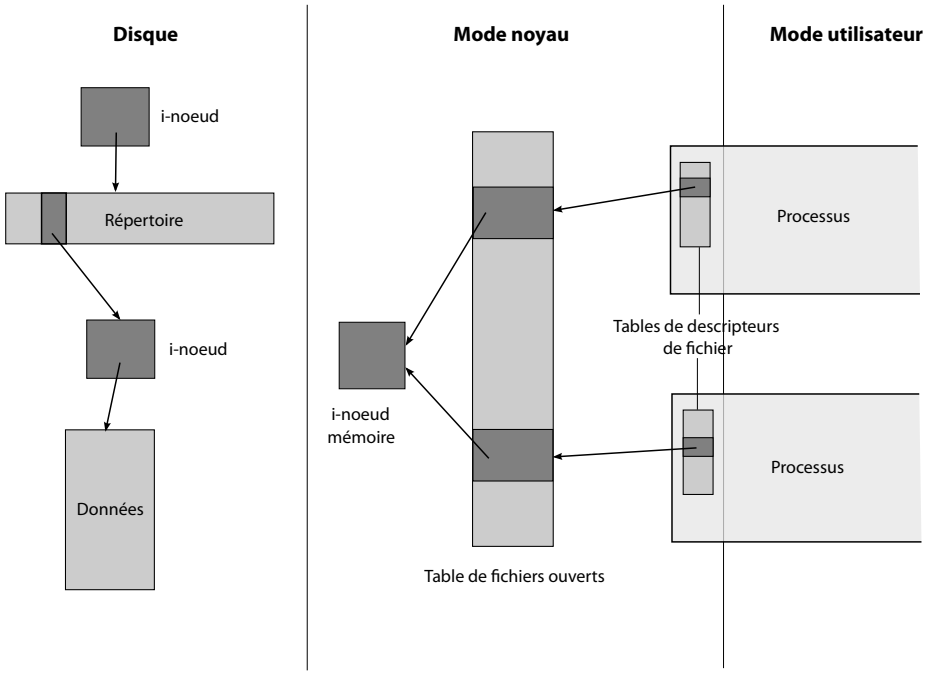


FIGURE 2.3 — Structures de données sur disque et en mémoire

Structures de données en mémoire Lorsque l'utilisateur demande au système de manipuler un fichier (en effectuant l'appel système `open`, voir ci-dessous), celui-ci charge l'i-nœud correspondant en mémoire ; il y a alors dans la mémoire du noyau un *i-nœud mémoire*, une *entrée de la table de fichiers ouverts* et une *entrée de la table de descripteurs de fichier du processus*.

L'i-nœud mémoire est une structure de données contenant le contenu de l'i-nœud (disque) ainsi que quelques données comptables supplémentaires (notamment le numéro de périphérique à partir duquel cet i-nœud a été chargé).

L'*entrée de la table de fichiers ouverts* est une structure de données globale qui réfère à l'i-nœud mémoire ; cette structure de données contient elle-même quelques champs supplémentaires, notamment le *pointeur de position courante* (voir partie 2.5 ci-dessous).

L'*entrée de la table de descripteurs de fichiers* est une structure qui réfère à l'entrée de la table de fichiers ouverts. À la différence des deux structures de données ci-dessus, qui sont *globales*, il s'agit d'une structure qui est locale au processus ayant ouvert le fichier.

Ces structures de données vivent dans la mémoire du noyau ; le code utilisateur ne peut donc pas y référer directement. Le code utilisateur indique une entrée de la table de fichiers à travers un petit entier, l'indice dans la table de descripteurs de fichiers, communément appelé lui-même *descripteur de fichier*.

2.2 Entrées/sorties de bas niveau

Sous Unix, les entrées/sorties sur les fichiers sont réalisées par les appels système `open`, `close`, `read`, `write`.

`open` L'appel système `open` est défini [1] par

```
int open(char *pathname, int flags,... /* int mode */);
```

Son rôle est de charger un i-nœud en mémoire, créer une entrée de la table de fichiers qui y réfère, et créer une entrée dans la table de descripteurs de fichiers du processus courant qui réfère à cette dernière. Il retourne le descripteur de fichiers correspondant, ou -1 en cas d'erreur (et la variable globale `errno` est alors positionnée).

Le paramètre `pathname` est le nom du fichier à ouvrir. Le paramètre `flags` peut valoir

- `O_RDONLY`, ouverture en lecture seulement ;
- `O_WRONLY`, ouverture en écriture seulement ;
- `O_RDWR`, ouverture en lecture et écriture.

Cette valeur peut en outre être combinée (à l'aide de la disjonction bit-à-bit « | ») avec

- `O_CREAT`, créer le fichier s'il n'existe pas ;
- `O_EXCL`, échouer si le fichier existe déjà ;
- `O_TRUNC`, tronquer le fichier s'il existe déjà ;
- `O_APPEND`, ouvrir le fichier en mode « ajout à la fin ».

Le troisième paramètre, `mode_t mode`, n'est présent que si `O_CREAT` est dans `flags`. Il spécifie les permissions maximales du fichier créé. On pourra le positionner à 666 octal (soit « `0666` » en C).

`close` L'appel système `close` libère une entrée de la table de descripteurs du processus courant, ce qui peut avoir pour effet de libérer une entrée de la table de fichiers ouverts et un i-nœud en mémoire. Il est défini comme

```
int close(int fd);
```

Il retourne 0 en cas de succès, et -1 en cas d'échec [2]. Cependant, la plupart du code Unix traditionnel ne vérifie pas le résultat retourné par `close` (ce qui cause parfois des problèmes sur les systèmes de fichiers montés à travers le réseau).

1. Plus précisément, c'est la fonction *stub* correspondante qui est définie comme-ça.
2. Et la variable `errno` est alors positionnée. Je ne le mentionne plus, désormais.

`read` L'appel système `read` est défini par

```
ssize_t read(int fd, void *buf, size_t count);
```

Sa fonction est de lire au plus `count` octets de données depuis le fichier spécifié par le descripteur de fichiers `fd` et de les stocker à l'adresse `buf`.

L'appel `read` retourne le nombre d'octets effectivement lus, 0 à la fin du fichier, ou -1 en cas d'erreur.

`write` L'appel système `write` est défini comme

```
ssize_t write(int fd, void *buf, size_t count);
```

Sa fonction est d'écrire au plus `count` octets de données qui se trouvent à l'adresse `buf` dans le fichier spécifié par le descripteur de fichiers `fd`.

L'appel `write` retourne le nombre d'octets effectivement écrits [3], ou -1 en cas d'erreur.

Exemple Le programme suivant effectue une copie de fichiers. Comme nous le verrons, il est extrêmement inefficace.

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

int main(int argc, char **argv)
{
    int fd1, fd2, rc;
    char buf;

    if(argc != 3) {
        fprintf(stderr, "Syntaxe: %s f1 f2\n", argv[0]);
        exit(1);
    }

    fd1 = open(argv[1], O_RDONLY);
    if(fd1 < 0) {
        perror("open(fd1)");
        exit(1);
    }
```

3. Une écriture de moins de `count` octets s'appelle une *écriture partielle*. Une écriture partielle n'est normalement pas possible sur un fichier, mais elle est commune sur d'autres types de descripteurs de fichiers, par exemple les *pipes* ou les *sockets*.

```
    fd2 = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if(fd2 < 0) {
        perror("open(fd2)");
        exit(1);
    }

    while(1) {
        rc = read(fd1, &buf, 1);
        if(rc < 0) {
             perror("read");
             exit(1);
        }
        if(rc == 0)
            break;

        rc = write(fd2, &buf, 1);
        if(rc < 0) {
            perror("write");
            exit(1);
        }
        if(rc != 1) {
            fprintf(stderr, "Écriture interrompue");
            exit(1);
        }
    }

    close(fd1);
    close(fd2);
    return 0;
}
```

Ce programme effectue deux appels système pour chaque octet copié — ce qui est tragique. Le portable que j'utilisais en 2012 était capable d'effectuer jusqu'à 3 000 000 d'appels système par seconde ; ce programme ne sera donc pas capable de copier plus d'1,5 Mo de données par seconde environ.

2.3 Tampons

Pour résoudre le problème de performances soulevé ci-dessus, il suffit d'effectuer des transferts de plus d'un octet pour chaque appel système. Il faudra pour cela disposer d'une zone de mémoire pour stocker les données en cours de transfert. Une telle zone s'appelle un *tampon* (*buffer* en anglais).

2.3.1 Tampon simple

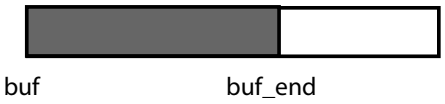


FIGURE 2.4 — Tampon simple

Un tampon simple (figure 2.4) est constitué d'une zone de mémoire `buf` et d'un entier `buf_end` indiquant la quantité de données dans le tampon :

```
void *buf;
int buf_end;
```

Les données valides dans le tampon se situent entre `buf` et buf + `buf_len` − 1.

Un tampon simple permet de lire les données de façon incrémentale ; cependant, les données doivent être écrites en une seule fois.

Exemple La version suivante du programme de copie utilise un tampon simple, et n'effectue que deux appels système tous les `BUFFER_SIZE` octets.

```
#define BUFFER_SIZE 4096
char buf[BUFFER_SIZE];
int buf_end;
...
    buf_end = 0;
    while(1) {
        rc = read(fd1, buf, BUFFER_SIZE);
        if(rc < 0) { perror("read"); exit(1); }
        if(rc == 0) break;
        buf_end = rc;
        rc = write(fd2, buf, buf_end);
        if(rc < 0) { perror("write"); exit(1); }
        if(rc != buf_end) {
            fprintf(stderr, "Écriture interrompue");
            exit(1);
        }
        buf_end = 0;
    }
...
```

2.3.2 Tampon avec pointeur de début des données

Un tampon avec pointeur de début des données (figure 2.5) contient un entier supplémentaire indiquant le début des données valides placées dans le tampon :

FIGURE 2.5 — Tampon avec pointeur de début des données

```
void *buf;
int buf_ptr;
int buf_end;
```

Les données valides se trouvent entre `buf + buf_ptr` et `buf + buf_end − 1`.

Une telle structure permet de lire et d'écrire les données de façon incrémentale ; cependant, lorsque le tampon est plein, il doit être complètement vidé avant qu'on ne puisse recommencer à lire.

Exemple Le programme précédent gère les lectures interrompues (`read` qui retourne moins de `BUFFER_SIZE` octets), mais pas les écritures interrompues (qui normalement ne peuvent pas avoir lieu sur un fichier). La version suivante utilise un tampon avec pointeur, et gère correctement les écritures interrompues :

```
#define BUFFER_SIZE 4096
char buf[BUFFER_SIZE];
int buf_ptr, buf_end;
...
    buf_ptr, buf_end = 0;
    while(1) {
        rc = read(fd1, buf, BUFFER_SIZE);
        if(rc < 0) { perror("read"); exit(1); }
        if(rc == 0) break;
        buf_end = rc;
        while(buf_ptr < buf_end) {
            rc = write(fd2, buf + buf_ptr, buf_end - buf_ptr);
            if(rc < 0) { perror("write"); exit(1); }
            buf_ptr += rc;
        }
        buf_ptr = buf_end = 0;
    }
...
```

2.3.3 Tampon circulaire

Avec un tampon avec pointeur, la variable `buf_end` ne revient au bord gauche du tampon que lorsque le tampon est vide ; lorsque la fin du tampon `buf_end` atteint le bord droit du tampon, une lecture n'est plus possible tant que le tampon n'est pas vidé.

FIGURE 2.6 — Tampon circulaire

Un *tampon circulaire* (figure 2.6) consiste des mêmes données qu'un tampon avec pointeur, mais les deux pointeurs sont interprétés *modulo* la taille du tampon : si `buf_end > buf_ptr`, les données valides ne sont pas connexes, mais constituées des deux parties `buf_ptr` à `BUFFER_SIZE` et 0 à `buf_end`.

Il existe une ambiguïté dans cette structure de données : lorsque `buf_end = buf_ptr`, il n'est pas clair si le tampon est vide ou s'il est plein. On peut différencier entre les deux conditions soit en évitant de mettre plus de `BUFFER_SIZE` − 1 octets dans le tampon, soit en utilisant une variable booléenne supplémentaire.

2.4 La bibliothèque `stdio`

La bibliothèque `stdio` a un double rôle : elle encapsule la manipulation des tampons dans un ensemble de fonctions pratiques à utiliser, et combine les entrées/sorties avec l'analyse lexicale et le formatage. Des bibliothèques analogues existent dans la plupart des langages de programmation, par exemple la bibliothèque `java.io` en Java.

Pour des raisons d'efficacité, il est parfois souhaitable de contourner `stdio` et manipuler les tampons soi-même ; par exemple, les programmes ci-dessus utilisent un seul tampon avec deux descripteurs de fichiers ; avec `stdio`, il faudrait utiliser deux tampons, et donc gâcher de la mémoire et effectuer des copies supplémentaires.

Du fait de leur portabilité, les fonctions de la bibliothèque `stdio` ont un certain nombre de limitations. En particulier, comme certains systèmes différencient entre fichiers binaires et fichiers texte, `stdio` ne permet pas facilement de mélanger les données textuelles et les données binaires au sein d'un même fichier (cette limitation peut être ignorée sur les systèmes POSIX).

2.4.1 La structure `FILE`

Les fonctions de `stdio` n'opèrent pas sur des descripteurs de fichiers, qui sont une notion spécifique à Unix, mais sur des pointeurs sur une structure qui représente un flot, la structure `FILE`. La définition de `FILE` dépend du système ; sur un système POSIX elle pourrait par exemple être définie comme suit :

```
#define BUFSIZ 4096

#define FILE_ERR 1
#define FILE_EOF 2

typedef struct _FILE {
```

```
    int fd;             /* descripteur de fichier */
    int flags;          /* FILE_ERR ou FILE_EOF */
    char *buffer        /* tampon */
    int buf_ptr;        /* position courante dans le tampon */
    int buf_end;        /* fin des données du tampon */
} FILE;
```

Le champ `fd` contient le descripteur de fichier associé à ce flot. Le champ `buffer` contient un pointeur sur un tampon de taille `BUFSIZ`. Le champ `buf_ptr` indique la position courante dans le tampon, et le champ `buf_end` indique la fin du tampon.

Le champ `flags` indique l'état du flot. Il vaut normalement 0 ; il peut valoir `FILE_ERR` si une erreur a eu lieu sur ce flot, et `FILE_EOF` si la fin du fichier a été atteinte. Dans une implémentation plus complète de `stdio`, le champ `flags` contiendra aussi des informations sur la politique de gestion du tampon associé au flot (voir partie 2.4.2 ci-dessous).

Une structure `FILE` est normalement créée par la fonction `fopen` :

```
FILE *fopen(const char *path, const char *mode);
```

Cette fonction ouvre le fichier nommé par `path` et construit une structure `FILE` associée au descripteur de fichier résultant.

Le paramètre `mode` indique le but de l'ouverture de fichier. Il s'agit d'une chaîne de caractères dont le premier élément vaut « `r` » pour une ouverture en lecture, « `w` » pour une ouverture en écriture, et « `a` » pour une ouverture en mode ajout. Ce caractère peut être suivi de « `+` » pour une ouverture en lecture et écriture simultanées, et « `b` » si le fichier ouvert est un fichier binaire plutôt qu'un fichier texte (cette dernière information est ignorée sur les systèmes POSIX).

Une structure `FILE` est détruite à l'aide de la fonction `fclose` :

```
int fclose(FILE *file);
```

Cette fonction ferme le descripteur de fichier, libère le tampon, puis libère la structure `FILE` elle-même.

2.4.2 Entrées/sorties de haut niveau

Les entrées/sorties `stdio` utilisent toujours un tampon ; cependant, plusieurs politiques différentes sont possibles pour décider quand ce tampon est vidé, i.e. quand l'appel système effectuant la sortie effective est effectué. Le tampon peut-être vidé seulement lorsqu'il est plein (c'est ce qui se passe lorsqu'un `FILE` est associé à un fichier), à la fin de chaque ligne (c'est ce qui se passe lorsqu'un `FILE` est associé à un terminal), ou après chaque appel de fonction d'entrée/sortie de haut niveau. La politique de gestion du tampon associé à un `FILE` peut être changée à l'aide de la fonction `setvbuf`.

Les fonctions d'entrée/sortie fondamentales sont `getc`, qui retourne un caractère lu sur un flot, et `putc`, qui écrit un caractère sur un flot. La fonction `fflush` permet de vider le tampon. En ignorant les complications liées aux fichiers ouverts en entrée et en sortie simultanément ainsi que celles liées aux tampons « par ligne », les fonctions `getc` et `putc` pourraient être définies comme suit :

```
int getc(FILE *f)
{
    if(f->buf_ptr >= f->read_end)
        _filbuf(f);
    if(f->flags & (FILE_ERR | FILE_EOF))
        return -1;
    return (unsigned char)fp->buf[fp->buf_ptr++];
}

int putc(char ch, FILE *f)
{
    if(f->buf_ptr >= BUFSIZ)
        _flshbuf(f);
    if(f->flags & FILE_ERR)
        return -1;
    fp->buf[fp->buf_ptr++] = ch;
    return ch;
}
```

La fonction `_filbuf` remplit le tampon en faisant une lecture depuis `f->fd` ; inversément, la fonction `_flshbuf` vide le tampon en faisant une ou plusieurs écritures. Ces fonctions sont analogues aux programmes vus dans la partie 2.3.

Comme `stdio` utilise des tampons, il est possible de faire toutes les entrées/sorties avec ces fonctions de façon relativement efficace. Cependant, `stdio` fournit aussi des fonctions d'entrée/sortie « par lots », `fread` et `fwrite`, semblables à `read` et `write`. Comme elles sont prévues pour fonctionner sur des systèmes où les fichiers ne sont pas forcément des suites d'octets, ces fonctions opèrent sur des tableaux d'éléments de taille quelconque :

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *file);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *file);
```

Le paramètre `nmemb` indique le nombre d'éléments à lire ou à écrire, tandis que `size` indique la taille de chaque élément ; le nombre total d'octets lus ou écrits est donc `nmemb` × `size`.

2.4.3 Interaction entre `stdio` et le système

Au démarrage du programme, `stdio` crée trois flots associés aux descripteurs de fichier 0, 1 et 2, et les affecte aux variables globales `stdin`, `stdout` et `stderr` respectivement.

La fonction `fdopen`, une extension POSIX à `stdio`, permet de créer un `FILE` à partir d'un descripteur de fichier existant :

```
FILE *fdopen(int fd, const char *mode);
```

Cette fonction est notamment utile lorsque `fd` n'a pas obtenu à l'aide d'`open`, mais par un autre appel système (par exemple `pipe` ou `socket`).

Inversément, la fonction `fileno` permet d'obtenir le descripteur de fichier associé à un `FILE` :

```
int fileno(FILE *file);
```

Enfin, la fonction `fflush` permet de vider le tampon associé à un `FILE` :

```
int fflush(FILE *file);
```

Appelée avec un pointeur nul, cette fonction vide les tampons de tous les `FILE` du processus, ce qui est utile notamment juste avant un appel à `fork`.

Comme `stdio` utilise des tampons, il n'est pas facile d'utiliser simultanément les appels système et les fonctions `stdio` sur le même descripteur de fichier. Dans le cas des sorties, il faut utiliser la fonction `fflush`, qui vide le tampon associé à un `FILE`, avant chaque appel système qui manipule le descripteur directement ; dans le cas des entrées, il faut faire un appel à `fseek` à chaque transition entre les appels système et les fonctions `stdio`.

2.4.4 Entrées/sorties formatées

Pour présenter des données à l'utilisateur, il faut les *formater*, c'est à dire en construire une *représentation textuelle*, une suite de caractères qui correspond à une présentation culturellement acceptée de ces données. Inversément, les données entrées par l'utilisateur doivent être *analysées*.

Formatage et analyse La suite `stdio` contient la fonction `snprintf` qui effectue le formatage des données :

```
int snprintf(char *buf, size_t size, const char *format, ...);
```

Cette fonction prend en paramètre un tampon `buf` de taille `size`, et y stocke une représentation guidée par `format` des paramètres optionnels qui suivent [4].

Si le formatage est réussi, `snprintf` retourne le nombre de caractères stockés dans `buf` (sans compter le `'\0'` final). Lorsque `buf` est trop court, le comportement dépend de la version : dans les implémentations C89, elle retourne -1, dans les implémentations C99 et C11, elle retourne la taille du tampon dont elle aurait eu besoin pour ne pas tronquer sa sortie. Cette différence entre les versions doit être gérée par le programmeur, comme dans l'exemple de la figure 2.7.

Inversement, la fonction `sscanf` effectue l'analyse :

```
int sscanf(const char *str, const char *format, ...);
```

Entrées/sorties formatées Les fonctions de formatage/analyse et les fonctions d'entrées/sorties sont combinées en des fonctions d'entrées/sorties formatées. Ainsi, la fonction `fprintf` formate ses arguments avant de les écrire sur un flot, et `fscanf` lit des données qu'elle analyse ensuite. Les fonctions `printf` et `scanf` sont des abbréviations pour `fprintf` et `fscanf` dans le cas où l'argument `file` vaut `stdout` et `stdin` respectivement.

4. La fonction semblable `sprintf` est à éviter, car elle peut déborder du tampon qui lui est passé.

```
char *format_integer(int i)
{
    char *buf;
    int n = 4, rc;
    while(1) {
        buf = malloc(n);
        if(buf == NULL)
            return NULL;
        rc = snprintf(buf, n, "%d", i);
        if(rc >= 0 && rc < n)
            return buf;
        free(buf);
        if(rc >= 0)
            n = rc + 1;
        else
            n = 2 * n;
    }
}
```

FIGURE 2.7 — Formatage portable entre C89 et C99

2.5 Le pointeur de position courante

Les lectures et les écritures dans un fichier sont par défaut *séquentielles* : les données sont lues ou écrites les unes à la suite de l'autre. Par exemple, lors de l'exécution de la séquence de code

```
rc = write(fd, "a", 1);
rc = write(fd, "b", 1);
```

l'octet « `b` » est écrit après l'octet « `a` ».

La position dans un fichier à laquelle se fait la prochaine lecture ou écriture est stockée dans le noyau dans un champ de l'entrée de fichier ouvert qui s'appelle le *pointeur de position courante* (figure 2.3). Lors de l'appel système `open`, le pointeur de position courante est initialisé à 0, soit le début du fichier.

2.5.1 L'appel système `lseek`

Le pointeur de position courante associé à un descripteur de fichier peut être lu et modifié à l'aide de l'appel système `lseek` :

```
off_t lseek(int fd, off_t offset, int whence);
```

Le paramètre `fd` est le descripteur de fichier dont l'entrée de la table de fichiers ouverts associée doit être modifiée. Le paramètre `offset` (« déplacement ») identifie la nouvelle position, et le paramètre `whence` (« à partir d'où ») spécifie l'interprétation de ce dernier. Il peut avoir les valeurs suivantes :

- `SEEK_SET` : `offset` spécifie un décalage à partir du début du fichier ;
- `SEEK_CUR` : `offset` spécifie un décalage à partir de la position courante ;
- `SEEK_END` : `offset` spécifie un décalage à partir de la fin du fichier.

L'appel `lseek` retourne la nouvelle valeur du pointeur de position courante, ou -1 en cas d'erreur (et alors `errno` est positionné).

> **Attention :** le type `off_t` est d'habitude un synonyme de `long` ou même `long long` ; attention donc au type des variables.

Exemples Pour déplacer le pointeur de fichier au début d'un fichier, on peut faire

```
lrc = lseek(fd, 0L, SEEK_SET);
```

Pour déplacer le pointeur de fichier à la fin d'un fichier, on peut faire

```
size = lseek(fd, 0L, SEEK_END);
```

Si l'appel réussit, la variable `size` contient la taille du fichier.

On peut déterminer la position courante à l'aide de l'appel

```
position = lseek(fd, 0L, SEEK_CUR);
```

2.5.2 Le mode « ajout à la fin »

Il est très courant de vouloir ajouter des données à la fin d'un fichier. En première approximation, ce n'est pas difficile :

```
fd = open("/var/log/messages", O_WRONLY);
lrc = lseek(fd, 0L, SEEK_END);
rc = write(fd, ...);          /* condition critique ! */
close(fd);
```

Malheureusement, une telle approche mène à une condition critique *(race condition)* qui peut causer une perte de données. Considérons en effet se qui peut se passer si deux processus veulent simultanément ajouter des données à la fin du même fichier :

```
/* processus A */              /* processus B */
lseek(fd, 0L, SEEK_END);
                               lseek(fd, 0L, SEEK_END)
                               write(fd, "b", 1)
write(fd, "a", 1)
```

Supposons, pour fixer les idées, que le fichier a initialement une taille de 1000 octets. Le processus A commence par se positionner à la fin du fichier, soit à la position 1000. Supposons maintenant qu'un changement de contexte intervient à ce moment, le contrôle passe au processus B, qui se positionne à la position 1000, et y écrit un octet « `b` ». Le contrôle repasse ensuite au processus A, qui est toujours positionné en 1000 ; il écrit donc un octet « `a` » à la position 1000, et écrase donc les données écrites par le processus B.

Une solution entièrement générale au problème de l'accès à des ressources partagées demande que les processus A et B effectuent une synchronisation, par exemple en posant des verrous sur la ressource commune ou en utilisant un troisième processus qui sert d'arbitre. Unix inclut cependant une solution simple pour le cas particulier de l'écriture à la fin du fichier.

Lorsque le *flag* `O_APPEND` est positionné dans le deuxième paramètre de `open`, le fichier est ouvert en mode *écriture à la fin (append mode)*. Le noyau repositionne alors le pointeur de position courante à la fin du fichier lors de toute opération d'écriture effectuée à travers ce pointeur de fichier. L'opération ayant entièrement lieu dans le noyau, celui-ci s'assure de l'exécution atomique du positionnement et de l'écriture.

2.6 Troncation et extension des fichiers

Lorsqu'un processus écrit à la fin du fichier, la taille de ce fichier augmente ; on dit alors que le fichier est *étendu*. L'opération inverse à l'extension s'appelle la *troncation*.

2.6.1 Extension implicite

Toute écriture au delà de la fin d'un fichier cause une extension de celui-ci. Par exemple, la séquence de code suivante provoque une extension d'un octet :

```
lrc = lseek(fd, 0L, SEEK_END);
rc = write(fd, "a", 1);
```

Une extension plus importante peut être réalisée en se positionnant au-delà de la fin du fichier et en écrivant des données ; les parties du fichier qui n'ont jamais été écrites sont alors automatiquement remplies de 0 par le noyau. Par exemple, la séquence suivante ajoute à la fin du fichier 1 Mo de zéros suivi d'un octet « `a` » :

```
lrc = lseek(fd, 1024 * 1024L, SEEK_END);
rc = write(fd, "a", 1);
```

Parenthèse : fichiers à trous En fait, les données non écrites ne sont pas forcément stockées sur disque : le noyau note simplement que la plage est « pleine de zéros », et produira des zéros lors d'une prochaine lecture. Ceci peut être constaté en consultant le champ `st_blocks` de l'i-nœud, par exemple à l'aide de la commande « `ls -s` ».

2.6.2 Troncation et extension explicite

Une troncation ou extension explicite peut se faire à l'aide des appels système `truncate` et `ftruncate`[5] :

```
int truncate(char *name, off_t length);
int ftruncate(int fd, off_t length);
```

5. Traditionnellement, ces appels servent à faire une troncation ; sur les Unix modernes, ils permettent aussi de faire une extension.

Comme beaucoup d'appels système agissant sur les fichiers, cet appel système existe en deux variantes : pour `truncate`, le fichier à tronquer ou étendre est spécifié à l'aide de son nom `name`, tandis que pour `ftruncate`, il est spécifié à l'aide d'un descripteur de fichier `fd`. Le paramètre `length` spécifie la nouvelle taille du fichier. S'il est inférieur à la taille actuelle, le fichier est tronqué, et des données sont perdues ; s'il y est supérieur, le fichier est étendu, et des zéros sont écrits (mais voyez le commentaire ci-dessus sur les fichiers à trous).

Ces appels système ne changent pas le pointeur de position courante, et retournent 0 en cas de succès, -1 en cas d'erreur (et alors `errno`… bon, vous avez compris).

`O_TRUNC` Il est très courant de vouloir tronquer un fichier à 0 juste après l'avoir ouvert. Pour éviter de faire un appel à `ftruncate` après chaque ouverture, la troncation peut être combinée à cette dernière en positionnant le drapeau `O_TRUNC` dans le deuxième paramètre de `open` :

```
fd = open("alamakota", O_WRONLY | O_CREAT | O_TRUNC, 0666);
```

À la différence des drapeaux `O_EXCL` et `O_APPEND`, qui sont essentiels pour éviter des situations critiques dans certains cas, le drapeau `O_TRUNC` n'est qu'une abréviation.

2.7 I-nœuds

L'i-nœud est une des structures fondamentales du système de fichiers Unix ; conceptuellement, l'i-nœud « c'est » le fichier.

Structure d'un i-nœud sur disque Un i-nœud sur disque contient, entre autres :

- le type du fichier qu'il représente (fichier ordinaire ou répertoire) ;
- le nombre de liens sur l'i-nœud ;
- une indication du propriétaire du fichier ;
- la taille des données du fichier ;
- les temps de dernier accès et de dernière modification des données du fichier, et le temps de dernière modification de l'i-nœud ;
- une structure de données (typiquement un arbre) qui permet de retrouver les données du fichier.

Structure d'un i-nœud en mémoire Lorsqu'il est chargé en mémoire (figure 2.3), un i-nœud contient en outre les données nécessaires pour retrouver l'i-nœud sur disque :

- le numéro du périphérique dont il provient ;
- le numéro d'i-nœud sur le périphérique.

2.7.1 Lecture des i-nœuds

Un i-nœud contient un certain nombre de données qui peuvent intéresser le programmeur. L'appel système `stat` et son cousin `fstat` permettent de les consulter. Ces appels système ont les prototypes suivants :

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

Ils différent par la façon de spécifier l'i-nœud à consulter : `stat` y accède par un nom de fichier, tandis que `fstat` utilise un descripteur de fichier [6]. Ils remplissent le tampon `buf` avec les informations de l'i-nœud, puis retournent 0 en cas de succès, -1 en cas d'échec (et alors la variable `errno` est positionnée).

La structure stat

La structure `stat` retournée par `stat` et `fstat` contient au moins les champs suivants :

```
struct stat {
    dev_t st_dev
    ino_t st_ino
    mode_t st_mode
    nlink_t st_nlink
    uid_t st_uid
    gid_t st_gid
    off_t st_size
    time_t st_atime
    time_t st_mtime
    time_t st_ctime
    blkcnt_t st_blocks
};
```

Les types opaques ci-dessus (`dev_t`, `ino_t`, etc.) sont souvent des synonymes de `int`, sauf `off_t`, qui est normalement un synonyme de `long` ou même `long long`.

Localisation de l'i-nœud Les champs `st_dev` et `st_ino` contiennent, respectivement, le numéro du périphérique où se trouve l'i-nœud et le numéro de l'i-nœud. Ils ont la propriété qu'à un moment donné la paire (`st_dev`, `st_ino`) identifie l'i-nœud de façon unique sur le système, ce qui permet par exemple de vérifier que deux descripteurs de fichier pointent sur le même fichier. Attention, un numéro d'i-nœud peut être réutilisé après qu'un fichier a été supprimé, ce qui implique qu'il n'est pas possible de se servir de ces champs pour identifier des fichiers à des moments différents.

Comptage des références Le champ `st_nlinks` contient le compte des références à l'i-nœud. Nous en parlons en plus de détail à la partie 2.7.4.

Type et permissions Le champ `st_mode` contient 16 bits qui servent à identifier le type de fichier et en décrire les permissions d'accès.

Les 12 bits d'ordre bas de `st_mode` décrivent les permissions.

6. Unix ne permet pas d'accéder directement à un i-nœud en mémoire.

Les 4 bits d'ordre haut décrivent le type de fichier. Les valeurs possibles pour ces bits incluent (en octal) :

- 0100000 ou `S_IFREG` : fichier ordinaire ;
- 0040000 ou `S_IFDIR` : répertoire ;

Outre les macros `S_IF...`, le fichier d'entête `<sys/stat.h>` définit des macros nommées `S_ISREG` et `S_ISDIR` qui permettent de tester ces valeurs.

Propriétaire Les champs `st_uid` et `st_gid` définissent le propriétaire du fichier. Il s'agit normalement des champs `euid` et `egid` du processus qui a créé l'i-nœud.

Taille Le champ `st_size` contient la taille du (contenu du) fichier, comptée en octets.

Temps Les champs `st_atime`, `st_mtime` et `st_ctime` contiennent les temps du fichier, en secondes depuis l'Époque (même unités que l'appel système `time` vu auparavant). Ils ont la sémantique suivante :

- `st_atime` est le *temps du dernier accès* : il est mis à jour à chaque fois que des données du fichier sont lues (par exemple par l'appel système `read`) [7] ;
- `st_mtime` est le *temps de dernière modification* : il est mis à jour à chaque fois que des données sont écrites dans le fichier ;
- `st_ctime` est le *temps de dernière modification du i-nœud* : il est mis à jour à chaque fois que le i-nœud est modifié (par exemple par les appels système du paragraphe 2.7.2 ci-dessous). Ce champ est parfois appelé *temps de création du fichier*.

Normalement, c'est `st_mtime` qui est intéressant ; l'auteur avoue qu'il ne s'est jamais servi des deux autres temps.

Les fonctions `localtime` et `strftime` permettent de convertir le temps Unix en temps local et de formater le résultat.

Taille des données Le champ `st_blocks` indique la place occupée sur disque par les données. L'unité dans laquelle est exprimée cette valeur n'est pas définie par POSIX (elle est de 512 octets sous la 7e édition, sous Unix BSD et sous Linux, mais elle vaut 1024 octets sur certains Unix Système V). Bien qu'il ne soit pas fiable sur les systèmes de fichiers modernes, ce champ permet parfois de détecter les fichiers à trous.

2.7.2 Modification des i-nœuds

Un i-nœud peut être modifié *implicitement*, par un appel système qui manipule les données du disque ou la structure de répertoires, ou *explicitement* par un appel système dont le rôle principal est de modifier l'i-nœud.

7. Les mises à jour du temps d'accès posent de sérieux problèmes de performances, et de ce fait certains systèmes ne les implémentent pas ; il est donc déconseillé de s'en servir.

Modification implicite

D'habitude, les champs d'un i-nœud sont modifiés implicitement, par des appels système qui modifient le contenu du fichier. Par exemple, toute extension ou troncation d'un fichier modifie les champs `st_size` et `st_mtime` d'un fichier, et toute écriture modifie `st_mtime`. De même, toute lecture modifie `st_atime` (s'il est implémenté). Toute création ou suppression de liens modifie `st_nlinks`. Toute création ou suppression de liens, et tout changement de propriétaire ou de permissions modifie le champ `st_ctime`.

Modification explicite

Il est aussi possible de modifier les champs d'un i-nœud explicitement à l'aide d'appels système dédiés à cela.

Changement de permissions Les permissions d'un fichier (les 12 bits bas du « mode » de l'i-nœud) peuvent être modifiés à l'aide des deux appels système suivants :

```
int chmod(char *path, int mode);
int fchmod(int fd, int mode);
```

Comme dans le cas de `stat` et `fstat`, `chmod` et `fchmod` diffèrent dans la façon dont est identifié l'i-nœud à modifier : `chmod` prend un nom de fichier, `fchmod` un descripteur. Ces appels retournent 0 en cas de succès, -1 en cas d'erreur.

Changement de propriétaire Le propriétaire d'un fichier peut être changé avec les appels système suivants :

```
int chown(char *path, int uid, int gid);
int fchown(int fd, int uid, int gid);
```

Sur tous les systèmes, l'utilisateur `root` (euid = 0) a le droit de changer le propriétaire de n'importe quel fichier. Sous Système V, un utilisateur a en outre le droit de « donner » un fichier qui lui appartient à un autre utilisateur ; sous Unix BSD et Linux, ceci n'est pas autorisé pour éviter de contourner le système de quotas. Dans certaines circonstances, un utilisateur a aussi le droit de changer le groupe d'un fichier qui lui appartient.

Changement de temps Les temps associés à un fichier peuvent être changés à l'aide de l'appel système `utime` :

```
int utime(char *filename, struct utimbuf *buf);
```

La structure `utimbuf` est définie comme suit :

```
struct utimbuf {
    int actime;
    int modtime;
};
```

Le champ `actime` contient le nouveau temps de dernier accès, et le champ `modtime` contient le nouveau temps de dernière modification.

2.7.3 Liens et renommage de fichiers

Comme nous l'avons vu, un lien à un i-nœud est normalement créé par un appel à `open` avec le *flag* `O_CREAT` positionné.

Création de liens supplémentaires

L'appel système `link` crée un nouveau nom et un nouveau lien vers un i-nœud existant. Cet appel permet donc de faire apparaître le même fichier à deux endroits de la hiérarchie de fichiers, ou sous deux noms différents. L'appel `link` a le prototype suivant :

```
int link(char *old, char *new);
```

où `oldpath` est le nom de fichier existant, et `new` le nom créé. Si `new` existe déjà, l'appel `link` échoue avec `errno` valant `EEXIST`.

Suppression de liens

L'appel `unlink` supprime un nom de fichier et le lien associé; s'il s'agissait du dernier lien pointant sur un i-nœud, le fichier lui-même (i-nœud et données) est supprimé aussi. Cet appel a le prototype suivant :

```
int unlink(char *filename);
```

Renommage et déplacement de fichiers

Sous la 7e édition, un fichier était renommé ou déplacé en créant un nouveau lien puis en supprimant l'ancien :

```
int old_rename(char *old, char *new) {
    int rc;
    unlink(new); /* pas de gestion d'erreurs */
    rc = link(old, new); if(rc < 0) return -1;
    rc = unlink(old); if(rc < 0) return -1;
    return 0;
}
```

Cette approche avait un défaut sérieux : comme `old_rename` est composé de plusieurs actions dont chacune peut échouer, il est possible de se retrouver après une erreur (par exemple de permissions) avec un renommage partiellement effectué, par exemple où `new` n'existe plus tandis que `old` n'a pas été renommé. De plus, comme `link` n'est pas autorisé sur les répertoires pour un utilisateur ordinaire, seul `root` pouvait renommer les répertoires.

Unix implémente de nos jours un appel système `rename` :

```
int rename(char *old, char *new);
```

Cet appel système ne peut pas échouer partiellement — après son exécution, soit le fichier a été renommé, soit rien n'a été fait. À la différence de `link`, il supprime le nouveau nom s'il existait déjà.

2.7.4 Comptage des références

Lorsque le dernier lien vers un i-nœud est supprimé, l'i-nœud lui-même et le contenu du fichier sont libérés, et l'espace peut être reutilisé pour d'autres fichiers. Intuitivement, l'appel système `unlink` sert à supprimer les fichiers.

Pour ce faire, le système maintient un champ entier `st_nlink` dans chaque i-nœud, le *compte de références* de ce dernier, qui est égal au nombre de liens existant vers cet i-nœud. Ce champ vaut initialement 1, il est incrémenté à chaque appel `link`, et décrémenté à chaque appel `unlink`. Lorsqu'il atteint 0, l'i-nœud et son contenu sont supprimés.

Le comptage de références, qui est une des techniques classiques de gestion de la mémoire, ne fonctionne pas en présence de cycles dans le graphe de références. Unix évite la présence de cycles dans le système de fichiers en interdisant la création de liens vers des répertoires.

Read after delete La suppression d'un i-nœud est retardée lorsqu'il existe un processus qui l'a ouvert, ou, en d'autres termes, lorsqu'il en existe une copie en mémoire. Un processus peut donc lire ou écrire sur un fichier vers lequel il n'existe plus aucun lien. Intuitivement, on peut lire un fichier après l'avoir supprimé.

Cette sémantique dite *read after delete* évite les conditions critiques entre suppression et lecture. De plus, elle est pratique lors de la gestion de fichiers temporaires, *i.e.* de fichiers qui doivent être supprimés lorsqu'un processus termine.

2.8 Manipulation de répertoires

Un *répertoire* (*directory* en Anglais) [8] est un fichier dont le contenu est une suite d'*entrées de répertoire*, chaque entrée étant composée d'un nom et d'un numéro d'i-nœud. Les répertoires sont identifiés par une valeur spécifique du champ *mode* de leur i-nœud, et le système les traite spécialement : il ne permet pas à l'utilisateur de les modifier directement, et les consulte lors de la résolution de noms.

2.8.1 Résolution de noms

La *résolution de noms* est le processus qui permet de convertir un nom de fichier en un i-nœud. La résolution de noms est effectuée lors de tout appel système qui prend un nom de fichier en paramètre, par exemple `open` ou `stat`.

Chaque système de fichiers a un i-nœud distingué, appelé son *i-nœud racine (root)*, qui contient un répertoire, appelé le *répertoire racine*. La résolution d'un nom de fichier absolu (un nom qui commence par un *slash* « / ») commence au répertoire racine du système de fichiers racine. Le système recherche le premier composant du nom de fichier dans le répertoire racine. S'il est trouvé, et s'il pointe sur un répertoire, le deuxième composant est recherché dans celui-ci. La résolution se poursuit ainsi jusqu'à épuisement du nom de fichier.

8. Le terme *folder* (dossier) a, à ma connaissance, été introduit par Mac OS (classique). Microsoft l'a adopté avec la sortie de Windows 95. Il n'est normalement pas utilisé sous Unix.

La résolution d'un nom *relatif* (un nom qui ne commence pas par un *slash*) se fait de façon analogue, mais en commençant à un i-nœud qui dépend du processus — le *répertoire courant* (voir paragraphe 3.1.5).

2.8.2 Lecture des répertoires

Il est parfois nécessaire de lire explicitement le contenu d'un nom de répertoire. Par exemple, le programme `ls` lit le contenu d'un répertoire et affiche les noms de fichier qu'il y trouve.

Les répertoires en 7e édition

Sous Unix 7e édition, les noms de fichier étaient limités à 14 caractères. Un répertoire était simplement une suite de structures de type `dirent` :

```
struct dirent {
    unsigned short d_ino;
    char d_name[14];
};
```

Un programme qui désirait lire le contenu d'un répertoire procédait de la façon habituelle : après un appel à `open`, il lisait simplement le contenu du répertoire à l'aide de `read` ; chaque suite de 16 octets lus était interprétée comme une structure de type `dirent`.

Interface de haut niveau

Sous 4.2BSD, la structure des répertoires est devenue plus compliquée : un répertoire pouvait contenir des noms de fichiers de longueur arbitraire, et il n'était plus possible de simplement interpréter un répertoire comme une suite d'enregistrements de 16 octets chacun. Afin d'éviter au programmeur d'analyser lui-même une structure de données complexe, une interface de haut niveau, analogue à `stdio`, a été définie.

L'incarnation moderne de cette interface s'appelle `dirent`. Elle définit deux structures de données : `DIR` (analogue à `FILE`) est une structure de données opaque représentant un répertoire ouvert. La structure `dirent` représente une entrée de répertoire ; elle contient au moins les champs suivants :

```
struct dirent {
    ino_t d_ino;
    char d_name[];
};
```

Comme dans la version traditionnelle (7e édition), le champ `d_ino` contient un numéro d'i-nœud, et le champ `d_name` le nom d'un fichier. À la différence de celle-ci, cependant, la longueur du champ `d_name` n'est pas spécifiée — il est terminé par un `\0`, comme une chaîne normale en C.

Les répertoires sont manipulés à l'aide de trois fonctions :

```
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
int closedir(DIR *dir);
```

La fonction `opendir` ouvre un répertoire et retourne un pointeur sur une structure de type `DIR` représentant le répertoire ouvert.

La fonction `readdir` retourne un pointeur sur l'entrée de répertoire suivante. Celui-ci peut pointer sur les tampons internes au `DIR` manipulé, ou même sur un tampon statique ; son contenu cesse donc d'être valable dès le prochain appel à `readdir`, et il faut en faire une copie s'il doit être utilisé par la suite.

Enfin, la fonction `closedir` ferme le répertoire et détruit la structure `DIR`.

2.9 Liens symboliques

Les liens (dits « durs », *hard links*), tels qu'on les as vus au paragraphe 2.7.3, sont un concept propre, élégant et sans ambiguïté. Cependant, ils ont certaines limitations qui les rendent parfois peu pratiques :

- il est impossible de faire un lien entre deux systèmes de fichiers distincts ;
- il est impossible (en pratique) de créer un lien vers un répertoire.

Les *liens symboliques* (4.2BSD) ont été développés pour éviter ces limitations. Il s'agit d'un concept peu élégant [9] et ayant une sémantique souvent douteuse, mais fort utile en pratique.

À la différence d'un lien (dur), un lien symbolique ne réfère pas à un i-nœud, mais à un nom. Lorsque le système rencontre un lien symbolique lors de la résolution d'un nom de fichier, il remplace le nom du lien par son contenu, et recommence la résolution. Par exemple, si `/usr/local` est un lien symbolique contenant `/mnt/disk2/local`, le nom de fichier

```
/usr/local/src/hello.c
```

est remplacé par

```
/mnt/disk2/local/src/hello.c
```

De façon analogue, si `/usr/share` est un lien symbolique contenant `../lib`, le nom de fichier

```
/usr/share/doc
```

est remplacé par

```
/usr/share/../lib/doc
```

Que se passe-t-il lorsqu'un lien symbolique contenant un nom relatif est déplacé ? Lorsque la cible d'un lien symbolique est renommée ?

9. Moins horrible, toutefois, que les *raccourcis* utilisés sur certains autres systèmes.

Création d'un lien symbolique Un lien symbolique est créé par l'appel système `symlink` :

```
int symlink(const char *contents, const char *name);
```

Un appel à `symlink` construit un lien symbolique nommé `name` dont le contenu est la chaîne contenue dans `contents`.

Utilisation des liens symboliques Dans la plupart des cas, l'utilisation des liens symboliques se fait de façon transparente lorsqu'un appel système effectue une résolution de noms. Par exemple, un appel à `open` sur le nom d'un lien symbolique s'appliquera automatiquement à la cible du lien.

Certains appels système ne traversent pas les liens symboliques. C'est notamment le cas de `unlink`, qui détruit le lien, et pas sa cible. L'appel système `stat` existe en deux variantes : `stat`, qui suit les liens, et `lstat`, qui ne le fait pas et permet donc d'obtenir des informations sur le lien lui-même.

On peut lire le contenu d'un lien symbolique à l'aide de l'appel système `readlink` :

```
ssize_t readlink(const char *path, char *buf, size_t bufsiz);
```

3 Processus et communication

3.1 Programmes et processus

Un *programme* est un fichier contenant du code pouvant être exécuté. Par exemple, le fichier `a.out` généré par le compilateur est un programme, comme le fichier `/bin/ls`. Lorsqu'il est *exécuté*, le programme est chargé en mémoire dans une structure de données qui s'appelle un *processus*. Il y a à tout moment un seul programme qui s'appelle `emacs` sur mon portable, mais il peut y avoir zéro, un ou plusieurs processus qui exécutent Emacs.

3.1.1 Protection mémoire

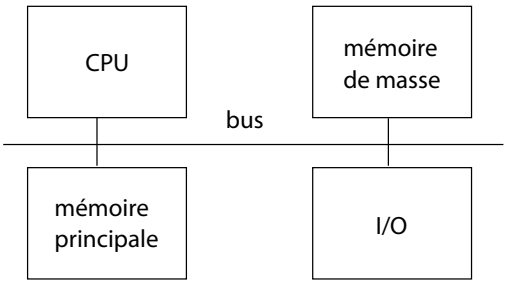


FIGURE 3.1 — Ordinateur sans PMU

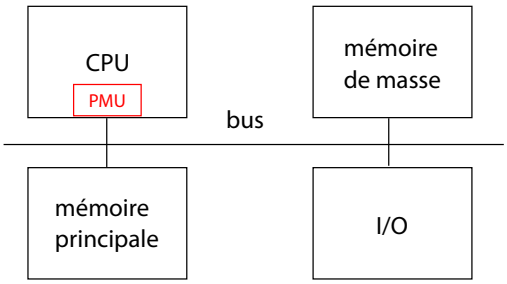


FIGURE 3.2 — Ordinateur avec PMU

D'une altitude de 10 000 pieds, un ordinateur ressemble à la figure 3.1. Le *processeur (CPU)*, qui exécute les instructions du programme utilisateur, est interconnecté à travers un *bus* à la mémoire principale, qui sert à stocker le programme à exécuter et les données qu'il manipule, ainsi qu'aux périphériques de stockage de masse (disque, SSD, etc.) et aux périphériques d'entrées/sorties (clavier, écran, interface réseau). Lorsque le processeur a besoin d'effectuer une lecture depuis la mémoire principale (obtenir une instruction ou une donnée) ou une écriture vers celle-ci, il envoie un message sur le bus et la mémoire y répond [1].

À part les plus petits systèmes embarqués, les processeurs modernes implémentent un mécanisme de *protection de la mémoire*. Entre le processeur et le bus est intercalé une *unité de protection mémoire (PMU)* (figure 3.2). À chaque accès à la mémoire, le PMU consulte la carte mémoire dont il dispose, et décide si l'accès est autorisé ; si ce n'est pas le cas, il génère une *exception matérielle* qui a pour effet de rendre la main au noyau du système d'exploitation. Celui-ci identifie le pro-

1. On peut donc voir l'ordinateur comme un réseau. On a d'ailleurs en programmation plusieurs des mêmes problèmes qu'en programmation réseau, notamment celui de cacher la latence du bus — nous en parlerons davantage lors du cours sur la hiérarchie mémoire.

cessus coupable, et le tue (sous Unix, avec un signal `SIGSEGV`). Le PMU permet donc au noyau de protéger des régions de mémoire en y interdisant les lectures, les écritures, ou les deux.

3.1.2 Structure d'un programme

Les systèmes contemporains utilisent des formats de fichier compliqués pour les programmes exécutables, *ELF* sous Unix ou *COFF* sous Windows. Les Unix plus anciens utilisaient un format appelé *a.out* (pour *Assembler Output*) [2].

Le format *a.out* a changé plusieurs fois au cours de son histoire, dans ce paragraphe je décris la variante utilisée par 2.11BSD (la dernière version d'Unix BSD qui pouvait s'exécuter sur un PDP-11). Le fichier commence par un entête qui a la forme suivante :

```
struct exec {
    int a_magic;
    unsigned int a_text;
    unsigned int a_data;
    unsigned int a_bss;
    unsigned int a_syms;
    unsigned int a_entry;
    unsigned int a_unused;
    unsigned int a_flag;
};
```

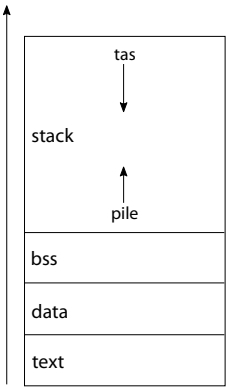
Comme les types `int` et `unsigned int` ont une taille de 16 bits sur un PDP-11, l'entête a une taille de 16 octets. Le champ `a_magic` contient un nombre magique qui permet d'identifier un fichier *a.out*. Le champ `a_text` contient la longueur de la section `text` ; le champ `a_data` contient la longueur de la section `data`, et enfin le champ `a_bss` contient la longueur de la section `bss`. Les détails des autres champs ne nous intéressent pas dans ce cours.

L'entête est suivi du contenu de la section `text`, de longueur `a_text`, qui contient le code du programme, et ensuite du contenu de la section `a_data`, qui contient les valeurs des variables globales initialisées. La section `bss`, qui contient les variables globales non-initialisées, n'est pas stockée dans le programme — elle est initialisée à 0 lors du chargement du programme.

3.1.3 Carte mémoire d'un processus

Lorsqu'un programme est exécuté sous 2.11BSD, le système crée un espace de mémoire qu'il peuple de la façon suivante :

2. Du coup, un fichier appelé `a.out` est le plus souvent un fichier ELF de nos jours.



- la section `text` est chargée à l'adresse 0, et protégée en écriture — seules les lectures et l'exécution sont autorisées ;
- la section `data` est chargée après la section `text`, à la première adresse multiple de la taille de page (8192 sur un PDP-11, 4096 sur le matériel dont vous disposez) ; les écritures et les lectures y sont autorisées ;
- la section `bss` est placée après la section `data`, et son contenu est initialisé à 0 ;
- la section `bss` est suivie d'une section `stack`, qui contient la pile d'appels de fonction et le *tas (heap)* pour les allocations effectuées à l'aide de `malloc`.

Quelques améliorations ont été apportées à ce schéma après 2.11BSD. La principale est que la section `text` n'est plus chargée à l'adresse 0 — la page 0 est protégées en lecture et en écriture, ce qui permet de détecter les déréférencements du pointeur nul.

Virtualisation de la mémoire Les adresses de mémoire manipulées par le code utilisateur ne correspondent pas forcément à des adresses physiques : la même adresse mémoire dans deux processus peut correspondre à deux locations physiques distinctes, et une adresse peut même ne pas correspondre à de la mémoire du tout, mais être simulée par des données stockées sur disque. La mémoire est donc *virtualisée*, le système présentant à l'utilisateur une abstraction qui ressemble à de la mémoire physique, mais peut être implémentée de diverses manières. Nous en discutons en un peu plus de détail à la partie 3.5.

3.1.4 L'ordonnanceur

Unix est un système à *temps partagé* : plusieurs processus peuvent être exécutables à un moment donné, le système donnant l'illusion d'une exécution simultanée en passant rapidement d'un processus à l'autre [3]. Il s'agit là d'un autre cas de *virtualisation*, cette fois-ci du processeur.

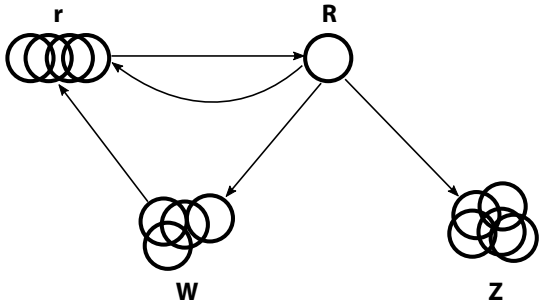


FIGURE 3.3 — Structures de l'ordonnanceur

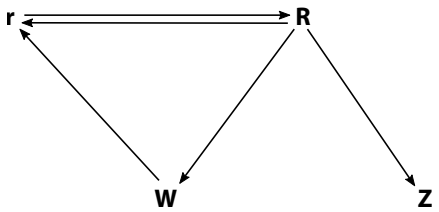


FIGURE 3.4 — Transitions des processus

La partie du noyau qui choisit quel processus doit s'exécuter à un moment donné s'appelle *l'ordonnanceur (scheduler)*. L'ordonnanceur maintient les structures de données suivantes (voir figure 3.3) :

3. Dans toute cette partie, nous faisons l'hypothèse simplificatrice d'un système à un seul processeur.

– le processus en train de s'exécuter (R) ;
– une file de processus prêts à s'exécuter (r) ;
– un ensemble de processus en attente d'un événement [4] (W) ;
– un ensemble de processus morts, les *zombies* (Z).

Un processus peut donc avoir les états suivants (figure 3.4) :

– R, « *Running* » en train de s'exécuter ;
– r, « *runnable* », prêt à s'exécuter ;
– W, « *Waiting* », en attente d'un événement ;
– Z, « *Zombie* », mort.

Les transitions suivantes entre états sont possibles :

– $\varnothing \longrightarrow$ r, création d'un processus ;
– r $\longrightarrow$ R et R $\longrightarrow$ r, changement de contexte *(context switch)* décidé par l'ordonnanceur ;
– R $\longrightarrow$ W, appel système bloquant (voir ci-dessous) ;
– W $\longrightarrow$ r, arrivée d'un événement ;
– R $\longrightarrow$ Z, mort d'un processus.

Appels système bloquants

Lorsqu'un processus est dans l'état « en train de s'exécuter » R, il peut exécuter des instructions du processeur (par exemple pour effectuer des calculs), ou faire des appels système. Parfois, le noyau peut satisfaire l'appel système immédiatement — par exemple, un appel à `time` calcule l'heure et retourne immédiatement au code utilisateur.

Souvent, cependant, un appel système demande une interaction prolongée avec le monde réel ; un processus exécutant un tel appel système est mis en attente d'un événement (état W), et ne sera réveillé (passé à l'état « prêt à s'exécuter » r) que lorsque l'appel système sera prêt à retourner. Un tel appel système est dit *bloquant*.

Considérons par exemple le cas d'un appel système `read` demandant des données stockées sur le disque. Au moment ou le processus effectue l'appel système, le contrôleur de disque est programmé pour lire les données, et le processus appelant est mis en attente de l'événement « requête de lecture terminée ». Quelques millisecondes plus tard (plusieurs millions de cycles du processeur), lorsque la requête aura abouti, le processus sera réveillé et remis dans l'état « prêt à s'exécuter ».

3.1.5 Manipulation des processus

En plus de la mémoire du processus et de son contexte, l'ordonnanceur maintient pour chaque processus une structure de données qui contient en particulier les données suivantes :

– le numéro du processus *pid* ;
– le numéro de son processus père, *ppid* ;
– le propriétaire « réel » du processus, *uid* et *gid*, et le propriétaire « effectif », *euid* et *egid* ;
– le répertoire courant du processus ;
– l'état du processus.

4. En pratique, il s'agit d'un ensemble de files, une par évènement possible, ou même d'une structure plus complexe.

La commande `ps`

La commande « `ps l` » (BSD) ou « `ps -l` » (SYSV) permet de connaître une floppée de données à propos des processus. Par défaut, elle n'affiche que les processus appartenant à l'utilisateur qui l'invoque, et la variante BSD restreint de plus sa sortie aux processus « intéressants » ; pour avoir la liste de tous les processus du système, il faut utiliser la commande « `ps algx` » (BSD) ou « `ps -el` » (SYSV).

Cette commande affiche en particulier :

- le numéro du processus et celui de son père (colonnes *pid* et *ppid*) ;
- le propriétaire réel du processus (colonnes *uid* et *gid*) ;
- l'état du processus (colonne *stat*), et, dans le cas d'un processus bloqué, l'événement sur lequel il est en attente (colonne *wchan*).

Appels système d'accès au processus

Identité du processus Le champ *ppid* organise l'ensemble des processus en une arborescence souvent applée la *hiérarchie* des processus. On dit que le processus p est le *père* de q lorsque $\text{ppid}(q) = p$; q est alors un *fils* de p. Lorsqu'un processus meurt, ses enfants sont « adoptés » par le processus « init », portant le numéro 1.

Le numéro du processus en train de s'exécuter est accessible à l'aide de l'appel `getpid` ; le numéro de son père à l'aide de `getppid` :

```
pid_t getpid(void);
pid_t getppid(void);
```

Propriétaire et privilèges du processus Comme nous l'avons vu dans la partie système de fichiers, sous Unix un principal de sécurité (propriétaire d'un fichier, d'un processus ou d'une autre ressource) est représenté par deux entiers : un *numéro d'utilisateur* et un *numéro de groupe*.

Un processus maintient deux propriétaires [5] : un *propriétaire réel*, qui représente l'utilisateur qui a créé ce processus, et un *propriétaire effectif*, qui code les privilèges dont dispose ce processus, par exemple pour modifier des fichiers.

Le propriétaire d'un processus peut être identifié à l'aide des appels système suivants :

```
uid_t getuid(void);
gid_t getgid(void);
uid_t geteuid(void);
gid_t getegid(void);
```

Les appels système `setuid`, `setgid`, `seteuid` et `setegid` permettent de changer de propriétaire. Leur sémantique est complexe, et ils ne sont pas portables : leur comportement diffère entre BSD et SYSV.

5. En fait, il y en a davantage — voyez `setresuid` pour plus d'informations.

Le répertoire courant Chaque processus contient une référence à un i-nœud distingué, le *répertoire courant (working directory)* du processus, qui sert à la résolution des noms de fichier relatifs (voir paragraphe 2.8.1). La valeur du répertoire courant peut être changée à l'aide des appels système `chdir` et `fchdir` :

```
int chdir(const char *path);
int fchdir(int fd);
```

Le répertoire courant est consulté à chaque fois qu'un nom relatif est passé à un appel système effectuant une résolution de noms. Son contenu peut notamment être lu en passant le nom `"."` à l'appel système `stat` ou à la fonction `opendir`.

Remarquez que le répertoire courant est un i-nœud et pas un nom. Le nom du répertoire courant n'est pas en général une notion bien définie, car il se peut que le répertoire courant ou un de ses ancêtres ait été supprimé (voyez *Read after delete* au paragraphe 2.7.4) ou qu'il y ait plusieurs liens qui y réfèrent. Cependant, on peut déterminer l'un des noms du répertoire courant (s'il existe) à l'aide de la fonction `getcwd` :

```
char *getcwd(char *buf, size_t size);
```

Certains programmes utilisent `getcwd` pour sauvegarder le répertoire courant afin de pouvoir y revenir plus tard. Losque c'est possible, il vaut mieux sauvegarder l'i-nœud lui-même à l'aide de `open(".")` et y revenir à l'aide de `fchdir`.

3.1.6 Vie et mort des processus

Intuitivement, la création d'un processus s'accompagne de l'exécution d'un programme ; cependant, sous Unix [6], la création d'un processus et l'exécution d'un programme sont deux opérations distinctes. La création d'un processus exécutant un nouveau programme recquiert donc l'exécution de deux appels système : `fork` et `exec`.

Création de processus

L'appel système `fork` Un processus est créé avec l'appel système `fork`

```
pid_t fork();
```

En cas de susccès, un appel à `fork` a pour effet de dupliquer le processus courant. Un appel à `fork` retourne deux fois [7] : une fois dans le père, où il retourne le *pid* du fils nouvellement créé, et une fois dans le fils, où il retourne 0.

En cas d'échec, `fork` ne retourne qu'une fois, avec une valeur de retour valant -1.

6. Mais pas sous Windows ou MS-DOS.
7. Au contraire d'un appel à `_exit`, qui retourne 0 fois.

L'appel système `wait` L'appel système `wait` sert à attendre la mort d'un fils :

```
pid_t wait(int *status)
```

Cet appel à le comportement suivant :

- si le processus courant n'a aucun fils, il retourne -1 avec `errno` valant `ECHILD` ;
- si le processus courant a au moins un fils zombie, un zombie est détruit et son *pid* est retourné par `wait` ;
- si aucun des fils du processus courant n'est un zombie, `wait` bloque en attendant la mort d'un fils.

Si `status` n'est pas `NULL`, l'appel à `wait` y stocke la raison de la mort du fils. Cette valeur est opaque, mais peut être analysée à l'aide des macros suivantes :

- `WIFEXITED(status)` retourne vrai si le fils est mort de façon normale, i.e. du fait d'un appel à `_exit` ;
- `WEXISTSTATUS(status)`, retourne le paramètre de `_exit` utilisé par le fils ; cette macro n'est valide que lorsque `WIFEXITED(status)` est vrai.

L'appel système `waitpid` L'appel système `waitpid` est une version étendue de `wait` :

```
pid_t waitpid(pid_t pid, int *status, int flags);
```

Le paramètre `pid` indique le processus à attendre ; lorsqu'il vaut -1, `waitpid` attend la mort de n'importe quel fils (comme `wait`). Le paramètre `flags` peut avoir les valeurs suivantes :

- 0 : dans ce cas `waitpid` attend la mort du fils (comme `wait`) ;
- `WNOHANG` : dans ce cas `waitpid` récupère le zombie si le processus est mort, mais retourne 0 immédiatement dans le cas contraire.

L'appel système `waitpid` avec `flags` valant `WNOHANG` est un exemple de variante *non-bloquante* d'un appel système bloquant, ce que nous étudierons de façon plus détaillée dans la partie 4.

Mort d'un processus

Normalement, un processus meurt lorsqu'il invoque l'appel système `_exit` :

```
void _exit(int status);
```

Lorsqu'un processus effectue un appel à `_exit`, il passe à l'état zombie, dans lequel il n'exécute plus de code. Le zombie disparaîtra dès que sont père fera un appel à `wait` (voir ci-dessous).

La foncontion `exit`, que vous avez l'habitude d'invoquer, est équivalente à `fflush(NULL)` suivi d'un appel à `_exit`. Voyez aussi le paragraphe 3.1.7 sur la relation entre `exit` et `return`.

3.1.7 Exécution de programme

L'exécution d'un programme se fait à l'aide de l'appel système `execve` :

```
int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

En cas de succès, `execve` remplace le processus courant par un processus qui exécute le programme contenu dans le fichier `filename` avec les paramètres donnés dans `argv` et avec un environnement égal à celui contenu dans `envp`. Dans ce cas, `execve` ne retourne pas — le contexte dans lequel il a été appelé a été détruit (remplacé par un contexte du programme `filename`), il n'y a donc pas « où » retourner.

Fonctions utilitaires L'appel système `execve` n'est pas toujours pratique à utiliser. La bibliothèque standard contient un certain nombre de *wrappers*[8] autour d'`execve`. Parmi ceux-ci, les plus utiles sont `execv`, qui duplique l'environnement du père, `execvp`, qui fait une recherche dans `PATH` lorsqu'il est passé un chemin relatif, et `execlp`, qui prend ses arguments en ligne terminés par un `NULL` (arguments *spread*) :

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

Parenthèse : `_start` Lorsqu'un programme est exécuté, la première fonction qui s'exécute est la fonction `_start` de la bibliothèque C. Cette fonction effectue les actions suivantes :

- elle calcule les arguments de ligne de commande et l'environnement, d'une façon dépendante du système ;
- elle stocke `environ` dans une variable globale ;
- elle invoque `main` en lui passant `argc` et `argv` ;
- si `main` retourne, elle invoque `exit`.

La fonction `_start` est la raison pour laquelle un retour normal de `main` (avec `return`) provoque une terminaison du programme.

3.1.8 Exécution d'un programme dans un nouveau processus

Pour exécuter un programme dans un nouveau processus, il faut d'abord créer un nouveau processus (`fork`) puis exécuter le processus dans le père (`execve`). Il faut ensuite s'arranger pour exécuter `wait` dans le père.

Dans le cas d'une exécution *synchrone*, où le père ne s'exécute pas pendant l'exécution du fils, le schéma typique est le suivant :

```
pid = fork();
if(pid < 0) {
    /* Gestion des erreurs */
} else if(pid > 0) {
    execlp(...);
    /* Gestion des erreurs ? */
    exit(1);
}
pid = wait(NULL);
```

8. Vous voyez une bonne traduction ?

Digression : le double `fork` Lors d'une exécution asynchrone, où le père continue à s'exécuter durant l'exécution du fils, il faut s'arranger pour que le père appelle `wait` après la mort du fils afin d'éliminer le zombie de celui-ci. Une astuce parfois utilisée consiste à déshériter le fils en appelant `fork` deux fois de suite, et en tuant le fils intermédiaire ; du coup, le petit-fils est adopté par le processus `init` :

```
pid = fork();
if(pid < 0) {
    /* Gestion d'erreurs */
else if(pid == 0) {
    /* Fils intermédiare */
    pid = fork();
    if(pid < 0) {
        /* Gestion d'erreurs impossible */
    } else if(pid == 0) {
        /* Petit-fils */
        execlp(...);
        /* Gestion d'erreurs impossible */
        exit(1);
    }
    exit(0);
}
/* Père, attend la mort du fils intermédiare */
pid = wait(NULL);
```

Il existe une autre technique permettant d'obtenir un résultat analogue sans modifier la hiérarche des processus, et qui consiste à ignorer le signal `SIGCHLD` (voir partie 3.10).

3.2 Communication par fichiers

Comme nous l'avons vu, chaque processus est isolé dans son espace de mémoire. Afin de partager les données entre les processus, il faut que ceux-ci communiquent explicitement.

La façon la plus habituelle de transférer les données est d'utiliser un fichier. Un processus *A* écrit les données dans un fichier, que le processus *B* lit ensuite ; le point de rendez-vous entre les deux processus est le nom de fichier. Par exemple, lorsque vous éditez un fichier `hello.c` puis vous le compilez, la communication entre l'éditeur de texte et le compilateur se fait à travers le fichier `hello.c`.

Lorsque plusieurs processus accèdent au même fichier, il est important que leurs actions soient *synchronisées*.

3.2.1 Synchronisation à l'aide de `wait`

Si un processus *A* écrit un fichier qu'un processus *B* doit lire, il est important que *B* attende que *A* ait fini d'écrire le fichier. Le cas le plus simple est celui où *A* est le fils de *B*, et *A* termine après

l'écriture du fichier — dans ce cas, il suffit que *B* attende la mort de *A* à l'aide de l'appel système `wait` ou `waitpid`.

Dans des cas plus compliqués, *B* ne termine pas immédiatement, ou alors *A* et *B* ne sont pas apparentés ; dans ce cas, il faut utiliser des primitives de communication plus compliquées, par exemple un octet écrit sur un tube nommé, un signal ou un sémaphore.

3.2.2 Cohérence des accès

Lorsque plusieurs processus accèdent simultanément au même fichier, il faut qu'ils se coordonnent pour faire des accès cohérents. Il existe deux cas.

Conflit écriture-écriture : *lost update* Si les processus *A* et *B* font simultanément un cycle lecture-modification-écriture sur un même fichier *f*, une mise à jour peut être écrasée par l'autre. Considérons par exemple le cas où *A* et *B* lisent un entier contenu dans le fichier, lui ajoutent 10, et écrivent le résultat. Si les deux accès ont lieu à peu près au même moment, il se peut que *A* et *B* lisent tous deux une valeur *n*, lui ajoutent 10, et écrivent tous deux $n + 10$. La première mise à jour a été perdue — on parle de *lost update* (figure 3.5).

A	*B*
`read(fd, &x, sizeof(x));`	
	`read(fd, &x, sizeof(x));`
`y = x + 10;`	
	`y = x + 10;`
`lseek(fd, 0L, SEEK_SET)`	
`write(fd, &y, sizeof(y));`	
	`lseek(fd, 0L, SEEK_SET)`
	`write(fd, &y, sizeof(y));`

FIGURE 3.5 — *Lost update*

Pour éviter ce problème, il faut garantir que chacun des cycles lecture-modification-écriture se fasse de façon atomique, c'est à dire sans pouvoir être interrompu par l'autre.

Conflit écriture-lecture : état incohérent Lorsqu'une mise à jour consiste de plusieurs écritures, il se peut qu'un lecteur observe un état incohérent — un état intermédiaire qui ne devrait pas exister. Considérons par exemple un fichier consistant de deux octets, et que l'algorithme utilisé demande qu'à tout moment au moins l'un des deux octets vaille 0. Si le fichier contient les octets (0, 1), et un processus fait :

```
unsigned char zero = 0, one = 1;
lseek(fd, 0L, SEEK_SET);
write(fd, &one, 1);
write(fd, &zero, 1);
```

alors un processus qui observe l'état du fichier entre les deux écritures peut observer l'état « impossible » (1, 1). Comme dans le cas précédent, la solution consiste à garantir que toutes les écritures qui constituent une mise à jour se fassent de façon atomique.

Écritures non-atomiques POSIX ne garantit pas que l'appel système `write` est atomique lorsqu'il est appliqué à un fichier ordinaire. Une écriture de deux octets peut potentiellement être implémentée comme deux écritures d'un octet chacune, ce qui mène au problème souligné dans le paragraphe précédent même si l'on prend soin d'implémenter chaque transaction par une seule écriture. (Si vous n'êtes pas convaincu, remplacez « octet » par « gigaoctet » et relisez ce paragraphe.)

À ma connaissance, POSIX ne fait de garanties d'atomicité pour l'appel système `write` que dans les cas suivants :

- lors d'une écriture sur un fichier ouvert avec `O_APPEND` (voyez le paragraphe 2.5.2), l'extension du fichier est atomique (ce qui garantit l'absence de conflits écriture-écriture) ;
- lors d'une écriture dans un tube ou une *socket*, l'écriture est atomique sous les conditions décrites au paragraphe 3.3.2.

3.2.3 *Locks*

Les propriétés du paragraphe précédent ne permettent les accès concurrents que dans des cas très spécifiques. Pour implémenter des accès concurrents plus généraux, il faudra utiliser des *primitives de synchronisation* dont l'atomicité est garantie. Pour cela, on peut soit utiliser la primitive habituelle de création de fichier, `open`, qui offre certaines garanties d'atomicité (paragraphe 3.2.3), ou alors utiliser des primitives *ad hoc* (paragraphe 3.2.3).

Lockfiles

La technique d'exclusion mutuelle la plus élémentaire [9] consiste à créer atomiquement (`O_EXCL`) un fichier « de *lock* » *(lockfile)* au début de la transaction, et à le supprimer à la fin.

Pour prendre un *lockfile*, un processus fait :

```
fd = open("fichier.lck", O_WRONLY | O_CREAT | O_EXCL, 0666);
```

Si le *lockfile* n'existe pas encore, l'opération réussit et le fichier est créé — le processus a le droit de modifier les données partagées. Si un autre processus a déjà créé le *lockfile*, alors l'opération échoue, avec `errno` valant `EEXIST` (c'est ce que demande l'option `O_EXCL`) ; il faut alors attendre un petit temps (par exemple 100 ms, mais ça dépend du problème) et recommencer.

Lorsqu'il a fini de manipuler les données, le processus supprime le *lockfile* à l'aide de

```
rc = unlink("fichier.lck");
```

ce qui permet à un autre processus de créer un nouveau *lockfile*.

9. Au sens qu'elle ne fait intervenir que des primitives que vous connaissez déjà. Ça ne veut pas forcément dire que c'est facile.

Remarquez que cette technique est correcte car l'option `O_EXCL` est atomique : si la création réussit, aucun autre processus n'a pu créer le fichier entre le moment où son existence a été vérifiée et le moment où le fichier a été créé. Il n'aurait pas été correct de vérifier l'existence du fichier (à l'aide de `access` ou `stat`) et seulement ensuite créer le fichier.

Les *lockfiles* ont quelques propriétés qui ne sont pas forcément pratiques. Tout d'abord, le processus qui n'a pas réussi à créer le fichier fait une attente active, ce qui n'est pas forcément une bonne idée, surtout sur une machine alimentée sur batterie (un téléphone portable, par exemple). Ensuite, si le processus qui détient le *lock* termine sans le relâcher (par exemple parce qu'il s'est planté), le *lockfile* persiste jusqu'à ce qu'un administrateur humain le supprime [10].

`flock`

L'appel système `flock` permet d'acquérir ou de relâcher de façon atomique un *lock*, une structure de données associée à une entrée de la table de fichiers ouverts (figure 2.3) qui peut servir à deux processus qui coopèrent de synchroniser leurs accès à un fichier fichier.

Il existe deux types de *locks* manipulés par `flock` : les *locks exclusifs*, utilisés par un processus qui désire modifier le fichier, et les *locks partagés*, utilisés par un processus qui désire seulement le lire. Les règles sont les suivantes :

- si un processus *A* détient un *lock* exclusif associé à un fichier *F*, aucun autre processus ne peut acquérir de *lock* associé au même fichier, exclusif ou partagé ;
- si un processus *A* détient un *lock* partagé associé à un fichier *F*, d'autres processus peuvent acquérir des *locks* partagés, mais pas des *locks* exclusifs.

L'appel système `flock` a le prototype suivant :

```
int flock(int fd, int operation)
```

Le paramètre `operation` indique l'opération a effectuer ; il vaut `LOCK_SH` ou `LOCK_EX` pour acquérir un *lock*, et `LOCK_UN` pour relâcher un *lock*. Si le *lock* demandé ne peut pas être acquis, l'appel `flock` bloque, sauf si le paramètre `operation` est combiné avec `LOCK_NB` *(non-blocking)*, dans quel cas il retourne -1 avec `errno` valant `EWOULDBLOCK`.

Un appel à `flock` sur un fichier sur lequel le processus appelant détient déjà un *lock* réussit, et a pour effet de changer le type du *lock*. Cependant, cette opération n'est pas forcément atomique (sur la plupart des systèmes, `flock` relâche l'ancien *lock* avant de prendre le nouveau), et même lorsqu'elle l'est, elle est généralement sujette à des *deadlocks* : si deux processus essaient simultanément de transformer un *lock* partagé en un *lock* exclusif, ils vont causer un *deadlock*. Pour ces raisons, il n'est à ma connaissance jamais utile de tranformer un *lock* sans le relâcher auparavant.

Comportement de `flock` à travers `fork` Un *flock* est attaché à une entrée de la table de fichiers ouverts (voyez la figure 2.3). Si un descripteur de fichiers est dupliqué par `fork` ou `dup2`, les deux copies du descripteur réfèrent à la même entrée de la table de fichiers ouverts — le père

10. Il n'est pas clair que ce soit un défaut, cela n dépend de l'application. Un processus qui meurt lorsqu'il détient un *lock* peut laisser la structure de données partagée dans un état incohérent, il peut donc être raisonnable de demander l'intervention d'un administrateur humain dans ce cas.

et le fils détiennent donc tous les deux le *lock*. Pour préserver la propriété qu'un *lock* n'est normalement détenu que par un seul processus, il est nécessaire que l'un des deux processus relâche le *lock* immédiatement après l'appel à `fork`, soit directement en appelant `fork`, soit en fermant le descripteur de fichier.

Cette propriété permet de facilement faire hériter un fils d'un *lock* : il suffit de prendre le *lock* avant `fork`, puis de fermer le descripteur dans le père.

`fcntl` L'appel système `flock`, dû à Unix BSD, est simple, correct et utile ; la norme POSIX d'origine a donc formalisé un appel système différent, `fcntl`, dû à Système V, qui est compliqué, incorrect, et a des fonctionnalités qui ne sont jamais utilisées en pratique. La principale fonctionnalité supplémentaire de `fcntl` est la possibilité d'attacher un *lock* a une partie d'un fichier.

Les locks `fcntl` sont attachés à l'i-nœud mémoire, ce qui les rend inutilisables en pratique : dès qu'un processus qui détient le *lock* ferme le descripteur, tous les processus perdent le *lock*. La seule façon d'utiliser `fcntl` est d'éviter d'utiliser `fork`, ce qui est difficile ou impossible, surtout lorsqu'on utilise des bibliothèques qui peuvent faire un appel à `fork` à tout moment.

Je n'ai jamais utilisé `fcntl`, et je vous conseille d'en faire de même.

Les *locks* sont consultatifs Revenons à la technique du *lockfile* décrite ci-dessus. Sa correction dépend du fait que tous les processus coopèrent et ne modifient un fichier qui s'ils détiennent le *lockfile* correspondant. Le système n'empêche pas un processus délinquant de contourner la discipline du *lockfile* et d'accéder directement au fichier protégé.

Il en est de même des *locks* : la convention qu'on n'accède pas à un fichier partagé sans détenir de *lock* dessus n'est pas vérifiée par le système. On dit que les *locks* sont *consultatifs* (par opposition aux *locks* dits *mandatoires*, utilisés notamment sous *Windows*).

3.3 Communication par messages

Il existe deux paradigmes principaux de communication : la communication par structures de données partagées, où plusieurs processus accèdent à une structure de données partagée, et le passage de messages, où les processus s'envoient des messages. La communication par fichiers partagés (paragraphe 3.2) et la communication par mémoire partagée (paragraphe 3.6) sont des instances de communication par structures partagées. La communication à travers les tubes (décrite dans ce paragraphe), à travers les tubes nommés et à travers *sockets* sont des instances de la communication par messages.

La mémoire partagée demande aux acteurs de se synchroniser de façon extérieure (sémaphores, mutex, variables de condition, moniteurs, transactions). Le passage de messages, par contre, est *auto-synchronisant (self-synchronising)* et ne demande pas de synchronisation supplémentaire : un message ne peut pas être lu avant d'avoir été émis.

Le principal défaut de la communication par messages est qu'elle force les données à être copiées, parfois même deux fois (une fois depuis l'émetteur vers un tampon, une deuxième fois depuis le tampon vers le récepteur), ce qui la rend inadaptée lorsque les données sont très volumineuses. De plus, certaines primitives de communication par messages demandent que les

données soient sérialisées à l'émetteur et désérialisées au récepteur, ce qui complique le travail du programmeur.

À la différence des structures de données partagées, la communication par messages se généralise naturellement à la communication à travers le réseau. Cependant, il n'est généralement pas suffisant d'utiliser la communication par messages pour qu'un programme parallèle se distribue, car le réseau introduit un certain nombre de problèmes qui n'existent pas en local (gestion des pannes, fiabilité, congestion) et qu'une application réseau a besoin de gérer explicitement.

3.3.1 Tubes anonymes

Conceptuellement, un tube est un tampon d'octets de taille bornée qui implémente des lectures et écritures bloquantes.

Création des tubes Un tube est créé à l'aide de l'appel système `pipe` :

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

L'appel système `pipe` crée un nouveau tube. Son bout écriture est stocké en `fd[1]`, et son bout lecture est stocké en `fd[0]`.

L'appel système `pipe` crée un tube dont les deux bouts sont dans le même processus. Pour que ce tube soit utile, il faut en dupliquer les bouts en faisant un appel à `fork`. Les tubes anonymes ne peuvent donc servir qu'à la communication entre processus apparentés (qui ont un ancêtre commun qui a créé le tube avant de créer les processus).

```
    int pipefd[2];
    int rc = pipe(pipefd);
    if(rc < 0) ...
    pid_t pid = fork();
    if(pid < 0) ...
    if(pid > 0) {
        close(pipefd[1]);
        rc = read(pipefd[0], ...);
        ...
    } else {
        close(pipefd[0]);
        rc = write(pipefd[1], ...);
    }
}
```

Remarquez que la première chose que fait chacun des processus après le retour de `fork` est fermer le bout du tube qui ne l'intéresse pas. Cet appel à `close` est essentiel pour que chacun des pairs puisse être notifié de la fermeture du tube (paragraphe 3.3.3).

La sémantique des entrées-sorties sur les tubes est décrite au paragraphe 3.3.3.

3.3.2 Tubes nommés

Les tubes « anonymes » ordinaires ne permettent de communiquer qu'entre processus apparentés : un tube qui permet aux processus *A* et *B* de communiquer doit être créé dans un ancêtre commun de *A* et *B*. Un *tube nommé* est un tube qui vit dans le système de fichiers : il a donc un nom qui sert de point de rendez-vous aux processus qui désirent communiquer.

Un tube nommé est créé à l'aide de la fonction `mkfifo` (qui, sous le capot, appelle l'appel système `mknod`). Les processus y accèdent en utilisant l'appel système `open`, comme pour un fichier ordinaire.

3.3.3 Semantique des tubes

Ouverture des tubes

Un tube anonyme est créé déjà ouvert. Un tube nommé, par contre, doit être explicitement ouvert à l'aide de la fonction `open`, aussi bien par l'émetteur que par le lecteur.

L'ouverture d'un tube nommé est bloquante : un appel à `open` bloque jusqu'à ce que le pair ait lui-aussi ouvert le tube. Plus précisément :

- une ouverture en écriture bloque jusqu'à ce qu'il y ait un lecteur ;
- une ouvertire en lecture bloque jusqu'à ce qu'il y ait un écrivain.

Tubes fermés

Les tubes ont une sémantique particulière lorsqu'un des pairs ferme la connexion. Si le dernier écrivain ferme le tube, le lecteur reçoit une indication de fin de fichier (`read` retourne 0). Si le dernier lecteur ferme le tube le premier, l'écrivain est tué par un signal `SIGPIPE`. Si ce signal est ignoré (voir paragraphe 3.10.3, alors l'appel à `write` retourne `EPIPE`.

Entrées-sorties sur les tubes

À la différence des *sockets*, les tubes ne font pas de démultiplexage (il n'y a pas d'analogue à `accept`) : les données des différents écrivains apparaissent comme un seul flot. Leur utilisation avec plusieurs écrivains simultanés demande donc un peu de soin afin d'éviter que les données des différents écrivains ne se mélangent. Il est selon moi impossible d'utiliser les tubes (nommés ou anonymes) avec plusieurs lecteurs simultanés.

Rappelons tout d'abord qu'un tube implémente une communication par *flots d'octets* : les frontières des `read` et `write` ne sont pas préservées, un seul `write` peut être scindé du côté du lecteur en plusieurs `read` et, inversément, plusieurs `write` peuvent être coalescés.

Atomicité des écritures En présence de plusieurs écrivains se pose le problème de l'atomicité des écritures sur un tube. Une écriture peut s'avérer non-atomique de deux façons différentes :

- si deux `write` ont lieu en même temps, les données des deux écritures pourraient s'entrelacer ;
- si un `write` fait une écriture partielle, un deuxième `write` pourrait insérer ses données juste après les données partielles écrites.

Posix fait les garanties d'atomicité suivantes :

- si un appel à `write` passe un tampon de taille inférieure à `PIPE_BUF`, alors l'écriture est atomique, et une écriture partielle n'est pas possible (le `write` bloque jusqu'à ce que tout le tampon puisse être écrit) ;
- dans le cas contraire (tampon de taille strictement supérieure à `PIPE_BUF`), il n'y a aucune garantie : les données peuvent être entrelacées avec les données d'une autre écriture, et une écriture partielle est possible.

Posix garantit que `PIPE_BUF` vaut au moins 512. (`PIPE_BUF` vaut 4096 sous Linux, 512 sous Mac OS X).

Atomicité des lectures POSIX ne fait aucune garantie sur l'atomicité des lectures sur les tubes : les lectures partielles sont toujours possibles, et le comportement de `read` simultanés n'est à ma connaissance pas spécifié.

3.3.4 *Sockets* de domaine Unix

Comme nous l'avons vu au paragraphe précédent, l'absence de multiplexage dans les tubes limite les possibilités de communication entre plusieurs pairs : un tube (anonyme ou nommé) n'est vraiment pratique que pour la communication point-à-point (entre deux pairs et non plusieurs). La communeauté réseau a résolu le problème en implémentant le *démultiplexage* à l'intérieur des primitives de communication : chaque donnée transmise est accompagnée d'une information indiquant son émetteur, et le système s'occuppe de trier les données selon l'émetteur.

Une *socket* est le bout d'un canal de communication. Une *socket par flots (stream)* implémente une sémantique par flots d'octets (comme un tube), et garantit que la communication est strictement point-à-point : s'il y a plusieurs écrivains ou plusieurs lecteurs, la *socket* est dupliquée.

Les *sockets* par flots utilisées en réseau sont basées sur un protocole nommé TCP. Il est bien sûr possible d'utiliser TCP pour la communication locale, mais TCP est difficile à utiliser : le point de rendez-vous est une *adresse de socket*, une paire d'une adresse IP et d'un numéro de port, ce qui n'est pas pratique à manipuler et n'obéit pas à un système de permissions. TCP n'est utilisable en production que s'il est accompagné d'un système de nommage (tel que DNS ou le système des URL de HTTP) et d'un système de sécurité (tel que TLS).

Une *socket de domaine Unix* est analogue à une *socket* de domaine Internet, mais elle ne permet que la communication locale à un seul hôte. Une *socket* de domaine Unix est représentée par un i-nœud, elle vit donc dans le système de fichiers (le point de rendez-vous est un nom de fichier), et obéit aux permissions du système de fichier ; il n'est donc pas nécessaire d'authentifier et chiffrer les données qui transitent dessus. Plus précisément, le bout d'une *socket* de domaine Unix est représenté par une structure `sockaddr_un` :

```
struct sockaddr_un {
    sa_family_t sun_family;
    char sun_path[108];
};
```

Le champ `sun_family` vaut `AF_UNIX`. Le champ `sun_path` est le nom du fichier qui représente la *socket* (un « chemin » dans le système de fichiers).

À part le type spécifique de l'adresse, les *sockets* de domaine Unix se manipulent exactement comme des *sockets* de domaine Internet. Dans le cas d'une *socket* par flots, la *socket* est créée par l'appel système `socket` :

```
int sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
```

Le serveur lie ensuite la *socket* à un nom de fichier (représenté par une structure `sockaddr_un`) à l'aide de l'appel système `bind`, il effectue `listen`, puis entre dans une boucle `accept`. Le client se connecte à l'aide de l'appel `connect` (dont le paramètre est une structure `sockaddr_un`).

Bien qu'une *socket* de domaine Unix soit représentée par un i-nœud, il n'est pas possible d'y accéder en utilisant l'appel système `open` : il faut utiliser les appels système rappelés dans le paragraphe précédent. En conséquence, il n'est pas possible de rediriger la sortie d'un programme vers une *socket* à l'aide des opérateurs « > » et « >> » du *shell*.

Autres types de *sockets* Unix En outre des *sockets* par flots, le domaine Unix implémente des *sockets* par datagrammes (`SOCK_DGRAM`), ayant une sémanique de communication non-fiable et non-ordonnée, par messages (les frontières des messages sont préservées), analogues aux *sockets* UDP du domaine Internet.

Certains systèmes (notamment Linux) implémentent un troisième type de *socket* de domaine Unix, `SOCK_SEQPACKET`, qui implémente une communication par messages (les frontières des messages sont préservés), fiables et ordonnés.

Autres fonctionnalités Outre les fonctionnalités communes aux *sockets*, les *sockets* de domaine Unix ont deux fonctionnalités spécifiques qui simplifient énormément l'écriture des applications sécurisées : la possibilité de transférer son *pid* de façon sécurisée et celle de transférer un descripteur de fichier ouvert à un autre processus.

3.4 Redirections

3.4.1 Descripteurs standard

Si tous les descripteurs de fichiers sont identiques du point de vue du noyau, les descripteurs de fichier 0, 1 et 2, dits *descripteurs standard* ont un rôle conventionnel :

- le descripteur de fichier 0 s'appelle l'*entrée standard*, et il sert à fournir l'entrée à un processus qui ne lit qu'à un seul endroit ;
- le descripteur de fichier 1 s'appelle la *sortie standard*, et il sert de destination pour la sortie d'un processus qui n'écrit qu'à un seul endroit ;
- le descripteur de fichier 2 s'appelle la *sortie d'erreur standard*, et il sert de destination pour les messages d'erreur.

Comme tous les descripteurs de fichiers, ces descripteurs standard sont hérités du père. Normalement, ils ont été associés au terminal par un parent, et permettent donc de lire et d'écrire sur le terminal.

3.4.2 Redirections

Plutôt que de forcer tous les programmes à implémenter la sortie sur le terminal, vers un fichier, vers l'imprimante etc., Unix permet de *rediriger* les descripteurs standard d'un processus vers un descripteur arbitraire, par exemple un fichier ouvert auparavant ou un tube (paragraphe 3.3.1 ci-dessous).

Une redirection se fait à l'aide de l'appel `dup2`, qui copie un descripteur de fichier :

```
int dup2(int oldfd, int newfd);
```

Un appel à `dup2` copie le descripteur `oldfd` en `newfd`, en fermant celui-ci auparavant s'il était ouvert; en cas de succès, il retourne `newfd`. Après un appel à `dup2`, les entrées `oldfd` et `newfd` de la table de descripteurs de fichiers pointent sur la même entrée de la table de fichiers ouverts — le pointeur de position courante est donc partagé (voir figure 2.3).

L'appel système `dup` est une version obsolète de `dup2` :

```
int dup(int oldfd);
```

Un appel à `dup` copie le descripteur de fichier `oldfd` vers la première entrée libre de la table de descripteurs de fichier, et retourne le numéro de cette dernière.

3.4.3 Exemple

Le fragment de code de la figure 3.6 combine les plus importants des appels système vus dans ce chapitre.

3.5 La mémoire virtuelle

Pour pouvoir implémenter le temps partagé, il est nécessaire que les différents processus qui sont simultanément présents en mémoire puissent résider à des adresses différentes en mémoire physique. On peut réaliser ça à deux niveaux différents. Une solution qui ne demande aucun support matériel consiste à compiler du code *indépendant de la position (Position Independent Code, PIC)* qui peut être chargé à n'importe quelle adresse. Deux processus peuvent alors exécuter le même programme, qui est chargé à deux adresses différentes. Typiquement, le coût du code indépendant de la position est entre 5% et 20%, selon l'architecture.

On peut aussi résoudre le problème au niveau matériel, en découplant les adresses de mémoire *virtuelles*, visibles par le logiciel, des adresses *physiques*, visibles par le matériel.

Sur un système à mémoire virtuelle, le PMU (figure 3.2) est remplacé par une *unité de gestion de la mémoire virtuelle (Memory Management Unit, MMU)*. À la différence d'un PMU, qui ne peut qu'autoriser ou refuser un accès à la mémoire principale, un MMU peut faire de la *translation de la mémoire* : à chaque accès à la mémoire, le MMU consulte ses tables internes et remplace l'adresse (virtuelle) qu'a spécifiée le logiciel par une adresse (physique) différente.

Le système d'exploitation maintient un espace de mémoire virtuelle distinct pour chaque processus. À chaque changement de contexte, le système d'exploitation reprogramme le MMU pour qu'il affecte à la mémoire virtuelle des adresses qui correspondent à l'espace mémoire du nouveau processus.

```
int fd[2], rc;
pid_t pid;

rc = pipe(fd);
if(rc < 0) ...

pid = fork();
if(pid < 0) ...

if(pid == 0) {
    close(fd[0]);
    rc = dup2(fd[1], 1);
    if(rc < 0) ...
    execlp("whoami", "whoami", NULL);
    ...
} else {
    char buf[101];          /* Pourquoi 101 ? */
    int buf_end = 0;
    close(fd[1]);
    while(buf_end < 100) {
        rc = read(fd[0], buf + buf_end, 100 - buf_end);
        if(rc < 0) ...
        if(rc == 0)
            break;
        buf_end += rc;
    }

    if(buf_end > 0 && buf[buf_end - 1] == '\n')
        buf_end--;
    buf[buf_end] = '\0';

    printf("Je suis %s\n", buf);

    wait(NULL);             /* Pourquoi faire ? */
}
```

FIGURE 3.6 — Redirection vers un tube

Il existe plusieurs techniques de virtualisation de mémoire, qui diffèrent par la structure des tables de MMU. Le matériel moderne utilise une technique qui s'appelle s'appelle la *pagination*.

3.5.1 Pagination

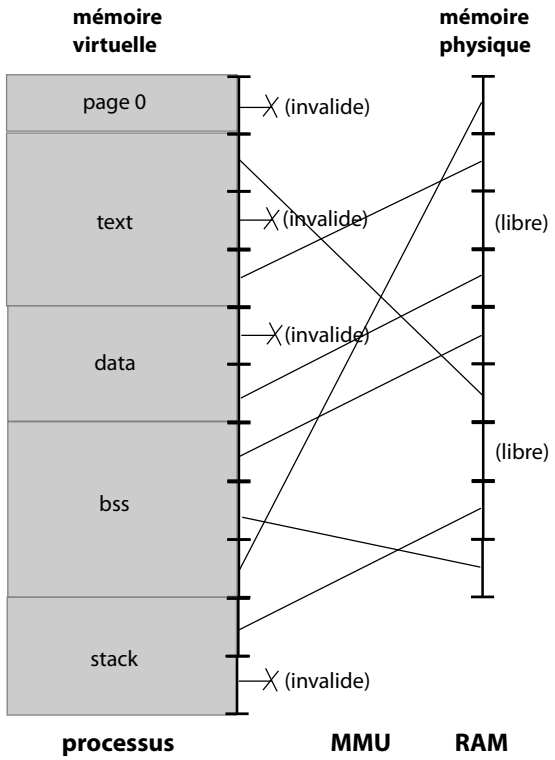


FIGURE 3.7 — Mémoire virtuelle d'un processus. (Les pages invalides des sections `text` et `data` seront chargées de manière paresseuse.)

Sur un système à pagination, la mémoire physique est divisée en unités de taille égale, appelées *pages*, typiquement d'une taille de 4096 octets (x86, AMD64, MIPS) ou 8192 (Alpha) (sur ARM, la taille de page dépend du système). Le MMU contient une structure de données qui, à chaque numéro de page de mémoire virtuelle associe un numéro de page de mémoire physique. Sur une machine où les adresses font n bits ($n = 32$ ou 64) et ayant des pages de 2^p octets ($p = 12$ ou 13), une adresse virtuelle est interprétée comme $n - p$ bits de numéro de page suivis de p bits de déplacement *(offset)* à l'intérieur de la page. Le numéro de page est converti par le MMU, le déplacement est laissé inchangé (figure 3.7).

Au niveau matériel, chaque entrée de la table de pages contient les données suivantes :

- un bit qui dit si la page est *valide* ; si ce n'est pas le cas, tout accès à cette page passe la main au noyau : on dit que le processus subit un *défaut de mémoire (memory fault)* ;
- si la page est valide, le numéro de la page de mémoire physique correspondante ;
- des bits qui indiquent les permissions de la page (lecture, écriture, exécution) ;

– des informations supplémentaires qui indiquent si la page a été lue ou écrite récemment.

Au niveau logiciel, le système maintient pour chaque page de mémoire virtuelle les informations supplémentaires suivantes :

– des informations qui indiquent d'où proviennent les données contenues dans la page : une page peut être *file-backed*, ce qui indique que les données proviennent d'un fichier, ou alors *anonyme*, ce qui indique qu'elle a initialement été initialisée à zéro ;
– pour une page valide, un bit qui indique si la page est *propre*, c'est à dire qu'elle est identique à son *backing store* (le fichier sur disque dans le cas d'une page *file-backed*, des zéros pour une page anonyme), ou alors si elle est *sale* ou *modifiée* ;
– un bit qui indique si la page est *partagée*, et alors qu'elle sera partagée à travers un *fork*, et, si elle est sale, qu'elle sera recopiée dans son *backing store*, ou alors qu'elle est *privée*, et alors elle sera dupliquée à travers un *fork*.

3.5.2 Applications

La pagination sert à implémenter plusieurs fonctionnalités fondamentales du système.

Protection de la mémoire Les pages qui ne correspondent pas à des adresses allouées par le programme sont marquées invalides, ce qui permet de tuer le processus dès qu'il essaie d'accéder aux adresses correspondantes (`SIGSEGV`). C'est aussi le cas de la page 0, ce qui permet de détecter les accès à travers le pointeur nul.

Partage de sections en lecture Lorsqu'un processus exécute un appel à `fork`, les pages correspondant à des sections du programme en lecture seulement (section `text`, voir paragraphe 3.1.3) ainsi que les zones *mappées* en lecture (paragraphe 3.6) ne sont pas dupliquées. Les tables du MMU sont programmées de façon à ce que les mêmes pages physiques apparaissent dans plusieurs processus simultanément.

Ces pages de mémoire qui réflètent le contenu d'un fichier sont appelées *file-backed*.

Copie paresseuse Lorsqu'un processus exécute un appel à `fork`, les autres pages du processus ne sont pas dupliquées non plus : elles sont partagées, et protégées en écriture. Tant que les deux processus n'effectuent que des lectures, les pages restent partagées. Si un processus effectue une écriture, le MMU suspend l'exécution du processus et invoque un gestionnaire du noyau qui duplique la page en cours d'écriture puis relance le processus à l'endroit où il a été interrompu.

Cette copie *paresseuse (Copy on Write, CoW)* permet d'optimiser l'exécution dans deux cas : lorsque les processus n'écrivent qu'une petite fraction des pages de leur espace mémoire, ou lorsque l'un des deux processus détruit son espace mémoire (`execve` ou `_exit`) juste après avoir effectué un `fork`.

Chargement paresseux Lorsqu'un programme est exécuté, le code et les données contenues dans le fichier exécutable ne sont pas vraiment chargées en mémoire ; les pages de mémoire du processus sont simplement marquées invalides. La première fois que le processus essaie d'accéder

à une page, il est suspendu par le MMU, et le système charge la page depuis le fichier, puis continue l'exécution du processus.

Allocation paresseuse Lorsqu'un programme est lancé, il a dans sa carte mémoire des zones de mémoire qui ne contiennent que des zéros binaires, notamment la section `bss` et la section `stack`. Ces zones ne sont pas vraiment allouées elles sont marquées invalides, et ne sont allouées que lorsque le processus essaie d'y accéder pour la première fois.

L'allocation paresseuse a une conséquence visible pour le programmeur : il se peut qu'une allocation de mémoire réussisse, mais que le système arrive à court de mémoire lorsque tous les programmes essaient de toucher la mémoire qu'ils ont allouée. Le système est alors obligé de tuer un processus arbitraire.

Pagination sur disque Enfin, lorsque le système se trouve à court de mémoire, il commence par libérer la mémoire *file-backed*, qu'il pourra toujours recharger depuis le disque en cas de besoin ; les pages de mémoire virtuelle correspondantes sont marquées invalides. Si cela ne suffit pas, il sauvegarde sur disque des pages de mémoire qui n'ont pas été touchées récemment. Si un processus essaie d'accéder à ces pages, il est suspendu et les pages correspondantes sont rechargées depuis le disque. On dit que le système *swappe* les pages correspondantes [11].

3.6 *Memory mapping*

L'appel système `mmap` sert à créer un *memory mapping*.

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Le paramètre `addr` indique l'adresse virtuelle désirée ; il vaudra toujours NULL pour nous, ce qui demande au système de choisir une adresse libre. Le paramètre `length` indique la taille du *mapping* (le système peut choisir une taille plus grande, il arrondit à un nombre entier de pages). Le paramètre `prot` indique les permissions de la zone de mémoire créée. Le paramètre `flags` indique s'il s'agit d'un *mapping* privé ou partagé, anonyme ou *file-backed*. Les paramètres `fd` et `offset` indiquent le fichier utilisé par un *mapping* qui est *file-backed* ; ils sont ignorés pour un *mapping* anonyme, mais il est conventionnel dans ce cas de les positionner à -1 et 0 respectivement [12]. Cet appel système retourne `MAP_FAILED` en cas d'échec (pas `NULL`), et alors `errno` est positionné.

Une zone de mémoire peut être désallouée à l'aide de `munmap` :

```
int munmap(void *addr, size_t length);
```

Les pages couvertes sont marquées invalides (un accès subséquent causera la réception d'un signal `SIGSEGV`) ; dans le cas d'un *mapping* anonyme ou privé, la mémoire physique correspondante est libérée et le contenu est perdu.

11. Ce qui est inexact : le *swapping* est une technique de gestion de la mémoire virtuelle obsolète, encore plus obsolète que la segmentation, qui elle-même a été rendue obsolète par la pagination.

12. Parce que SunOS 4.

3.6.1 *Mapping* anonyme privé

`mmap` peut servir à allouer une zone de mémoire anonyme privée :

```
p = mmap(NULL, 16 * 4096,
         PROT_READ | PROT_WRITE,
         MAP_PRIVATE | MAP_ANONYMOUS,
         -1, 0);
if(p == MAP_FAILED)
    ...
```

Cette technique est normalement utilisée par `malloc` pour obtenir de la mémoire [13], mais rien n'empêche un programme de contourner `malloc` par exemple s'il implémente un allocateur de mémoire spécialisé.

3.6.2 *Mapping* anonyme partagé

`mmap` peut servir à allouer de la mémoire anonyme qui sera partagée après un appel à `fork`, ce qui permet à des processus apparentés de communiquer sans passer par le tampon d'un tube ou d'une *socket* (attention cependant aux *race conditions* et aux problèmes de cohérence, voyez le paragraphe 3.8). La différence entre les processus et les *threads* est donc seulement une différence de comportement par défaut : les *threads* partagent tout sauf ce qu'ils ne partagent pas, alors que les processus ne partagent que la mémoire qu'ils ont explicitement marquée comme partagée.

```
p = mmap(NULL, 16 * 4096,
         PROT_READ | PROT_WRITE,
         MAP_SHARED | MAP_ANONYMOUS,
         -1, 0);
if(p == MAP_FAILED)
    ...
```

3.6.3 *Mapping* privé de fichiers

`mmap` peut créer de la mémoire *file-backed*, et donc de donner l'illusion qu'un fichier se trouve en mémoire principale.

```
fd = open(... O_RDONLY ...);
if(fd < 0)
    ...
rc = fstat(fd, &st);
if(rc < 0)
    ...
p = mmap(NULL, st.st_size,
         PROT_READ,         /* ou PROT_READ | PROT_WRITE */
```

13. Sur Linux, `malloc` utilise aussi un appel système plus simple nommé `brk`. Sous Mac OS X, `malloc` utilise uniquement `mmap`.

```
          MAP_PRIVATE,
          fd, 0);
if(p == MAP_FAILED)
    ...
```

Une lecture de la zone de mémoire retourne la partie correspondante du fichier sous-jacent. Si les permissions permettent les écritures, alors une écriture cause la copie d'une page : elle n'est pas visible aux autres processus ayant *mappé* le même fichier, et elle n'est pas réflétée dans le fichier lui-même (un `read` retournera les donées d'origine).

3.6.4 *Mapping* de fichiers partagé

Enfin, `mmap` peut donner l'illusion qu'un fichier se trouve en mémoire principale et de refléter toute modification à la mémoire dans ce dernier.

```
fd = open(... O_RDWR ...);
if(fd < 0)
    ...
rc = fstat(fd, &st);
if(rc < 0)
    ...
p = mmap(NULL, st.st_size,
          PROT_READ | PROT_WRITE,
          MAP_SHARED,
          fd, 0);
if(p == MAP_FAILED)
    ...
```

Une écriture dans la zone de mémoire correspondante est réflétée non seulement dans les autres *mappings* du même fichier, mais aussi dans le fichier sur disque. Cependant, cette écriture peut être retardée : un accès aux données à travers le système de fichiers (`read`) ne réflète pas immédiatement les changements effectués. Nous verrons au paragraphe 3.8.3 l'appel système `msync` qui permet de resynchroniser la mémoire virtuelle avec le système de fichiers.

3.6.5 Problèmes liés à `mmap`

L'allocation de mémoire anonyme, privée ou partagée, à l'aide de `mmap` ne pose pas de problèmes particuliers : la sémantique est bien définie, le mécanisme (allocation de pages à la demande) est bien compris et ne cause pas de problèmes de performance particuliers.

Le *mapping* de fichiers est plus problématique. Tout d'abord, avec `mmap` le format des données sur disque est forcément le même que le format des données en mémoire, ce qui peut empêcher certaines optimisations, par exemple la compression à la volée, et peut rendre difficile la portabilité des fichiers entre architectures.

Par ailleurs, les pages mappées par `mmap` sont peuplées à la discrétion du système, et le système peut parfois en lire trop, ou, au contraire, ne pas lire suffisamment de données à l'avance,

ce qui suspend le processus lors d'un accès à la mémoire pendant un temps imprévisible. Ces problèmes peuvent parfois être mitigés à l'aide de l'appel système `posix_madvise`, qui indique explicitement au système comment gérer une plage de mémoire.

La situation est légèrement différente pour les *mappings* partagés en écriture. Le système écrit les données de manière asynchrone, mais l'application peut demander à les écrire immédiatement à l'aide de `msync` (voir paragraphe 3.8.3). Par contre, il n'existe aucune manière, à ma connaissance, de gérer les erreurs d'écriture.

Ces problèmes font que certains programmeurs préfèrent les entrées-sorties explicites (`read` et `write` à travers des tampons) aux fichiers *mappés*, surtout lorsqu'il s'agit d'écrire des données [14].

3.7 Mémoire partagée et sémaphores POSIX

L'option `MAP_SHARED` de `mmap` permet d'établir des zones de mémoire partagée entre processus. Comme avec les tubes, il y a deux cas : le cas anonyme, qui est limité au partage entre processus apparentés (il faut établir la zone de mémoire partagée avant de faire `fork`), et le cas non-anonyme, qui permet de partager des données entre processus arbitraires.

Le cas anonyme est le plus facile (comme d'habitude). Avant d'appeler `fork`, le processus père fait un *mapping* anonyme partagé. Après le `fork`, la zone de mémoire virtuelle correspondante dans les deux processus est *backed* par la même zone de mémoire physique, et ce sont donc les mêmes données qui sont visibles dans les deux processus.

Le cas non-anonyme est un peu plus compliqué. Il faut d'abord créer un fichier qui servira de *backing store* à la zone de mémoire partagée (`open(O_CREAT)`), donner la bonne longueur au fichier (`ftruncate`), puis s'arranger pour que les deux fichiers effectuent un `mmap` sur le fichier partagé. Cette approche soulève deux problèmes :

- où placer le fichier partagé ? Le répertoire *home* d'un utilisateur n'est pas forcément un bon choix (les permissions peuvent être restrictives, et il peut s'agir d'un système de fichiers réseau).
- comment éviter que le système n'écrive les données dans le système de fichiers, ce qui pourrait créer des délais inutiles ?

La solution la plus simple à ce problème est de définir un répertoire où tous les utilisateurs ont le droit d'écriture, et qui est traité spécialement par le système pour minimiser les écritures sur disque. Par exemple, sous Linux, le répertoire `/dev/shm/` joue ce rôle, et toute application peut créer une zone de mémoire partagée efficace simplement en créant le *backing store* dans `/dev/shm/`.

Cette solution est simple et élégante, et n'a donc pas été retenue par la norme POSIX.

3.7.1 Mémoire partagée POSIX

Une zone de mémoire partagée POSIX se comporte comme un fichier, mais sa localisation dans le système de fichiers n'est pas définie. One accède à une zone de mémoire partagée POSIX à l'aide de la fonction `shm_open` :

14. Andrew Crotty, Viktor Leis and Andrew Pavlo. Are You Sure You Want to Use MMAP in Your Database Management System ? *CIDR 2022, Conference on Innovative Data Systems Research*. 2022.

```
int shm_open(const char *name, int oflag, mode_t mode)
```

Cette fonction a la même interface que la fonction `open`, sauf que le paramètre `name` ne peut pas contenir de *slashes* « / ». Elle accepte plusieurs des mêmes *flags* que `open`, notamment `O_CREAT` et `O_TRUNC`.

Comme pour un fichier ordinaire, la zone est redimensionnée à l'aide de `ftruncate`. Elle peut ensuite être partagée à l'aide de `mmap`.

Pour supprimer une zone de mémoire partagée, il faut appeler `shm_unlink` :

```
int shm_unlink(const char *name);
```

Sous Linux, `shm_open` est équivalent à un appel à `open` sur un fichier se trouvant dans `/dev/shm/`. Sous d'autres systèmes, l'implémentation peut être différente.

3.7.2 Sémaphores POSIX

Comme pour toute structure de données partagée, il n'est pas en général correct d'accéder à une zone de mémoire partagée sans utiliser de primitives de synchronisation (voir paragraphe 3.8). La solution habituelle est de placer dans la zone de mémoire partagée des primitives de synchronisation de la bibliothèque *pthreads* (`pthread_mutex_t`, `pthread_cond_t`, etc.). Il faudra alors utiliser l'option `PTHREAD_PROCESS_SHARED` pour indiquer qu'elles serviront à synchroniser des processus distincts, et pas seulement des *threads* au sein d'un même processus.

Pour les cas où le programmeur désire éviter la complexité de la bibliothèque *pthreads*, POSIX fournit une primitive simple et élégante, le *sémaphore POSIX*. Il s'agit d'une primitive : elle n'est pas forcément pratique à manipuler directement, mais elle permet de construire des primitives de synchronisation utiles comme des *mutexes* ou des variables de condition.

Pour placer un sémaphore POSIX à l'adresse spécifiée par `sem` (qui sera typiquement dans une zone de mémoire partagée), on appelle `sem_init` :

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Le paramètre `pshared` vaudra 1 (il indique que le sémaphore sera partagé entre processus et non seulement entre *threads*), et `value` est la valeur initiale du sémaphore. Les opérations *P* et *V* s'écrivent respectivement :

```
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

Il existe d'autres opérations sur les sémaphores POSIX, telles que `sem_timedwait` et `sem_value`.

Sémaphores nommés

Les sémaphores décrits ci-dessus doivent être placés dans une zone de mémoire partagée, qui peut être anonyme ou non. POSIX définit aussi une notion de *sémaphore nommé*, qui consiste essentiellement d'une zone de mémoire partagée POSIX contenant un seul sémaphore. Les sémaphores nommés sont créés à l'aide de `sem_open`, et détruits à l'aide de `sem_unlink`.

Les sémaphores nommés ne sont généralement pas utiles : il est naturel de placer un sémaphore dans la zone de mémoire partagée qu'il protège. Malheureusement, Mac OS X ne supporte pas les sémaphores placés dans une zone existante, et force donc le programmeur à utiliser `sem_open` même dans le cas où c'est `sem_init` qui serait pratique.

3.8 Cohérence des accès

Les différentes mémoires qui se trouvent dans un ordinateur ont des latences très différentes. À un extrême, un disque à rouille tournante a une latence moyenne d'une demi-révolution en moyenne ; à 6000 tours par minute, c'est 5 ms. La mémoire principale (RAM) a une latence intermédiaire, de l'ordre de 50 à 100 ns. Les registres du processeur ont une latence d'une fraction de nanoseconde. Pour acquérir une intuition de ces valeurs, il suffit de multiplier les temps par la vitesse de la lumière : les données sur disque sont très loin de nous, à 1500 km ; les données en mémoire principale sont à 15 à 30 m ; et les registres sont à quelques centimètres (voir figure 3.8).

Type de mémoire	latence typique	latence × c
Disque dur	5 ms	1500 km
SSD (lecture)	100 µs	30 km
RAM	100 ns	30 m
Cache L2	20 ns	6 m
Cache L1	2 ns	60 cm
Registre	0.2 ns	6 cm

FIGURE 3.8 — Latence typique des différents types de mémoire

Pour cacher cette latence, les systèmes implémentent des *caches* [15], des zones de mémoire qui se trouvent près du processeur et qui contiennent une copie d'une partie des données se trouvant plus loin. Il existe principalement deux types de caches : le cache du système de fichiers, implémenté par le système d'exploitation, et les caches du processeur, implémentés par le matériel.

En plus des problèmes d'atomicité des écritures et des lectures (étudiés dans les chapitres précédents), ces hiérarchies complexes introduisent deux problèmes :

- *cohérence* : est-ce que deux processus ou *threads* voient les mêmes données à un moment donné ;
- *persistance* : est-ce que les données qui doivent être écrites sur un support persistant (le disque) ont été écrites de façon à survivre à une panne de courant.

3.8.1 Le *buffer cache*

Le *buffer cache* ou *page cache* est une structure de données maintenue en mémoire principale par le système d'exploitation. Lorsque le système a besoin de faire une lecture depuis le système de fichiers (par exemple parce qu'un processus a fait un appel à `read`, ou parce qu'il faut satisfaire

15. Prononcé [kæʃ] en anglais, comme le mot *cash*.

un défaut de page), il consulte le cache : si les données ne s'y trouvent pas, il les recopie depuis le disque dans le cache ; si par contre elles s'y trouvent déjà (parce qu'elles ont été lues récemment), le système s'en sert sans consulter le périphérique de stockage de masse. Si le périphérique de stockage de masse n'est pas occuppé, le système peut parfois anticiper une lecture, et copier les données vers le cache avant que le processus utilisateur les demande : c'est le *prefetching*.

Il en est de même pour les écritures : un appel à `write` recopie les données dans le cache, les données seront écrites sur le périphérique de stockage de masse plus tard, lorsque ce dernier ne sera pas occuppé (c'est la politique *write-back*, par opposition à *write-through*). Le fait de retarder les écritures permet aussi de les agréger (faire plusieurs écritures en une seule opération, ce qui est particulièrement intéressant sur un disque rotatif) et parfois même de les éviter entièrement (si les données sont rendues obsolètes par une écriture subséquente).

Le *buffer cache* a une taille variable, mais bornée [16], il est donc nécessaire de libérer des pages du cache lorsque sa taille croît trop. L'algorithme qui décide quelles données éliminer du cache s'appelle l'algorithme d'*éviction*, et il dépend du système.

Conséquences Après un `write`, les données se trouvent dans le *buffer cache* ; elles seront écrites sur le disque plus tard, traditionnellement dans la minute qui suit sous Unix [17]. Normalement, ce délai n'est pas détectable par un programme : un appel à `read` retourne les données du cache, ce qui fait que les anciennes données se trouvant sur le disque *(stale)* ne sont pas visibles au programme. Le *buffer cache* ne cause donc pas de problème de cohérence entre processus.

Cependant, le *buffer cache* introduit un problème de persistance : si le système se plante après qu'un processus a fait un appel à `write`, il se peut que les données n'aient pas encore été écrites sur le disque et soient donc perdues. Dans le pire cas, un programme ouvre un fichier avec `O_TRUNC` puis écrit une nouvelle version des données ; si le système choisit d'effectuer la troncation immédiatement, mais de retarder l'écriture des données, un plantage du système peut mener à une situation où ni l'ancienne ni la nouvelle version des données ne sont disponibles.

L'appel système `fsync` permet de gérer explicitement la cohérence du *buffer cache*. Lorsque `fsync` est exécuté sur un descripteur de fichiers, les données associées à ce descripteur sont stockées de façon persistante avant que `fsync` retourne [18]. Attention, `fsync` est une opération coûteuse, surtout sur les systèmes de fichiers à journal, il est donc important de ne pas en abuser.

Les deux séquences de la figure 3.9 sont traditionnellement utilisées pour sauvegarder les données ; sur un système de fichiers pas trop boggué, elles devraient garantir qu'au moins une des deux versions d'un fichier se trouve sur le disque quel que soit le moment où le système se plante.

3.8.2 Les caches du processeur

Les caches du processeur se trouvent entre le processeur et la mémoire principale (figure 3.10). Il y a typiquement deux niveaux de cache : un cache de niveau 1 (*level 1*, L1), spécifique à chaque cœur, et un cache de niveau 2 (L2), partagé entre plusieurs cœurs (figure 3.10). Les processeurs pour serveurs ont parfois aussi un cache de niveau 3 (L3), les processeurs embarqués n'ont souvent

16. Sous Linux, la commande `free` affiche sa taille.
17. Les Unix optimisés pour les ordinateurs portables ont des délais plus longs, jusqu'à 10 minutes sous Linux.
18. `fsync` est boggué sous Mac OS X (regardez la page de man, c'est documenté).

```
fd = open("toto.tmp",
          O_WRONLY | O_EXCL,
          0666);
if(fd < 0)
    ...
rc = write(fd, ...);
if(rc < 0)
    ...
rc = fsync(fd);
if(rc < 0)
    ...
close(fd);
rc = rename("toto.tmp", "toto");
```

```
rc = rename("toto", "toto~");
if(rc < 0)
    ...
fd = open("toto",
          O_WRONLY | O_EXCL,
          0666);
if(fd < 0)
    ...
rc = write(fd, ...);
if(rc < 0)
    ...
rc = fsync(fd);
if(fd < 0)
    ...
close(fd);
rc = unlink("toto~");
```

FIGURE 3.9 — Séquences de code pour une sauvegarde fiable

qu'un cache de niveau 1 [19].

Lorsque le processeur lit une donnée, il consulte le cache de niveau 1 ; si elle ne s'y trouve pas, il consulte le cache de niveau 2 ; enfin, si elle ne s'y trouve pas, il consulte la mémoire principale. Une fois trouvée, la donnée est recopiée dans la hiérarchie de caches (depuis la mémoire principale vers le cache L2, et depuis le cache L2 vers le cache L1).

Le processeur optimise ces opérations de plusieurs façons. Tout d'abord, lors d'une lecture il lit un bloc entier de données, une *ligne de cache* (64 octets sur Intel), et pas seulement la donnée demandée. De plus, lorsqu'il détecte des lectures séquentielles, il anticipe les lectures en préchargeant *(prefetch)* les données dans le cache avant que le programme en ait besoin. Ces optimisations font que les lectures séquentielles sont jusqu'à 100 fois plus rapides que les lectures aléatoires [20].

Le processeur optimise aussi les écritures. Lorsque le programme écrit dans la mémoire (par exemple en affectant une valeur à une variable globale), la ligne de cache entière est lue dans le cache L1 et modifiée par le processeur ; l'écriture de la ligne de cache *(writeback)* se fera plus tard, ou peut-être jamais.

Conséquences Si une zone de mémoire apparaît dans l'espace mémoire de plusieurs processus ou *threads*, le système peut choisir de les exécuter sur des cœurs ou des processeurs différents, ce qui fera qu'ils verront des caches différents (voir figure 3.10). Les propriétés de cohérence garanties par le matériel dépendent de l'architecture, et elles sont assez faibles sur plusieurs architectures assez répandues (notamment ARM, utilisé dans les téléphones portables, ainsi que POWER/PowerPC), ce qui fait qu'on ne peut pas compter sur une mémoire physique cohérente entre cœurs.

19. Sous Linux, la commande `cat /proc/cpuinfo` permet de déterminer la taille totale des caches du processeur.

20. Et maintenant vous savez pourquoi les B-trees sont préférés aux AVL. Le programmeur maniaque choisit l'arité de son B-tree pour qu'un nœud ait exactement la taille d'une ligne de cache.

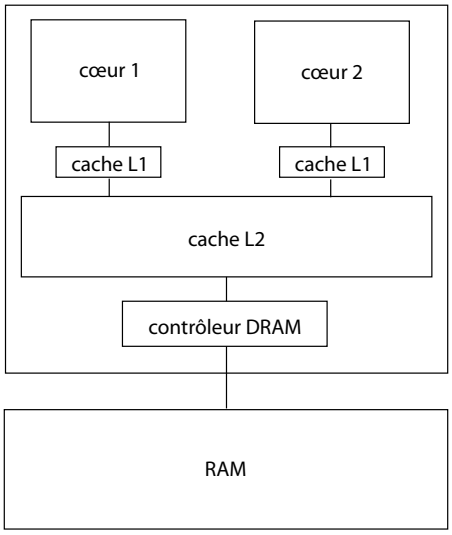


FIGURE 3.10 — Le processeur et ses caches

Considérons un programme qui exécute en parallèle (dans deux *threads*) les deux fragments de code suivants, où `x` et `y` sont des variables partagées initialisées à 0 :

```
     int x = 0, y = 0;
            |
x = 1;      |    printf("%d\n", x);
y = 1;      |    printf("%d\n", y);
```

Intuitivement, on s'attendrait à ce que le *thread* de droite puisse observer les séquences $(0, 0)$, $(1, 0)$ ou $(1, 1)$. Cependant, sur une architecture non-cohérente, il se peut que l'écriture de `x` soit retardée jusqu'après l'écriture de `y`, ou alors que la lecture de `y` soit anticipée ; il est alors possible (mais peu probable) d'observer la séquence $(0, 1)$. Une conséquence pratique est qu'une implémentation naïve des algorithmes d'exclusion mutuelle de Dekker ou Peterson est incorrecte sur ARM ou POWER.

La solution consiste à introduire des *barrières de mémoire (memory fences)*, des instructions qui imposent un ordre globalement visible aux opérations sur la mémoire. Intuitivement, une barrière est une instruction qui ne fait rien, mais qui empêche les instructions qui la précèdent d'être retardées, et les instructions qui la suivent d'être anticipées — personne ne « double » une barrière. La nature précise des barrières disponibles dépend de l'architecture du processeur, mais les normes C11 et POSIX font les garanties suivantes :

- chaque primitive de synchronisation (`mutex_lock`, `mutex_unlock` etc.) contient une barrière ;
- les opérations atomiques cohérentes (voir chapitre suivant) contiennent des barrières.

En conséquence, si chaque accès à une donnée partagée est (correctement) protégé par un *lock*, aucun problème de cohérence ne sera observable.

3.8.3 Cohérence de `mmap`

Il y a une troisième source possible d'incohérence dans un système à mémoire virtuelle : lorsque deux processus ont fait un `mmap` partagé (`MAP_SHARED`) sur le même fichier (soit parce que chacun a fait `mmap`, soit parce qu'un `mmap` a été fait avant un `fork`), les données sont visible chacun des deux *mappings*, et la version du système de fichiers, accessible par `read` et `write`. POSIX ne fait à ma connaissance aucune garantie sur la cohérence de ces trois versions.

Gestion explicite de la cohérence Pour être portable, la cohérence doit donc être gérée explicitement par le programmeur. Cela peut se faire à l'aide de l'appel système `msync` :

```
int msync(void *addr, size_t length, int flags);
```

Si le paramètre `flags` vaut `MS_SYNC`, les données de la plage d'adresses sont copiées dans le système de fichiers (le *buffer cache*), ce qui les rend visibles à `read`. Si le paramètre `flags` vaut `MS_SYNC | MS_INVALIDATE`, alors les données sont rendues visibles non seulement au système de fichiers mais aussi aux autres processus qui ont mappé cette plage de données.

Cohérence sur un système raisonnable Si POSIX fait peu de garanties, la plupart des concepteurs de systèmes de mémoire virtuelle sont raisonnables, et n'introduisent pas gratuitement d'incohérences entre *mappings* [21]. De ce fait, sur la plupart des systèmes les copies en mémoire ont les mêmes propriétés de cohérence que le matériel, et la communication par mémoire partagée est donc possible sans `MS_INVALIDATE` à condition de comprendre les propriétés de cohérence du matériel.

Attention cependant, ceci ne s'applique qu'aux copies visibles en mémoire virtuelle — la copie visible au système de fichiers (`read`) n'est pas cohérente sans utiliser des `msync` explicites.

3.9 Opérations atomiques

Comme nous l'avons vu dans les parties précédentes, les accès à la mémoire ne sont ni atomiques ni cohérents, et il est donc nécessaire de protéger les variables partagées par des primitives de synchronisation (*mutex* ou *sémaphores*).

Les primitives de synchronisation sont implémentées à l'aide d'opérations *atomiques et cohérentes* implémentées par le matériel lui-même. Ces opérations permettent aussi d'implémenter des algorithmes dits *non-bloquants* qui permettent d'accéder à des valeurs partagées sans utiliser de primitives de synchronisation explicites, ce qui est généralement difficile mais permet parfois de gagner en capacité de passage à l'échelle *(scaling)* d'un programme ; typiquement, ces techniques sont nécessaires au-delà de 4 ou 6 cœurs.

Depuis sa version de 2011, le langage C contient une interface portable à ces opérations, déclarées dans l'entête `stdatomic.h`.

21. HP-UX constitue une exception notable.

3.9.1 Entiers atomiques

Le type `atomic_int` représente un entier qui peut être manipulé de façon atomique et cohérente. Il supporte notamment les opérations suivantes :

```
void atomic_store(volatile atomic_int *object, int desired);
int atomic_load(volatile atomic_int *object);
int atomic_fetch_add(volatile atomic_int *object, int operand);
int atomic_exchange(volatile atomic_int *object, int desired);
bool atomic_compare_exchange_strong(volatile atomic_int *object,
                                    int *expected, int desired);
bool atomic_compare_exchange_weak(volatile atomic_int *object,
                                  int *expected, int desired);
```

Les fonctions `atomic_store` et `atomic_load` écrivent et lisent la valeur d'un `atomic_int`. La fonction `atomic_fetch_add` ajoute atomiquement une valeur à un `atomic_int`, et retourne la valeur précédente de ce dernier. La fonction `atomic_exchange` stocke `desired` dans `object` et retourne la valeur précédente de ce dernier.

La fonction `atomic_compare_exchange_strong` est plus intéressante. Elle compare son paramètre `object` à `expected`. Si les valeurs sont égales, elle stocke `desired` dans `object` et retourne `true`; sinon, elle stocke `object` dans `expected`, et retourne `false`. La variante `_weak` est analogue, mais elle peut occasionnellement échouer (retourner `false`) alors que la comparaison est vraie.

Les opérations ci-dessus sont atomiques et très fortement cohérentes — très précisément, elles sont *séquentiellement consistantes*. Il existe des variantes *explicites* de ces fonctions qui prennent un paramètre supplémentaire qui indique explicitement les propriétés de consistence demandées — je vous déconseille de vous en servir, sauf si vous savez vraiment très bien ce que vous faites.

Exemple : compteur Il est parfois nécessaire d'obtenir une suite globalement croissante d'entiers, par exemple pour affecter des identificateurs uniques aux objets qu'on alloue. Une implémentation d'un compteur global pourrait s'écrire comme suit :

```
static int counter = 0;
static pthread_mutex_t mu = PTHREAD_MUTEX_INITIALISER;

int next(void) {
    int v;
    pthread_mutex_lock(&mu);
    v = counter;
    counter++;
    pthread_mutex_unlock(&mu);
    return v;
}
```

Remarquez l'utilisation d'un *mutex* pour garantir l'atomicité de la lecture de `x` et de l'incrémentation. Une fonction ayant le même comportement pourrait s'écrire à l'aide d'un compteur atomique et de la fonction `fetch_add` :

```
static atomic_int counter;
...
atomic_store(&x, 0);
...
int next(void) {
    return atomic_fetch_add(&counter, 1);
}
```

Remarquez que `atomic_int` n'est pas forcément identique à `int`, il faut donc initialiser le compteur explicitement à l'aide de `atomic_store`.

Exemple : opération atomique arbitraire La fonction `atomic_fetch_add` permet d'effectuer une addition de façon atomique ; mais qu'en est-il des autres opérations ? Par exemple, comment pourrait-on élever une valeur au carré de façon atomique ? La fonction `atomic_compare_exchange`, normalement appelée *CAS (compare and swap)*, est *universelle* sur les entiers : avec un peu de travail, elle permet d'implémenter n'importe quelle opération atomique et cohérente.

Considérons le code suivant :

```
x = x * x;                  // non-atomique
```

Bien sûr, cette opération n'est pas atomique. On modifie cet exemple pour séparer le calcul des lectures et écritures, et on vérifie que la valeur n'a pas été modifiée entre temps :

```
while(1) {
    int a = x
    int b = a * a;
    if(x == a) {
        x = b;              // race
        break;
    }
}
```

Comme le test et l'affectation ne s'exécutent pas de façon atomique, il y a une *race condition* — ce qui est exactement le problème que l'opération CAS permet d'éviter :

```
while(1) {
    int a = atomic_load(&x);
    int b = a * a;
    if(atomic_compare_exchange_weak(&x, &a, b))
        break;
}
```

3.9.2 Implémementation des *locks*

Un *spinlock* est le type de *lock* le plus simple qui existe : comme un *mutex*, il supporte deux opérations, que nous appelerons `lock` et `unlock`. À la différence d'un *mutex*, un *thread* qui tente

de prendre un *spinlock* déjà pris continue à utiliser du temps processeur tant que le *spinlock* n'est pas libéré.

Une implémentation naïve (fausse) d'un *spinlock* pourrait représenter ce dernier par un entier valant 0 s'il est libre et 1 s'il est pris (le choix est arbitraire) :

```
static int spinlock = 0;

void lock() {
    while(spinlock != 0)
        sched_yield();
    spinlock = 1;           // race condition!
}

void unlock() {
    spinlock = 0;
}
```

Comme le test et la modification dans la fonction `lock` ne s'effectuent pas de façon atomique, il y a une *race condition* dans la fonction `lock` : il se peut que deux *threads* testent simultanément le *spinlock*, le trouvent libre, et le prennent tous les deux. Commençons par récrire ce code pour qu'il fasse aparaître une opération CAS :

```
static int spinlock = 0;

void lock() {
    while(1) {
        int expected = 0;
        if(spinlock == expected) {     // race condition!
            spinlock = 1;
            break;
        }
        sched_yield();
    }
}

void unlock() {
    spinlock = 0;
}
```

On peut maintenant éviter la *race condition* à l'aide d'un CAS atomique :

```
static atomic_int spinlock = 0;

void lock() {
    while(1) {
        int rc, expected = 0;
```

```
            rc = atomic_compare_exchange_weak(&spinlock, &expected, 1);
            if(rc)
                break;
            sched_yield();
        }
    }

    void unlock() {
        atomic_store(&spinlock, 0);
    }
```

3.9.3 Digression : mémoire transactionnelle

Il est notoire que l'utilisation des primitives de synchronisation est difficile et sujette à erreur. Une technique expérimentale mais prometteuse qui est plus facile pour le programmeur est la *mémoire transactionnelle*, qui permet d'effectuer une série d'opérations sur la mémoire partagée de façon atomique, et de recommencer si l'opération échoue, un peu comme une transaction de bases de données. Il existe une proposition d'intégrer la mémoire transactionnelle à une future version de C++.

La syntaxe proposée est la suivante :

```
atomic {
    ...
}
```

Dans cette syntaxe, la mise au carré atomique d'une valeur, que nous avons implémentée ci-dessus à l'aide d'une boucle CAS, s'écrit tout simplement :

```
atomic {
    x = x * x;
}
```

mais des opérations plus complexes sont possibles :

```
atomic {
    if(x > 0) {
        x--;
        succes = 1;
    }
}
```

Il existe au moins deux techniques pour compiler les opérations de mémoire transactionnelle. La plus efficace compile un bloc `atomic` vers une boucle CAS qui effectue toutes les opérations sur des copies locales des variables puis effectue toutes les affectations du bloc de façon atomique ; cette technique recquiert du matériel qui supporte une opération de style CAS opérant sur un nombre arbitraire de locations (les processeurs Intel récents supportent une telle opération dans

l'extension *TSX*). L'autre technique, inspirée par les bases de données, accumule les écritures dans un journal, puis essaie de les effectuer toutes dans un bloc de code protégé par un *mutex* récursif global.

La mémoire transactionnelle n'est pas encore entièrement au point. En particulier, les implémentations actuelles ne sont pas très rapides, il n'est pas clair comment doit se comporter la mémoire transactionnelle en cas de congestion (comment s'assurer que la boucle CAS termine), et comment faire pour bloquer (quel est l'analogue transactionnel des variables de condition). Cependant, j'ai bon espoir qu'un jour la mémoire transactionnelle rendra une bonne partie de ce cours obsolète.

3.10 Signaux

On a souvent besoin de demander au système de nous notifier d'un événement. Par exemple, lorsqu'un processus fait un `read` ou `select` sur une *socket* ou un tube, il demande a être réveillé lorsque les données seront disponibles.

Toutes les notifications que nous avons vues jusque là sont *synchrones*, au sens qu'elles sont communiquées au processus lorsqu'il les demande[22]. Il est parfois utile de recevoir une notification *asynchrone*, dès qu'un événement a lieu même si on ne l'attend pas explicitement. Par exemple, lorsque l'utilisateur appuie sur *Control-C* pour interrompre un programme, ce-dernier doit réagir immédiatement même s'il est en attente dans un appel système bloquant ou en plein calcul dans une boucle.

Sous Unix, les notifications asynchrones sont représentées par un *signal*. À la différence des notifications synchrones, dont la destination est une structure de données explicite (une *socket*, un tube, etc.), un *signal* est destiné directement à un processus : on parle parfois de modèle *Actor*, par opposition au modèle *CSP*.

3.10.1 Quelques signaux utiles

La liste complète des signaux peut être obtenue à l'aide de la commande du *shell* `kill -l`. On peut citer en particulier :

- `SIGSEGV`, `SIGBUS` et `SIGILL` qui sont envoyés à un processus qui a effectué un accès illégal à la mémoire ou une opération illégale ;
- `SIGABRT`, qui est utilisé par la fonction `abort` ;
- `SIGINT`, `SIGQUIT`, `SIGTERM`, `SIGKILL` qui sont envoyés pour demander la terminaison d'un processus — en particulier, le pilote de clavier envoie `SIGINT` pour `^C` et `SIGQUIT` pour `^\` ;
- `SIGPIPE` qui est généré lors d'une écriture sur un tube fermé ;
- `SIGUSR1` et `SIGUSR2` qui sont réservés à l'utilisateur.

Chacun de ces signaux a une *action par défaut*, une action qui est effectuée par un processus qui le reçoit en l'absence de dispositions explicites. Les signaux `SIGINT` et `SIGPIPE` terminent le

22. Attention, le mot *synchrone* a ici un sens différent de celui qu'il a lorsqu'on parle de communication synchrone ou asynchrone.

programme par défaut, les autres signaux ci-dessus génèrent un *core dump* [23] puis terminent le programme. Voyez `man 7 signal` pour plus de détails.

3.10.2 Envoi de signaux

On peut envoyer un signal a un processus à l'aide de l'appel système `kill` :

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

La fonction `raise` envoie un signal au processus courant : un appel à `raise(signo)` est équivalent à `kill(getpid(), signo)`. Un appel à `abort` débloque le signal `SIGABRT` puis invoque `raise(SIGABRT)`, ce qui cause un *core dump* puis termine le programme. La macro `assert` invoque `abort` en cas d'échec.

3.10.3 Gestion des signaux

Traditionnellement, les signaux étaient capturés à l'aide de l'appel système `signal`. Cependant, `signal` n'est pas portable (sauf pour ignorer un signal à l'aide de `SIG_IGN`), et c'est `sigaction` qu'il faut utiliser.

L'appel système `sigaction` a le prototype suivant :

```
int sigaction(int signum, const struct sigaction *action,
              struct sigaction *oldaction);
```

Le paramètre `signum` est le numéro du signal à capturer. Le paramètre `action` décrit la nouvelle disposition du signal ; le paramètre `oldaction` vaudra `NULL` pour nous.

La structure `sigaction` contient au moins les champs suivants :

```
struct sigaction {
     void     (*sa_handler)(int);
     void     (*sa_sigaction)(int, siginfo_t *, void *);
     sigset_t   sa_mask;
     int        sa_flags;
 };
```

Le champ `sa_handler` définit l'action à effectuer ; c'est soit un pointeur sur un gestionnaire (une fonction prenant un paramètre entier), soit une des constantes `SIG_DFL` (action par défaut) ou `SIG_IGN` (ignorer le signal). Le champ `sa_flags` vaudra 0 dans ce cours, `sa_mask` vaudra 0 pour le moment, et `sa_sigaction` vaudra `NULL`.

Un exemple est donné dans la figure 3.11. Vous remarquerez que le gestionnaire utilise un appel à `write` plutôt que `printf` — les signaux sont asynchrones, et il n'est pas certain que les structures de données de `stdio` soient dans un état cohérent lorsque le gestionnaire est invoqué (paragraphe 3.10.5).

23. Une image de la mémoire stockée dans un fichier nommé `core` et exploitable par un déboggueur.

```
void
bennon_handler(int signo)
{
    write(1, "Ben non.\n", 9);
}

struct sigaction sa;
int rc;
memset(&sa, 0, sizeof(sa));
sa.sa_handler = bennon_handler;
sa.sa_flags = 0;
rc = sigaction(SIGINT, &sa, NULL);
if(rc < 0)
    ...
```

FIGURE 3.11 — Utilisation triviale de `sigaction`

3.10.4 Les signaux ne sont pas fiables

À la différence des appels système `write` et `sendmsg`, l'appel système `kill` ne bloque jamais. Le processus qui reçoit un signal ne maintient pas une file de signaux reçus de taille non-bornée, mais simplement une *bitmap* qui indique, pour chaque signal, si le signal a été reçu (et pas encore géré). De ce fait :

- les signaux peuvent être *dédupliqués* : si le même signal a été émis plusieurs fois, il peut n'être remis qu'une seule fois au destinataire ;
- les signaux peuvent être remis dans le désordre.

Ces propriétés ne sont pas un problème si les signaux sont utilisés par exemple pour demander la terminaison d'un processus, mais elles sont problématiques si on utilise les signaux comme une primitive de communication générale. Ne le faites donc pas.

3.10.5 Les signaux sont asynchrones

Un gestionnaire de signal peut s'exécuter à tout moment. De ce fait, un gestionnaire de signal ne doit appeler aucune fonction qui peut dépendre de la consistance de structures de données globales. Un gestionnaire de signal ne peut effectuer de façon fiable que les actions suivantes :

- lire et affecter une variable globale de type `volatile sig_atomic_t`, et
- appeler un petit nombre d'appels système et de fonctions documentées comme *async signal-safe* dans la norme POSIX.

Je ne connais que trois techniques pour écrire des gestionnaires de signal fiables : se limiter à appeler des fonctions *async signal-safe* dans les gestionnaires, convertir les notifications asynchrones en notifications synchrones à l'aide d'une variable globale, et convertir les notifications en notifications synchrones à l'aide de `sigblock`.

Gestionnaires simples

La norme POSIX documente un petit nombre de fonctions comme étant *async signal-safe* : elles peuvent être appelées dans un gestionnaire de signal asynchrone. Parfois, un gestionnaire de signal n'a pas besoin d'accéder aux structures de données du programme, et peut être entièrement écrit à l'aide de fonctions *async signal-safe*. C'est le cas dans le fragment de code de la figure 3.11.

Utilisation d'une variable globale

Si le programme est écrit à l'aide d'une boucle à événements, il passe régulièrement au même endroit de la boucle. Il est alors possible d'écrire un gestionnaire d'événement qui signalise la capture d'un signal à l'aide d'une variable globale, et c'est le programme principal qui s'occuppera d'effectuer l'action désirée, en contexte synchrone. Pour s'assurer de la cohérence de la variable qui sert à la communication, cette dernière doit être de type `volatile sig_atomic_t`.

Un exemple de cette technique est donné dans la figure 3.12.

Signaux bloqués

Il est possible de demander au système de retarder la livraison *(delivery)* d'un signal au processus. Un signal dont la livraison est retardée est dit *bloqué*, et il est délivré lorsqu'il est *débloqué*.

Il existe toute une zoologie d'appels système permettant de bloquer et débloquer les signaux. L'appel système portable est `sigprocmask`, qui manipule un ensemble de signaux représentés par un `sigset_t`.

Manipulation des ensembles de signaux Deux fonctions sont utiles pour manipuler les ensembles de signaux représentés par le type opaque `sigset_t` :

```
int sigemptyset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
```

La fonction `sigemptyset` initialise son paramètre à l'ensemble vide. La fonction `sigaddset` ajoute le signal `signum` à son paramètre `set`.

Blocage des signaux Un ensemble de signaux est bloqué ou débloqué à l'aide de `sigprocmask` :

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

Le paramètre `how` indique l'action à effectuer : s'il vaut `SIG_BLOCK`, les éléments de `set` sont ajoutés à l'ensemble de signaux bloqués ; s'il vaut `SIG_UNBLOCK`, ils sont supprimés, et s'il vaut `SIG_SETMASK`, l'ensemble des signaux bloqués est remplacé par `set`. Dans tous les cas, l'ancien ensemble de signaux bloqués est stocké dans `oldset` (qui peut valoir `NULL`).

La figure 3.13 donne un exemple artificiel d'utilisation de cet appel système ; nous en verrons un exemple plus convaincant au paragraphe 4.4.1.

```
volatile sig_atomic_t quitter = 0;

void
quit_handler(int signo)
{
    quitter = 1;
}

int
main()
{
    struct sigaction sa;
    int rc;

    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = quit_handler;
    sa.sa_flags = 0;
    rc = sigaction(SIGINT, &sa, NULL);
    if(rc < 0) abort();

    while(!quitter) {
        printf("J'ai pas encore fini.\n");
        sleep(1);
    }
    printf("J'ai fini.\n");
    return 0;
}
```

FIGURE 3.12 — Conversion d'un signal asynchrone en événement synchrone

```
volatile sig_atomic_t value = 42;

void
handler(int signo)
{
    char buf[40];
    int rc;
    rc = snprintf(buf, 40, "%d\n", (int)value);
    if(rc >= 0 && rc < 40)
        write(1, buf, rc);
}

int
main()
{
    struct sigaction sa;
    int rc;

    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = handler;
    rc = sigaction(SIGUSR1, &sa, NULL);
    if(rc < 0) abort();

    while(1) {
        sigset_t old, new;
        sigemptyset(&new);
        sigaddset(&new, SIGUSR1);
        rc = sigprocmask(SIG_BLOCK, &new, &old);
        if(rc < 0) abort();
        value = 57;
        sleep(1);
        value = 42;
        rc = sigprocmask(SIG_SETMASK, &old, NULL);
        if(rc < 0) abort();
    }

    return 0;
}
```

FIGURE 3.13 — Un exemple artificiel d'utilisation de signaux bloqués. Ce programme affiche toujours 42 lorsqu'il reçoit un signal `SIGUSR1`, avec au plus une seconde de délai.

3.11 Signaux et appels système bloquants

Lorsqu'un signal est délivré à un processus qui est dans un appel système bloquant, l'appel système retourne -1 avec `errno` valant `EINTR` [24]. Il est *essentiel* de gérer ce cas dans tout programme qui peut recevoir des signaux durant son fonctionnement normal.

Dans le cas le plus simple, il suffit de relancer l'appel système interrompu par `EINTR` :

```
do {
    rc = write(...);
} while(rc < 0 && errno == EINTR);
```

Dans la plupart des cas, cependant, il est désirable d'effectuer des actions (telles que vérifier la valeur des variables globales de type `sig_atomic_t`) avant de revenir dans l'appel système bloquant. Par exemple, dans une boucle à événements (paragraphe 4.4), on peut généralement traiter `EINTR` de la même façon que `EAGAIN`.

24. Il est possible de désactiver ce mécanisme en spécifiant `SA_RESTART` dans le champ `sa_flags` lors de l'appel à `sigaction`. En pratique, ce mécanisme est rarement utile, car on voudrait spécifier le comportement selon l'appel système, pas selon le signal.

4 Entrées-sorties non-bloquantes

Les appels système `read` et `write`, lorsqu'ils sont appliqués à des tubes (ou des périphériques, ou des *sockets*) peuvent *bloquer* pendant un temps indéfini. Cette sémantique est souvent celle qui convient, mais elle pose problème lorsqu'on veut appliquer des *time-outs* aux opérations d'entrées-sorties, ou lire des données sur plusieurs descripteurs de fichiers simultanément sans utiliser de *threads*.

Un descripteur de fichier peut être mis en mode *non-bloquant* en passant le *flag* `O_NONBLOCK` à l'appel système `open`. Une opération de lecture ou écriture sur un tel descripteur ne bloque jamais ; si elle devait bloquer, elle retourne -1 avec `errno` valant `EWOULDBLOCK` (« cette opération aurait bloqué »).

4.1 Mode non-bloquant

Les *flags* d'un descripteur obtenu à l'aide de l'appel système `open` sont affectés lors de l'ouverture. Souvent, cependant, les descripteurs sont obtenus à l'aide d'appels système qui ne permettent pas d'affecter les *flags*, notamment `pipe` et `socket`.

Les *flags* d'un descripteur existant peuvent être récupérés et affectés à l'aide de l'appel système `fcntl` :

```
int fcntl(int fd, F_GETFL);
int fcntl(int fd, F_SETFL, int value);
```

La première forme ci-dessus permet de récupérer les *flags* ; en cas de réussite, elle retourne les flags. La deuxième affecte les *flags* à la valeur passée en troisième paramètre.

Le *flag* `O_NONBLOCK`, qui nous intéresse, peut donc être positionné sur un descripteur de fichier `fd` à l'aide du fragment de code suivant :

```
rc = fcntl(fd, F_GETFL);
if(rc < 0)
    ...
rc = fcntl(fd, F_SETFL, rc | O_NONBLOCK);
if(rc < 0)
    ...
```

4.2 Attente active

Sur un descripteur non-bloquant, il est possible de simuler une lecture bloquante à l'aide d'une *attente active*, c'est à dire en vérifiant répétitivement si l'opération peut réussir sans bloquer :

```
while(1) {
    rc = read(fd, buf, 512);
    if(rc >= 0 || errno != EWOULDBLOCK && errno != EINTR)
        break;
    sched_yield();
}
```

Vous remarquerez au passage que cette attente active prend en compte la possibilité d'un appel système interrompu par un signal (voir cours précédent).

Une attente active est plus flexible qu'un appel système bloquant. Par exemple, une attente avec *time-out* s'implémente simplement en interrompant l'attente active au bout d'un certain temps :

```
debut = time(NULL);
while(1) {
    rc = read(fd, buf, 512);
    if(rc >= 0 || errno != EWOULDBLOCK || errno != EINTR)
        break;
    if(time(NULL) >= debut + 10)
        break;
    sched_yield();
}
```

De même, une lecture sur deux descripteurs de fichiers `fd1` et `fd2` peut se faire en faisant une attente active simultanément sur `fd1` et `fd2` :

```
while(1) {
    rc1 = read(fd1, buf, 512);
    if(rc1 >= 0 || errno != EWOULDBLOCK || errno != EINTR)
        break;
    rc2 = read(fd2, buf, 512);
    if(rc2 >= 0 || errno != EWOULDBLOCK || errno != EINTR)
        break;
    sched_yield();
}
```

Comme son nom indique, l'attente active utilise du temps de processeur pendant l'attente ; de ce fait, elle n'est pas utilisable en pratique, surtout sur les systèmes alimentés par batterie. Les mitigations qui consistent à céder le processeur pendant l'attente (à l'aide de `sched_yield` ou `usleep`) ne résolvent pas vraiment le problème.

4.3 L'appel système `select`

L'appel système `select` permet d'effectuer l'équivalent d'une attente active sans avoir besoin de boucler activement. L'appel `select` prend en paramètre des ensembles de descripteurs de fichiers (`fd_set`) et un *time-out* sous forme d'une structure `timeval`.

4.3.1 Mesure du temps

Nous avons déjà vu le type `time_t`, qui représente un temps en secondes, soit relatif, soit absolu (mesuré depuis l'Époque), et l'appel système `time`, qui retourne un temps absolu représenté comme un `time_t`.

La structure `timeval` 4BSD a introduit la structure `timeval`, qui représente un temps, absolu ou relatif, mesuré en secondes et micro-secondes :

```
struct timeval {
    time_t tv_sec;
    int tv_usec;
}
```

Le champ `tv_usec` représente les microsecondes, et doit être compris entre 0 et 999 999.

Le temps absolu peut être obtenu à l'aide de l'appel système `gettimeofday` :

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

Le paramètre `tv` pointe sur un tampon qui contiendra le temps courant après l'appel. Le paramètre `tz` est obsolète, et doit valoir `NULL`.

La structure `timespec` Les versions récentes de POSIX ont remplacé la structure `timeval` par la structure `timespec`, qui mesure le temps avec une granularité (mais pas forcément une précision) d'une nanoseconde :

```
struct timespec {
    time_t tv_sec;
    int tv_usec;
}
```

Le temps absolu peut être obtenu à l'aide de l'appel système `clock_gettime` :

```
int clock_gettime(clockid_t clk_id, struct timespec *tp);
```

Le paramètre `clk_id` identifie l'horloge à utiliser ; il vaut soit `CLOCK_REALTIME`, qui identifie l'horloge temps réel utilisée par `time` et `gettimeofday`, ou `CLOCK_MONOTONIC`, qui identifie une horloge dite *monotone*, qui mesure le temps depuis une origine arbitraire et qui ne recule jamais, même si l'administrateur change l'heure ou la date.

4.3.2 Ensembles de descripteurs

Un ensemble de descripteurs est représenté par le type opaque `fd_set`. Les opérations sur ce type sont :

- `FD_ZERO(fd_set *set)`, qui affecte l'enseble vide à `set` ;
- `FD_SET(int fd, fd_set *set)`, qui ajoute l'élément `fd` à `set` ;
- `FD_ISSET(int fd, fd_set *set)`, qui retourne vrai si `fd` est membre de `set`.

4.3.3 L'appel système `select`

L'appel système `select` permet d'attendre qu'un descripteur de fichier parmi un ensemble soit prêt à effectuer une opération d'entrées-sorties sans bloquer, ou qu'un intervalle de temps se soit écoulé.

```
int select(int nfds,
           fd_set *readfds, fd_set *writefds,  fd_set *exceptfds,
           struct timeval *timeout);
```

Les paramètres `readfds`, `writefds` et `exceptfds` sont les ensembles de descripteurs sur lesquels on attend, et `timeout` est un temps relatif qui borne le temps pendant lequel on attend. Le paramètre `nfds` doit être une borne exclusive des descripteurs de fichiers contenus dans les trois ensembles.

Un appel à `select` bloque justqu'à ce qu'une des conditions suivantes soit vraie :

- un des descripteurs contenus dans `readfds` est prêt pour une lecture ; ou
- un des descripteurs contenus dans `writefds` est prêt pour une écriture ; ou
- un des descripteurs contenus dans `exceptfds` est dans une situation exceptionnelle (hors programme pour ce cours) ; ou
- un temps égal ou supérieur à `timeout` s'est écoulé.

L'appel à `select` retourne 0 dans ce dernier cas, et le nombre de descripteurs prêts dans les autres cas. Les descripteurs prêts sont alors positionnés dans les trois ensembles.

Dans tous les cas, la valeur de `timeout` est indéfinie (arbitraire) après l'appel.

4.3.4 Exemples

Une lecture bloquante sur un descripteur se simule en bloquant d'abord à l'aide de `select`, puis en effectuant une lecture :

```
fd_set readfds;
FD_ZERO(&readfds);
FD_SET(fd, &readfds);
rc = select(fd + 1, &readfds, NULL, NULL, NULL);
if(rc > 0 && FD_ISSET(fd, &readfds))
    rc = read(fd, buf, 512);
```

Comme dans le cas d'une attente active, il est facile d'ajouter un *timeout* :

```
struct timeval tv = {10, 0};
fd_set readfds;
FD_ZERO(&readfds);
FD_SET(fd, &readfds);
rc = select(fd + 1, &readfds, NULL, NULL, &tv);
if(rc > 0 && FD_ISSET(fd, &readfds))
    rc = read(fd, buf, 512);
```

Il est tout aussi facile de généraliser cette attente à plusieurs descripteurs :

```
fd_set readfds;
FD_ZERO(&readfds);
FD_SET(fd1, &readfds);
FD_SET(fd2, &readfds);
rc = select((fd1 >= fd2 ? fd1 : fd2) + 1,
            &readfds, NULL, NULL, NULL);
if(rc > 0 && FD_ISSET(fd, &readfds))
    rc = read(fd, buf, 512);
```

4.3.5 Bug

La plupart des systèmes Unix ont le bug suivant : un appel à `select` peut parfois retourner un descripteur de fichier qui n'est pas prêt. De ce fait, il est nécessaire de mettre tous les descripteurs de fichier en mode non bloquant même si on ne fait jamais une opération bloquante que sur un descripteur qui a été retourné par `select`.

4.4 Boucles à événements

Avec l'appel système `select`, le dernier exemple du paragraphe 4.2 peut s'écrire comme sur la figure 4.1. Ce fragment de code a la structure suivante :

```
while(1) {
    Attendre un événement
    Gérer l'événement
}
```

Ce type de code s'appelle une *boucle à événements*. S'il est possible d'écrire les boucles à événement à la main, elles sont généralement implémentées dans une bibliothèque et le programmeur n'a qu'à fournir les gestionnaires d'événement. Par exemple, une interface graphique écrite en Java exécute une boucle à événements qui attend un événement de l'utilisateur (un clic de souris, etc.) puis invoque un gestionnaire *(handler)* fourni par le programmeur.

Il existe plusieurs bibliothèques de boucles à événements pour Unix, on m'a notamment dit du bien de `libev` et `libevent` [1].

4.4.1 Boucles à événements et signaux

Une boucle à événements a généralement besoin de gérer des signaux, par exemple pour faire le ménage avant de terminer le programme, pour afficher des informations (traditionnellement dédié à `SIGUSR1`) ou pour relire les fichiers de configuration (traditionnellement dédié à `SIGUSR2`). Une approche naïve à la gestion de `SIGINT` pourrait se faire comme dans la figure 4.2, en utilisant la technique de conversion en notification synchrone vue précédemment.

On remarque que ce programme souffre d'une *race condition* : si le gestionnaire de signal est exécuté entre le test de `please_quit` et l'entrée dans `select`, ce dernier bloque indéfiniment.

1. Windows a une implémentation de `select` terriblement inefficace, et ces bibliothèques sont donc très lentes sous Windows. Windows dispose des *ports de complétion*, qui, eux, passent à l'échelle de façon merveilleuse.

```
while(1) {
    fd_set fds;
    int rc;
    FD_ZERO(&fds);
    FD_SET(fd1, &fds);
    FD_SET(fd2, &fds);
    rc = select(max(fd1, fd2) + 1, &fds, NULL, NULL, NULL);
    if(rc < 0) {
        if(errno == EINTR)
            continue;
        perror("Aaargh");
        abort();
    }
    if(FD_ISSET(fd1, &fds) {
        int offset, rc2;
        unsigned char buf[512];
        rc = read(fd1, buf, 512);
        if(rc <= 0) {
             if(rc < 0 && errno == EINTR)
                 continue;
             break;
        }

        offset = 0;
        while(offset < rc) {
            rc2 = write(fd2, buf + offset, rc - offset);
            if(rc2 < 0) {
                if(errno == EINTR)
                    continue
                perror("Eek");
                abort();
            }
            offset += rc2;
        }
    }

    if(FD_ISSET(fd1, &fds) {
        ...
    }
}
```

FIGURE 4.1 — Une copie entre deux descripteurs en style boucle à événements

```
volatile sig_atomic_t please_quit = 0;

void
quit_handler(int signo)
{
    please_quit = 1;
}

...
memset(&sa, 0, sizeof(sa));
sa.sa_handler = quit_handler;
sa.sa_flags = 0;
rc = sigaction(SIGINT, &sa, NULL);
...
while(1) {
    fd_set fds;
    int rc;

    if(please_quit)
        break;

    /* Si le signal arrive ici, on bloque dans select. */

    FD_ZERO(&fds);
    FD_SET(fd1, &fds);
    FD_SET(fd2, &fds);
    rc = select(max(fd1, fd2) + 1, &fds, NULL, NULL, NULL);
    ...
}
```

FIGURE 4.2 — Boucle à événement avec gestion de signal. Ce programme souffre d'une *race condition* qui peut causer un blocage de durée non-bornée.

```
while(1) {
    fd_set fds;
    sigset_t old, new;
    int rc;

    sigemptyset(&new);
    sigaddset(&new, SIGINT);
    rc = sigprocmask(SIG_BLOCK, &new, &old);
    if(rc < 0) abort();

    if(please_quit)
        break;

    FD_ZERO(&fds);
    FD_SET(fd1, &fds);
    FD_SET(fd2, &fds);
    rc = pselect(max(fd1, fd2) + 1, &old,
                 &fds, NULL, NULL, NULL);
    ...
}
```

FIGURE 4.3 — Boucle à événement avec gestion de signal sans *race condition*.

Déblocage atomique des signaux

La solution au problème souligné ci-dessous consiste à bloquer les signaux d'intérêt avant de tester la variable globale. Mais comment peut-on les débloquer ?

- si on les débloque avant d'entrer dans `select`, la *race condition* existe encore ;
- si on les débloque après `select`, ce dernier ne sera pas interrompu en cas d'arrivée du signal.

Il est donc nécessaire de débloquer le signal et d'entrer dans `select` de façon atomique, ce qui est fait par l'appel système `pselect` :

```
int pselect(int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, const struct timespec *timeout,
            const sigset_t *sigmask);
```

Cet appel système est analogue à `select`, sauf qu'il effectue l'équivalent de `sigprocmask(SIG_SETMASK)` avant de bloquer ; si le signal débloqué était en attente, `pselect` retourne immédiatement -1 avec `errno` valant `EINTR`.

Le fragment de code de la figure 4.3 résoud la *race condition* du programme précédent.