Programmation système

Juliusz Chroboczek

septembre

Table des matières

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 la reécriture d'une quantité potentiellement grande de code ;
- le code n'est pas modulaire : l'accès au matériel est mélangé à la logique du programme elle-même.

De nos jours, 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 di érentes 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 su sant 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 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 .

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 e ectue 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 et 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 simple *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 par exemple souvent le cas de la mémoire et des consoles. 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*.

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 , servira de base à ce cours. La famille Unix a aujourd'hui de nombreux représentants, aussi bien libres (Linux, FreeBSD, NetBSD, Minix etc.) que moins libres (Solaris) et propriétaires (HP/UX, Mac OS X etc.).

1.2.1 Fonctions *stub*

Le mécanisme exact d'appel système dépend du matériel. Pour cette raison, chaque système Unix inclut dans la librairie C standard (`libc`) un certain nombre de fonctions dites *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 , et les « vraies » fonctions, qu'il documente dans la section .

Convention d'appel et variable `errno` La librairie 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 ; lorsqu'il échoue, elle stocke le numéro de l'erreur dans la variable `errno`, et retourne une valeur strictement négative .

La variabe `errno` ainsi que des constantes symboliques pour les codes d'erreurs sont déclarées dans le fichier d'entête `<errno.h>`. Pour a cher 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* e ectue les actions suivantes :

- stocker les arguments de l'appel dans les bons registres ;
- invoquer l'appel système ;
- interpréter la valeur de retour et, si besoin, positionner la variable `errno`.

1.2.2 Le standard POSIX

Au début des années , Unix s'était morcelé en un certain nombre de systèmes semblables mais incompatibles. L'écriture d'un programme portable entre les di érents dialectes d'Unix était devenue un exercice inutilement excitant.

Il y a quelques exceptions à cette convention.

Ce mécanisme, qui sert à compenser l'absence d'exceptions en C, pose certains problèmes pour l'écriture de code fiable en présence de *threads* multiples.

Le standard *IEEE 1003*, communément appelé *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 ce standard.

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 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. Typiquement, `time_t` est un alias pour `int`, défini par

```
typedef int time_t;
```

mais rien n'empêche une implémentation de faire, par exemple,

```
typedef long int time_t;
```

POSIX considère la di érence entre une fonction et un appel système comme étant un détail de l'implémentation, et ne di érencie donc pas entre les sections et du manuel.

Depuis , chaque version du standard POSIX est identique à une version de la *Single Unix Specification*, qui est disponible en ligne à l'adresse `http://www.unix.org/`.

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

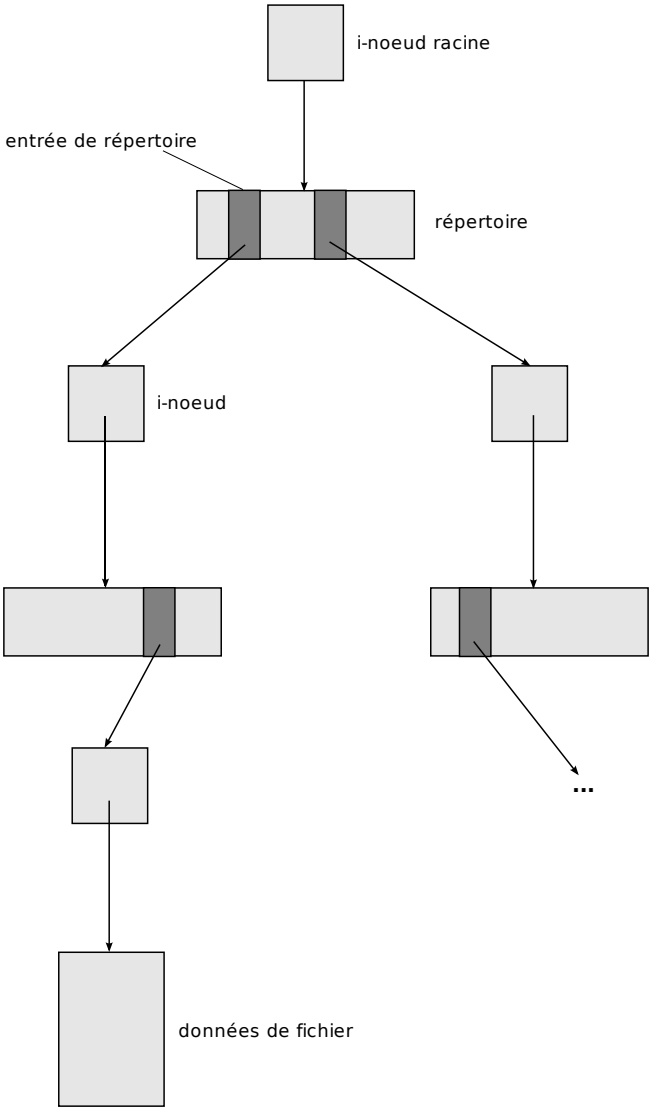


Figure . : Vision concrète du système de fichiers

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 su samment 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*.

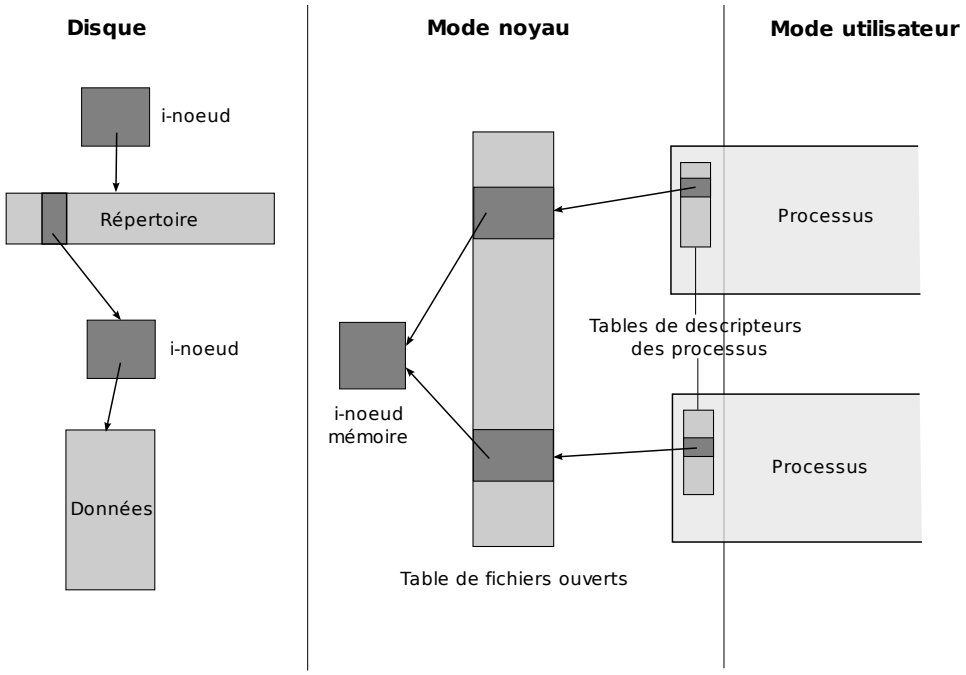


Figure . : Structures de données sur disque et en mémoire

Structures de données en mémoire Lorsque l'utilisateur demande au système de manipuler un fichier (en e ectuant 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 . 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 di érence 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 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 - 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_8 (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 e et 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 en cas de succès, et - en cas d'échec . 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).

Plus précisément, c'est la fonction *stub* correspondante qui est définie comme-ça.

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 e ectivement lus, à la fin du fichier, ou - 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 e ectivement écrits , ou - en cas d'erreur.

Exemple Le programme suivant e ectue une copie de fichiers. Comme nous le verrons, il est extrêmement ine cace.

```
#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);
    }

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

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*.

```
        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 e ectue deux appels système pour chaque octet copié — ce qui est tragique. Mon portable est capable d'e ectuer jusqu'à d'appels système par seconde ; ce programme ne sera donc pas capable de copier plus d' , Mo de données par seconde environ.

2.3 Tampons

Pour résoudre le problème de performances soulevé ci-dessus, il su t d'e ectuer 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

Un tampon simple (fig. .) 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;
```

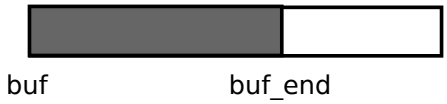


Figure . : Tampon simple

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'e ectue 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

Figure . : Tampon avec pointeur de début des données

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

```
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, une fois qu'on a commencé à écrire les données, le tampon doit être vidé avant qu'on 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

Figure . : 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.

Un *tampon circulaire* (fig. .) consiste des mêmes données qu'un tampon avec pointeur, mais les deux pointeurs sont interprétés modulo la taille du tampon. Lorsque `buf_end` > `buf_ptr`,

Ÿ # p@r4 E4 Pð P O

les données valides ne sont plus connexes, mais constituées des deux parties `buf_ptr` à `BUFFER_SIZE` et 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 di érencier 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

B M F

Le champ `flags` indique l'état du flot. Il vaut normalement ; 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 vraie implémentation de `stdio`, le champ `flags` contiendra aussi de l'information qui indique la politique de gestion du tampon associé au flot (voir partie . . 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 di érentes sont possibles pour décider quand ce tampon est vidé, i.e. quand l'appel système e ectuant la sortie e ective est e ectué. 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;
}
```

Exercice: écrivez les fonctions `_filbuf` et `_flshbuf`.

Comme `stdio` utilise des tampons, il est possible de faire toutes les entrées/sorties avec ces fonctions. 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 plutôt que sur des tableaux d'octets :

```
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'é[illegible]re, tandis que s[illegible]e indique la taille de chaque élément ; le nombre total d'octets lus [illegible] écrits est donc `nmemb * s[illegible]e`.

Exercice: écrivez les fonctions `fread` et `fwrite`, d'a[illegible]d en termes de `getc` et `pu[illegible]`, ensuite en manipulant directement les tampons.

2.4.3 Interaction entre `stdio` et le système

Au démarrage du programme, `stdio` crée trois flo[illegible]ssociés aux descripteurs de[illegible]ier , et , et les a ecte [illegible] et `set` [illegible] `stdio`

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 e ectue 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 .

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 C , elle retourne - , dans les implémentations C , elle retourne la taille du tampon dont elle aurait eu besoin. Cette di érence entre les versions doit être gérée par le programmeur :

```
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;
    }
}
```

Inversement, la fonction `sscanf` e ectue l'analyse :

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

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

Entrées/sorties formatées Les fonctions de formatage et d'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.

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 .). Lors de l'appel système `open`, le pointeur de position courante est initialisé à , 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 `-` en cas d'erreur (et alors `errno` est positionné).

> **Attention :** le type `off_t` est d'habitude un synonyme de `long`, et pas de `int` ; 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 di cile :

```
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 e et 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 octets. Le processus A commence par se positionner à la fin du fichier, soit à la position . Supposons maintenant qu'un changement de contexte intervient à ce moment, le contrôle passe au processus B, qui se positionne à la position , et y écrit un octet « b ». Le contrôle repasse ensuite au processus A, qui est toujours positionné en ; il écrit donc un octet « a » à la position , 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 e ectuent 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 drapeau `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 e ectuée 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 par le noyau. Par exemple, la séquence suivante ajoute à la fin du fichier 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 « `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` :

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

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 en cas de succès, - en cas d'erreur (et alors `errno`… bon, vous avez compris).

`O_TRUNC` Il est très courant de vouloir tronquer un fichier à 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 di érence 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.

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

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, 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 di érent 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 . Ils remplissent le tampon `buf` avec les informations de l'i-nœud, puis retournent en cas de succès, - 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
```

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

```
        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`.

Localisation de l'i-nœud Les champs `st_dev` et `st_ino` cont

– `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 . . ci-dessous). Ce champ est parfois appelé *temps de création du fichier*.

Normalement, c'est `st_mtime` qui est intéressant ; l'auteur de ces lignes avoue qu'il ne s'est jamais encore 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 octets sous la 7e édition, sous Unix BSD et sous Linux, mais elle vaut 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.

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`. 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 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` di èrent dans la façon dont est identifié l'i-nœud à modifier : `chmod` prend un nom de fichier, `fchmod` un descripteur. Ces appels retournent en cas de succès, - 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 d'un fichier peuvent être changés à l'aide de l'appel système suivant :

```
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. (Le temps de dernère modification de l'i-nœud est mis à jour par cet appel.)

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 di érents.

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);
    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 e ectué, 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.

Aujourd'hui, Unix implémente un appel système `rename` :

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

Cet appel système ne peut pas échouer partiellement. À la di érence 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, son *compte de références*, qui est égal au nombre de liens existant vers cet i-nœud. Ce champ vaut initialement , il est incrémenté à chaque appel `link`, et décrémenté à chaque appel `unlink`. Lorsqu'il atteint , 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) 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 e ectuée 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*, 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. 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.

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 . .).

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 a che les noms de fichier qu'il y trouve.

Les répertoires en 7e édition

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

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

```
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 octets lus était interprétée comme une structure de type `dirent`.

Interface de haut niveau

Sous . BSD, 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 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 (e édition), le champ `d_ino` contient un numéro d'inœud, et le champ `d_name` le nom d'un fichier. À la di érence 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

3 Processus

Un *programme* est un fichier contenant une suite d'instructions exécutables et les données associées. Lorsque ce programme est *exécuté*, il est d'abord chargé en mémoire puis exécuté par le processeur.

Un *processus* est une instance d'un programme en train de s'exécuter. Informellement, un processus c'est une zone mémoire et un flot de contrôle. Plus précisément, un processus consiste de :

- une zone de mémoire contenant le code exécutable (le segment de *texte*), les variables globales (le segment de *données*), les variables locales et la chaîne d'activation des fonctions (la *pile*) et les données allouées dynamiquement (le *tas*) ;
- un *contexte*, c'est à dire l'ensemble des registres du processeur, notamment celui qui indique la prochaine instrucution à exécuter et celui qui indique le sommet de la pile ;
- une petite structure du noyau, contenant des informations supplémentaires à propos du processus, tels que son numéro, son propriétaire et son répertoire courant.

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.

3.1 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 . Il s'agit là d'un autre cas de *virtualisation*, cette fois-ci du processeur.

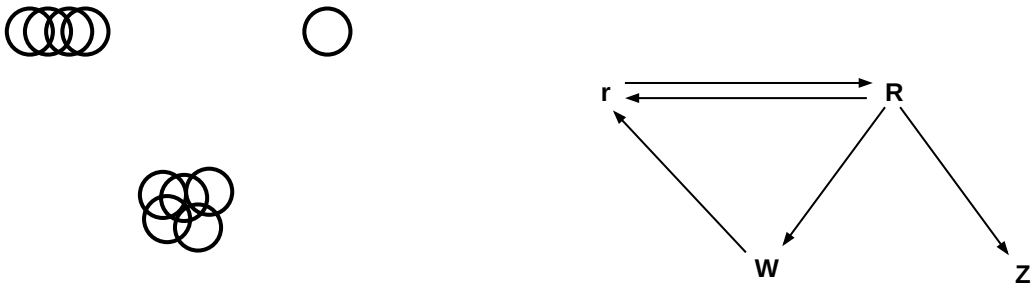
La partie du noyau qui choisit quel processus doit s'exécuter à un moment donné s'appelle *l'ordonnanceur (scheduler)*. L'ordonnanceur maintient les structures de données suivantes (figure . (a)) :

- 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 (W) ;
- un ensemble de processus morts, les *zombies* (Z).

Un processus peut donc avoir les états suivants (Figure . (b)) :

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

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



(a) (b)

Figure . : Structures de données de l'ordonnanceur (a) et transitions des processus (b)

– 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 :

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

3.1.1 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 e ectuer 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 actuelle immédiatement et

")

3.2 Contenu d'un processus Unix

En plus de la mémoire du processus et de son contexte, un processus Unix contient une composante en mode noyau 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 « e ectif », *euid* et *egid* ;
- le répertoire courant du processus ;
- l'état du processus.

3.2.1 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'a che 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 a che 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*).

3.2.2 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 .

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 : 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 :

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

```
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 di ère entre BSD et SYSV.

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 . .). 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 e ectuant une résolution de noms. Son contenu peut notamment être lu en passant le nom `"."` à l'appel système `stat` ou à la fonction `opendir`.

3.3 Vie et mort des processus

Intuitivement, la création d'un processus s'accompagne de l'exécution d'un programme ; cependant, sous Unix , 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`.

3.3.1 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 e et de dupliquer le processus courant. Un appel à `fork` retourne deux fois : une fois dans le père, où il retourne le *pid* du fils nouvellement créé, et une fois dans le fils, où il retourne .

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

Mais pas sous Windows ou MS-DOS.

Au contraire d'un appel à `_exit`, qui retourne 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 - 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 - , `waitpid` attend la mort de n'importe quel fils (comme `wait`). Le paramètre `flags` peut avoir les valeurs suivantes :

- : 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 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 .

3.3.2 Mort d'un processus

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

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

Lorsqu'un processus e ectue un appel à `_exec`, 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 . . sur la relation entre `exit` et `return`.

3.4 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.

3.4.1 Fonctions utilitaires

L'appel système `execve` n'est pas toujours pratique à utiliser. La librairie standard contient un certain nombre de *wrappers* 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, ...);
```

3.4.2 Parenthèse : `_start`

Lorsqu'un programme est exécuté, la première fonction qui s'exécute est la fonction `_start` de la bibliothèque C. Cette fonction e ectue 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.

3.5 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 :

Vous voyez une bonne traduction ?

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

3.5.1 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 souvent utilisée consiste à deshériter 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 — hors programme pour ce cours — permettant d'obtenir le même résultat, qui consiste à ignorer le signal `SIGCHLD`.

3.6 Redirections et tubes

3.6.1 Descripteurs standard

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

– le descripteur de fichier 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 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 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.

3.6.2 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 . . 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 .).

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.

3.6.3 Tubes

Comme nous l'avons vu en cours, la communication entre deux processus peut se faire à travers un fichier. Une telle approche demande un mécanisme supplémentaire de synchronisation (par exemple `wait`), et risque de laisser des fichiers temporaires sur le disque.

Un *tube* consiste de deux descripteurs de fichier, appelés le *bout écriture* et le *bout lecture* du tube, connectés à travers un tampon. Une donnée écrite sur le bout écriture est stockée dans le tampon, et une lecture retourne les données contenues dans le tampon.

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]`. La taille du tampon interne du tube n'est pas spécifiée, mais POSIX garantit qu'elle fait au moins octets.

Sémantique des entrées-sorties sur les tubes Les lectures sur les tubes ont la sémantique suivante :

- un appel système `read` peut retourner un nombre d'octets strictement positif mais inférieur à celui demandé même si on n'a pas atteint une fin de fichier (on parle alors de *lecture partielle*) ;
- un appel système `read` retourne (indication de fin de fichier) si l'écrivain a fermé le tube et celui-ci est vide ;
- un appel système `read` e ectué sur un tube vide bloque.

Les écritures ont la sémantique suivante :

- un appel système `write` peut écrire un nombre d'octets strictement positif mais inférieur à celui demandé (on parle alors d'*écriture partielle*) ;
- un appel système `write` e ectué sur un tube que le lecteur a fermé tue le processus avec un signal `SIGPIPE` ;
- un appel système `write` e ectué sur un tube plein bloque.

Cette sémantique est plus relâchée que celle des entrées-sorties sur disque, où les entrées/sorties ne bloquent jamais, où une lecture partielle n'est possible qu'à la fin du fichier, et où une écriture partielle est impossible. En pratique, cela signifie qu'on ne pourra pas utiliser un tampon simple avec les tubes, il faudra utiliser un tampon avec pointeur de début ou un tampon circulaire (voir partie .).

3.7 Exemple

Le fragment de code suivant combine les plus importants des appels système vus dans cette partie :

```
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? */
```

Ce qu'on peut éviter en ignorant `SIGPIPE` ; dans ce cas, le `write` retourne - avec `errno` valant `EPIPE`.

```
    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? */
}
```

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.

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 - 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 a ectés lors de l'ouverture. Souvent, cependant, les descripteurs sont obtenus à l'aide d'appels système qui ne permettent pas d'a ecter les *flags*, notamment l'appel système `pipe` et, comme nous le verrons au deuxième semestre, l'appel système `socket`.

Les *flags* d'un descripteur existant peuvent être récupérés et a ectés 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 a ecte 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)
        break;
}
```

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)
        break;
    if(time(NULL) >= debut + 10)
        break;
}
```

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)
        break;
    rc2 = read(fd2, buf, 512);
    if(rc2 >= 0 || errno != EWOULDBLOCK)
        break;
}
```

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. 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'e ectuer 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 La structure `timeval`

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`.

BSD 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 micro-secondes, et doit être compris entre et .

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`.

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

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 e ectuer 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 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 e ectuant une lecture :

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

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

```
struct timeval tv = {10, 0};
fd_set readfds;
FD_ZERO(&readfds);
FD_SET(fd, &readfds);
rc = select(fd + 1, &readfds, NULL, NULL, &tv);
if(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(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 faire une opération bloquante que sur un descripteur qui a été retourné par `select`.