SINF1252 : Systèmes informatiques

Version 2014

O. Bonaventure, G. Detal, C. Paasch

25 May 2015

Table des matières

1 Introduction **1**
1.1 Introduction . . . 1

2 Langage C **9**
2.1 Le langage C . . . 9
2.2 Types de données . . . 15
2.3 Déclarations . . . 34
2.4 Unions et énumérations . . . 36
2.5 Organisation de la mémoire . . . 38
2.6 Compléments de C . . . 50

3 Structure des ordinateurs **61**
3.1 Organisation des ordinateurs . . . 61
3.2 Etude de cas : Architecture [IA32] . . . 67

4 Systèmes Multiprocesseurs **81**
4.1 Utilisation de plusieurs threads . . . 81
4.2 Communication entre threads . . . 88
4.3 Coordination entre threads . . . 92
4.4 Les sémaphores . . . 107
4.5 Compléments sur les threads POSIX . . . 116
4.6 Loi de Amdahl . . . 122
4.7 Les processus . . . 123

5 Fichiers **143**
5.1 Gestion des utilisateurs . . . 143
5.2 Systèmes de fichiers . . . 144
5.3 Signaux . . . 159
5.4 Sémaphores nommés . . . 171
5.5 Partage de fichiers . . . 173

6 Mémoire virtuelle **181**
6.1 La mémoire virtuelle . . . 181
6.2 Annexes . . . 203

Bibliographie **209**

Index **213**

CHAPITRE 1

Introduction

1.1 Introduction

Les systèmes informatiques jouent un rôle de plus en plus important dans notre société. En une septantaine d'années les ordinateurs se sont rapidement améliorés et démocratisés. Aujourd'hui, notre société est de plus en plus dépendante des systèmes informatiques.

1.1.1 Composition d'un système informatique

Le système informatique le plus simple est composé d'un *processeur* (*CPU* en anglais) ou unité de calcul et d'une mémoire. Le processeur est un circuit électronique qui est capable d'effectuer de nombreuses tâches :

- lire de l'information en mémoire
- écrire de l'information en mémoire
- réaliser des calculs

L'architecture des ordinateurs est basée sur l'architecture dite de Von Neumann. Suivant cette architecture, un ordinateur est composé d'un processeur qui exécute un programme se trouvant en mémoire. La mémoire contient à la fois le programme à exécuter et les données qui sont manipulées par le programme.

L'élément de base pour stocker et représenter de l'information dans un système informatique est le *bit*. Un bit (binary digit en anglais) peut prendre deux valeurs qui par convention sont représentées par :

- `1`
- `0`

Physiquement, un bit est représenté sous la forme d'un signal électrique ou optique lorsqu'il est transmis et d'une charge électrique ou sous forme magnétique lorsqu'il est stocké. Nous n'aborderons pas ces détails technologiques dans le cadre de ce cours. Ils font l'objet de nombreux cours d'électronique.

Le bit est l'unité de base de stockage et de transfert de l'information. En général, les systèmes informatiques ne traitent pas des bits individuellement [1].

La composition de plusieurs bits donne lieu à des blocs de données qui peuvent être utiles dans différentes applications informatiques. Ainsi, un *nibble* est un bloc de 4 bits consécutifs tandis qu'un *octet* (ou *byte* en anglais) est un bloc de 8 bits consécutifs. On parlera de mots (word en anglais) pour des groupes comprenant généralement 32 bits et de long mot pour des groupes de 64 bits.

Le processeur et la mémoire ne sont pas les deux seuls composants d'un système informatique. Celui-ci doit également pouvoir interagir avec le monde extérieur, ne fut-ce que pour pouvoir charger le programme à exécuter et les données à analyser. Cette interaction se réalise grâce à un grand nombre de dispositifs d'entrées/sorties et de stockage. Parmi ceux-ci, on peut citer :

- le clavier qui permet à l'utilisateur d'entrer des caractères
- l'écran qui permet à l'utilisateur de visualiser le fonctionnement des programmes et les résultats qu'ils produisent
- l'imprimante qui permet à l'ordinateur d'écrire sur papier les résultats de l'exécution de programmes

1. Dans certaines applications, par exemple dans les réseaux informatiques, il peut être utile d'accéder à la valeur d'un bit particulier qui joue par exemple le rôle d'un drapeau. Celui-ci se trouve cependant généralement à l'intérieur d'une structure de données comprenant un ensemble de bits.

— le disque-dur, les clés USB, les CDs et DVDs qui permettent de stocker les données sous la forme de fichiers et de répertoires
— la souris, le joystick ou la tablette graphique qui permettent à l'utilisateur de fournir à l'ordinateur des indications de positionnement
— le scanner qui permet à l'ordinateur de transformer un document en une image numérique
— le haut-parleur avec lequel l'ordinateur peut diffuser différentes sortes de son
— le microphone et la caméra qui permettent à l'ordinateur de capturer des informations sonores et visuelles pour les stocker ou les traiter

1.1.2 Unix

Unix est aujourd'hui un nom générique [2] correspondant à une famille de systèmes d'exploitation. La première version de Unix a été développée pour faciliter le traitement de documents sur mini-ordinateur.

Quelques variantes de Unix

De nombreuses variantes de Unix ont été produites durant les quarante dernières années. Il est impossible de les décrire toutes, mais en voici quelques unes.

— *Unix*. Initialement développé aux Bell Laboratories d'AT&T, Unix a été ensuite développé par d'autres entreprises. C'est aujourd'hui une marque déposée par `The Open group`, voir http://www.unix.org/
— *BSD Unix*. Les premières versions de Unix étaient librement distribuées par Bell Labs. Avec le temps, des variantes de Unix sont apparues. La variante développée par l'université de Berkeley en Californie a été historiquement importante car c'est dans cette variante que de nombreuses innovations ont été introduites dont notamment les implémentations des protocoles TCP/IP utilisés sur Internet. Aujourd'hui, *FreeBSD* et *OpenBSD* sont deux descendants de *BSD Unix*. Ils sont largement utilisés dans de nombreux serveurs et systèmes embarqués. *MacOS*, développé par Apple, s'appuie fortement sur un noyau et des utilitaires provenant de *FreeBSD*.
— *Minix* est un système d'exploitation développé initialement par *Andrew Tanenbaum* à l'université d'Amsterdam. *Minix* est fréquemment utilisé pour l'apprentissage du fonctionnement des systèmes d'exploitation.
— *Linux* est un noyau de système d'exploitation largement inspiré de *Unix* et *Minix*. Développé par *Linus Torvalds* durant ses études d'informatique, il est devenu la variante de Unix la plus utilisée à travers le monde. Il est maintenant développé par des centaines de développeurs qui collaborent via Internet.
— *Solaris* est le nom commercial de la variante Unix d'Oracle.

Dans le cadre de ce cours, nous nous focaliserons sur le système *GNU/Linux*, c'est-à-dire un système qui intègre le noyau *Linux* et les librairies et utilitaires développés par le projet *GNU* de la *FSF*.

Un système Unix est composé de trois grands types de logiciels :

— Le noyau du système d'exploitation qui est chargé automatiquement au démarrage de la machine et qui prend en charge toutes les interactions entre les logiciels et le matériel.
— De nombreuses librairies qui facilitent l'écriture et le développement d'applications
— De nombreux programmes utilitaires précompilés qui peuvent résoudre un grand nombre de problèmes courants. Certains de ces utilitaires sont chargés automatiquement lors du démarrage de la machine. La plupart sont exécutés uniquement à la demande des utilisateurs.

Le rôle principal du noyau du système d'exploitation est de gérer les ressources matérielles (processeur, mémoire, dispositifs d'entrées/sorties et de stockage) de façon à ce qu'elles soient accessibles à toutes les applications qui s'exécutent sur le système. Gérer les ressources matérielles nécessite d'inclure dans le systèmes d'exploitation des interfaces programmatiques (Application Programming Interfaces - *API*) qui facilitent leur utilisation par les applications. Les dispositifs de stockage sont une belle illustration de ce principe. Il existe de nombreux dispositifs de stockage (disque dur, clé USB, CD, DVD, mémoire flash, ...). Chacun de ces dispositifs a des caractéristiques électriques et mécaniques propres. Ils permettent en général la lecture et/ou l'écriture de blocs de données de quelques centaines d'octets. Nous reviendrons sur leur fonctionnement ultérieurement. Peu d'applications sont

2. Formellement, Unix est une marque déposée par l'Open Group un ensemble d'entreprises qui développent des standards dans le monde de l'informatique. La première version de Unix écrite en C date de 1973, http://www.unix.org/what_is_unix/history_timeline.html

capables de piloter directement de tels dispositifs pour y lire ou y écrire de tels blocs de données. Par contre, la majorité des applications sont capables de les utiliser par l'intermédiaire du système de fichiers. Le système de fichiers (arborescence des fichiers) et l'API associée (open(2), close(2), read(2) write(2)) sont un exemple des services fournis par le système d'exploitation aux applications. Le système de fichiers regroupe l'ensemble des fichiers qui sont accessibles depuis un système sous une arborescence unique, quel que soit le nombre de dispositifs de stockage utilisé. La racine de cette arborescence est le répertoire / par convention. Ce répertoire contient généralement une dizaine de sous répertoires dont les noms varient d'une variante de Unix à l'autre. Généralement, on retrouve dans la racine les sous-répertoires suivants :

- `/usr` : sous-répertoire contenant la plupart des utilitaires et librairies installées sur le système
- `/bin` et `/sbin` : sous-répertoire contenant quelques utilitaires de base nécessaires à l'administrateur du système
- `/tmp` : sous-répertoire contenant des fichiers temporaires. Son contenu est généralement effacé au redémarrage du système.
- `/etc` : sous-répertoire contenant les fichiers de configuration du système
- `/home` : sous-répertoire contenant les répertoires personnels des utilisateurs du système
- `/dev` : sous-répertoire contenant des fichiers spéciaux
- `/root` : sous-répertoire contenant des fichiers propres à l'administrateur système. Dans certains variantes de Unix, ces fichiers sont stockés dans le répertoire racine.

Un autre service est le partage de la mémoire et du processus. La plupart des systèmes d'exploitation supportent l'exécution simultanée de plusieurs applications. Pour ce faire, le système d'exploitation partage la mémoire disponible entre les différentes applications en cours d'exécution. Il est également responsable du partage du temps d'exécution sur le ou les processeurs de façon à ce que toutes les applications en cours puissent s'exécuter.

Unix s'appuye sur la notion de processus. Une application est composée de un ou plusieurs processus. Un processus peut être défini comme un ensemble cohérent d'instructions qui utilisent une partie de la mémoire et sont exécutées sur un des processeurs du système. L'exécution d'un processus est initiée par le système d'exploitation (généralement suite à une requête faite par un autre processus). Un processus peut s'exécuter pendant une fraction de secondes, quelques secondes ou des journées entières. Pendant son exécution, le processus peut potentiellement accéder aux différentes ressources (processeurs, mémoire, dispositifs d'entrées/sorties et de stockage) du système. A la fin de son exécution, le processus se termine [3] et libère les ressources qui lui ont été allouées par le système d'exploitation. Sous Unix, tout processus retourne au processus qui l'avait initié le résultat de son exécution qui est résumée en un nombre entier. Cette valeur de retour est utilisée en général pour déterminer si l'exécution d'un processus s'est déroulée correctement (zéro comme valeur de retour) ou non (valeur de retour différente de zéro).

Dans le cadre de ce cours, nous aurons l'occasion de voir en détails de nombreuses librairies d'un système Unix et verrons le fonctionnement d'appels systèmes qui permettent aux logiciels d'interagir directement avec le noyau. Le système Unix étant majoritairement écrit en langage C, ce langage est le langage de choix pour de nombreuses applications. Nous le verrons donc en détails.

Utilitaires

Unix a été conçu à l'époque des mini-ordinateurs. Un mini-ordinateur servait plusieurs utilisateurs en même temps. Ceux-ci y étaient connectés par l'intermédiaire d'un terminal équipé d'un écran et d'un clavier. Les programmes traitaient les données entrées par l'utilisateur via le clavier ou stockées sur le disque. Les résultats de l'exécution des ces programmes étaient affichés à l'écran, sauvegardés sur disque ou parfois imprimés sur papier. Le fonctionnement de nombreux utilitaires Unix a été influencé par cet environnement. A tout processus Unix, on associe :

- une entrée standard (*stdin* en anglais) qui est un flux d'informations par lequel le processus reçoit les données à traiter. Par défaut, l'entrée standard est associée au clavier.
- une sortie standard (*stdout* en anglais) qui est un flux d'informations sur lequel le processus écrit le résultat de son traitement. Par défaut, la sortie standard est associée au terminal.
- une sortie d'erreur standard (*stderr* en anglais) qui est un flux de données sur lequel le processus écrira les messages d'erreur éventuels. Par défaut, la sortie d'erreur standard est associée à l'écran.

Unix ayant été initialement développé pour manipuler des documents contenant du texte, il comprend de nombreux utilitaires facilitant ces traitements. Une description détaillée de l'ensemble de ces utilitaires sort du cadre de ce

3. Certains processus sont lancés automatiquement au démarrage du système et ne se terminent qu'à son arrêt. Ces processus sont souvent appelés des *daemons*. Il peut s'agir de services qui fonctionnent en permanence sur la machine, comme par exemple un serveur web ou un *daemon* d'authentification.

cours. De nombreux livres et ressources Internet fournissent une description détaillée. Voici cependant une brève présentation de quelques utilitaires de manipulation de texte qui peuvent s'avérer très utiles en pratique.

- cat(1) : utilitaire permettant notamment d'afficher le contenu d'un fichier sur la sortie standard
- echo(1) : utilitaire permettant d'afficher sur la sortie standard une chaîne de caractères passée en argument
- head(1) et tail(1) : utilitaires permettant respectivement d'extraire le début ou la fin d'un fichier
- wc(1) : utilitaire permettant de compter le nombre de caractères et de lignes d'un fichier
- grep(1) : utilitaire permettant notamment d'extraire d'un fichier texte les lignes qui contiennent ou ne contiennent pas une chaîne de caractères passée en argument
- sort(1) : utilitaire permettant de trier les lignes d'un fichier texte
- uniq(1) : utilitaire permettant de filtrer le contenu d'un fichier texte afin d'en extraire les lignes qui sont dupliquées
- more(1) : utilitaire permettant d'afficher page par page un fichier texte sur la sortie standard (less(1) est une variante courante de more(1))
- gzip(1) et gunzip(1) : utilitaires permettant respectivement de compresser et de décompresser des fichiers. Les fichiers compressés prennent moins de place sur le disque que les fichiers standard et ont par convention un nom qui se termine par `.gz`.
- tar(1) : utilitaire permettant de regrouper plusieurs fichiers dans une archive. Souvent utilisé en combinaison avec gzip(1) pour réaliser des backups ou distribuer des logiciels.
- sed(1) : utilitaire permettant d'éditer, c'est-à-dire de modifier les caractères présents dans un flux de données.
- awk(1) : utilitaire incluant un petit langage de programmation et qui permet d'écrire rapidement de nombreux programmes de manipulation de fichiers textes

La plupart des utilitaires fournis avec un système Unix ont été conçus pour être utilisés en combinaison avec d'autres. Cette combinaison efficace de plusieurs petits utilitaire est un des points forts des systèmes Unix par rapport à d'autres systèmes d'exploitation.

Shell

Avant le développement des interfaces graphiques telles que *X11*, *Gnome*, *CDE* ou *Aqua*, l'utilisateur interagissait exclusivement avec l'ordinateur par l'intermédiaire d'un interpréteur de commandes. Dans le monde Unix, le terme anglais *shell* est le plus souvent utilisé pour désigner cet interpréteur et nous ferons de même. Avec les interfaces graphiques actuelles, le shell est accessible par l'intermédiaire d'une application qui est généralement appelée `terminal` ou `console`.

Un *shell* est un programme qui a été spécialement conçu pour faciliter l'utilisation d'un système Unix via le clavier. De nombreux shells Unix existent. Les plus simples permettent à l'utilisateur de taper une série de commandes à exécuter en les combinant. Les plus avancés sont des interpréteurs de commandes qui supportent un langage complet permettant le développement de scripts plus ou moins ambitieux. Dans le cadre de ce cours, nous utiliserons bash(1) qui est un des shells les plus populaires et les plus complets. La plupart des commandes bash(1) que nous utiliserons sont cependant compatibles avec de nombreux autres shells tels que zsh ou csh.

Lorsqu'un utilisateur se connecte à un système Unix, en direct ou à travers une connexion réseau, le système vérifie son mot de passe puis exécute automatiquement le shell qui est associé à cet utilisateur depuis son répertoire par défaut. Ce shell permet à l'utilisateur d'exécuter et de combiner des commandes. Un shell supporte deux types de commande : les commandes internes qu'il implémente directement et les commandes externes qui font appel à un utilitaire stocké sur disque. Les utilitaires présentés dans la section précédente sont des exemples de commandes externes. Voici quelques exemples d'utilisation de commandes externes.

```
$ cat exemple.txt
Un simple fichier de textes
aaaaaaaaaa bbbbbb
bbbbb ccccccccccc
eeeee fffffffff
aaaaaaaaaa bbbbbb
$ grep fichier exemple.txt
Un simple fichier de textes
$ wc exemple.txt
       5      13      98 exemple.txt
```

La puissance du *shell* ne vient pas de sa capacité d'exécuter des commandes individuelles telles que ci-dessus. Elle vient de la possibilité de combiner ces commandes en redirigeant les entrées et sorties standards. Les shells Unix supportent différentes formes de redirection. Tout d'abord, il est possible de forcer un programme à lire son entrée standard depuis un fichier plutôt que depuis le clavier. Cela se fait en ajoutant à la fin de la ligne de commande le caractère < suivi du nom du fichier à lire. Ensuite, il est possible de rediriger la sortie standard vers un fichier. Cela se fait en utilisant > ou >>. Lorsqu'une commande est suivie de `> file`, le fichier `file` est créé si il n'existait pas et remis à zéro si il existait et la sortie standard de cette commande est redirigée vers le fichier `file`. Lorsqu'un commande est suivie de `>> file`, la sortie standard est sauvegardée à la fin du fichier `file` (si `file` n'existait pas, il est créé). Des informations plus complètes sur les mécanismes de redirection de bash(1) peuvent être obtenues dans le chapitre 20 de [ABS].

```
$ echo "Un petit fichier de textes" > file.txt
$ echo "aaaaa bbbbb" >> file.txt
$ echo "bbbb ccc" >> file.txt
$ grep -v bbbb < file.txt > file.out
$ cat file.out
Un petit fichier de textes
```

Les shells Unix supportent un second mécanisme qui est encore plus intéressant pour combiner plusieurs programmes. Il s'agit de la redirection de la sortie standard d'un programme vers l'entrée standard d'un autre sans passer par un fichier intermédiaire. Cela se réalise avec le symbole | (*pipe* en anglais). L'exemple suivant illustre quelques combinaisons d'utilitaires de manipulation de texte.

```
$ echo "Un petit texte" | wc -c
      15
$ echo "bbbb ccc" >> file.txt
$ echo "aaaaa bbbbb" >> file.txt
$ echo "bbbb ccc" >> file.txt
$ cat file.txt
bbbb ccc
aaaaa bbbbb
bbbb ccc
$ cat file.txt | sort | uniq
aaaaa bbbbb
bbbb ccc
```

Le premier exemple est d'utiliser echo(1) pour générer du texte et le passer directement à wc(1) qui compte le nombre de caractères. Le deuxième exemple utilise cat(1) pour afficher sur la sortie standard le contenu d'un fichier. Cette sortie est reliée à sort(1) qui trie le texte reçu sur son entrée standard en ordre alphabétique croissant. Cette sortie en ordre alphabétique est reliée à uniq(1) qui la filtre pour en retirer les lignes dupliquées.

Tout shell Unix peut également s'utiliser comme un interpréteur de commande qui permet d'interpréter des scripts. Un système Unix peut exécuter deux types de programmes :

- des programmes exécutables en langage machine. C'est le cas de la plupart des utilitaires dont nous avons parlé jusqu'ici.
- des programmes écrits dans un langage interprété. C'est le cas des programmes écrits pour le shell, mais également pour d'autres langages interprétés comme python ou perl.

Lors de l'exécution d'un programme, le système d'exploitation reconnaît [4] si il s'agit d'un programme directement exécutable ou d'un programme interprété en analysant les premiers octets du fichier. Par convention, sous Unix, les deux premiers caractères d'un programme écrit dans un langage qui doit être interprété sont `#!`. Ils sont suivis par le nom complet de l'interpréteur qui doit être utilisé pour interpréter le programme.

Le programme bash(1) le plus simple est le suivant :

```
#!/bin/bash
echo "Hello, world"
```

L'exécution de ce script shell retourne la sortie suivante :

4. Sous Unix et contrairement à d'autres systèmes d'exploitation, le suffixe d'un nom de fichier ne joue pas de rôle particulier pour indiquer si un fichier contient un programme exécutable ou non. Comme nous le verrons ultérieurement, le système de fichiers Unix contient des bits de permission qui indiquent notamment si un fichier est exécutable ou non.

```
Hello, world
```

Par convention en bash(1), le caractère # marque le début d'un commentaire en début ou en cours de ligne. Comme tout langage, bash(1) permet à l'utilisateur de définir des variables. Celles-ci peuvent contenir des chaînes de caractères ou des nombres. Le script ci-dessous utilise deux variables, PROG et COURS et les utilise pour afficher un texte avec la commande echo.

```
#!/bin/bash
PROG="SINF"
COURS=1252
echo $PROG$COURS
```

Un script bash(1) peut également prendre des arguments passés en ligne de commande. Par convention, ceux-ci ont comme noms $1, $2, $3, ... L'exemple ci-dessous illustre l'utilisation de ces arguments.

```
#!/bin/bash
# $# nombre d'arguments
# $1 $2 $3 ... arguments
echo "Vous avez passe" $# "arguments"
echo "Le premier argument est :" $1
echo "Liste des arguments :" $@
```

L'exécution de ce script produit la sortie suivante :

```
Vous avez passe 2 arguments
Le premier argument est : SINF
Liste des arguments : SINF 1252
```

Concernant le traitement des arguments par un script bash, il est utile de noter que lorsque l'on appelle un script en redirigeant son entrée ou sa sortie standard, le script n'est pas informé de cette redirection. Ainsi, si l'on exécute le script précédent en faisant `args.sh arg1 > args.out`, le fichier `args.out` contient les lignes suivantes :

```
Vous avez passe 2 arguments
Le premier argument est : SINF
Liste des arguments : SINF 1252
```

bash(1) supporte la construction `if [ condition ]; then ... fi` qui permet notamment de comparer les valeurs de variables. bash(1) définit de nombreuses conditions différentes, dont notamment :

- `$i -eq $j` est vraie lorsque les deux variables `$i` et `$j` contiennent le même nombre.
- `$i -lt $j` est vraie lorsque la valeur de la variable `$i` est numériquement strictement inférieure à celle de la variable `$j`
- `$i -ge $j` est vraie lorsque la valeur de la variable `$i` est numériquement supérieure ou égale à celle de la variable `$j`
- `$s = $t` est vraie lorsque la chaîne de caractères contenue dans la variable `$s` est égale à celle qui est contenue dans la variable `$t`
- `-z $s` est vraie lorsque la chaîne de caractères contenue dans la variable `$s` est vide

D'autres types de test sont définis dans la page de manuel : bash(1). Le script ci-dessous fournit un premier exemple d'utilisation de tests avec bash(1).

```
#!/bin/bash
# Vérifie si les deux nombres passés en arguments sont égaux
if [ $# -ne 2 ]; then
    echo "Erreur, deux arguments sont nécessaires" > /dev/stderr
    exit 2
fi
if [ $1 -eq $2 ]; then
        echo "Nombres égaux"
else
        echo "Nombres différents"
fi
exit 0
```

Tout d'abord, ce script vérifie qu'il a bien été appelé avec deux arguments. Vérifier qu'un programme reçoit bien les arguments qu'il attend est une règle de bonne pratique qu'il est bon de respecter dès le début. Si le script n'est pas appelé avec le bon nombre d'arguments, un message d'erreur est affiché sur la sortie d'erreur standard et le script se termine avec un code de retour. Ces codes de retour sont importants car ils permettent à un autre programme, par exemple un autre script bash(1) de vérifier le bon déroulement d'un programme appelé. Le script `src/eq.sh` utilise des appels explicites à exit(1posix) même si par défaut, un script bash(1) qui n'en contient pas retourne un code de retour nul à la fin de son exécution.

Un autre exemple d'utilisation des codes de retour est le script `src/wordin.sh` repris ci-dessous qui utilise grep(1) pour déterminer si un mot passé en argument est présent dans un fichier texte. Pour cela, il exploite la variable spéciale `$?` dans laquelle bash(1) sauve le code de retour du dernier programme exécuté par le script.

```
#!/bin/bash
# wordin.sh
# Vérifie si le mot passé en premier argument est présent
# dans le fichier passé comme second argument
if [ $# -ne 2 ]; then
    echo "Erreur, deux arguments sont nécessaires" > /dev/stderr
    exit 2
fi
grep $1 $2 >/dev/null
# $? contient la valeur de retour de grep
if [ $? -eq 0 ]; then
        echo "Présent"
        exit 0
else
        echo "Absent"
        exit 1
fi
```

Ce programme utilise le fichier spécial `/dev/null`. Celui-ci est en pratique l'équivalent d'un trou noir. Il accepte toutes les données en écriture mais celles-ci ne peuvent jamais être relues. `/dev/null` est très utile lorsque l'on veut ignorer la sortie d'un programme et éviter qu'elle ne s'affiche sur le terminal. bash(1) supporte également `/dev/stdin` pour représenter l'entrée standard, `/dev/stdout` pour la sortie standard et `/dev/stderr` pour l'erreur standard.

Une description complète de bash(1) sort du cadre de ce cours. De nombreuses références à ce sujet sont disponibles [Cooper2011].

CHAPITRE 2

Langage C

2.1 Le langage C

Différents langages permettent au programmeur de construire des programmes qui seront exécutés par le processeur. En réalité, le processeur ne comprend qu'un langage : le langage machine. Ce langage est un langage binaire dans lequel toutes les commandes et toutes les données sont représentés sous la forme de séquences de bits.

Le langage machine est peu adapté aux humains et il est extrêmement rare qu'un informaticien doive manipuler des programmes directement en langage machine. Par contre, pour certaines tâches bien spécifiques, comme par exemple le développement de routines spéciales qui doivent être les plus rapides possibles ou qui doivent interagir directement avec le matériel, il est important de pouvoir efficacemment générer du langage machine. Cela peut se faire en utilisant un langage d'assemblage. Chaque famille de processeurs a un langage d'assemblage qui lui est propre. Le langage d'assemblage permet d'exprimer de façon symbolique les différentes instructions qu'un processeur doit exécuter. Nous aurons l'occasion de traiter à plusieurs reprises des exemples en langage d'assemblage dans le cadre de ce cours. Cela nous permettra de mieux comprendre la façon dont le processeur fonctionne et exécute les programmes. Le langage d'assemblage est converti en langage machine grâce à un *assembleur*.

Le langage d'assemblage est le plus proche du processeur. Il permet d'écrire des programmes compacts et efficaces. C'est aussi souvent la seule façon d'utiliser des instructions spéciales du processeur qui permettent d'interagir directement avec le matériel pour par exemple commander les dispositifs d'entrée/sortie. C'est essentiellement dans les systèmes embarqués qui disposent de peu de mémoire et pour quelques fonctions spécifiques des systèmes d'exploitation que le langage d'assemblage est utilisé de nos jours. La plupart des programmes applicatifs et la grande majorité des systèmes d'exploitation sont écrits dans des langages de plus haut niveau.

Le langage *C* [KernighanRitchie1998], développé dans les années 70 pour écrire les premières versions du système d'exploitation *Unix*, est aujourd'hui l'un des langages de programmation les plus utilisés pour développer des programmes qui doivent être rapides ou doivent interagir avec le matériel. La plupart des systèmes d'exploitation sont écrits en langage C.

Le langage C a été conçu à l'origine comme un langage proche du processeur qui peut être facilement compilé, c'est-à-dire traduit en langage machine, tout en conservant de bonnes performances.

La plupart des livres qui abordent la programmation en langage C commencent par présenter un programme très simple qui affiche à l'écran le message *Hello, world !*.

```c
/************************************
 * Hello.c
 *
 * Programme affichant sur la sortie
 * standard le message "Hello, world!"
 *
 ************************************/

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

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

```
{
   // affiche sur la sortie standard
   printf("Hello, world!\n");

   return EXIT_SUCCESS;
}
```

Pour être exécuté, ce programme doit être compilé. Il existe de nombreux compilateurs permettant de transformer le langage C en langage machine. Dans le cadre de ce cours, nous utiliserons gcc(1). Dans certains cas, nous pourrons être amenés à utiliser d'autres compilateurs comme *llvm*.

La compilation du programme `src/hello.c` peut s'effectuer comme suit sur une machine de type Unix :

```
$ gcc -Wall -o hello hello.c
$ ls -l
total 80
-rwxr-xr-x  1 obo  obo  8704 15 jan 22:32 hello
-rw-r--r--  1 obo  obo   288 15 jan 22:32 hello.c
```

gcc(1) supporte de très nombreuses options et nous aurons l'occasion de discuter de plusieurs d'entre elles dans le cadre de ce cours. Pour cette première utilisation, nous avons choisi l'option *-Wall* qui force gcc(1) à afficher tous les messages de type *warning* (dans cet exemple il n'y en a pas) et l'option *-o* suivie du nom de fichier *hello* qui indique le nom du fichier dans lequel le programme exécutable doit être sauvegardé par le compilateur [1].

Lorsqu'il est exécuté, le programme *hello* affiche simplement le message suivant sur la sortie standard :

```
$ ./hello
hello, world
$
```

Même si ce programme est très simple, il illustre quelques concepts de base en langage C. Tout d'abord comme en Java, les compilateurs récents supportent deux façons d'indiquer des commentaires en C :

- un commentaire sur une ligne est précédé des caractères *//*
- un commentaire qui comprend plusieurs lignes débute par */** et se termine par ***/*

Ensuite, un programme écrit en langage C comprend principalement des expressions en langage C mais également des expressions qui doivent être traduites par le *préprocesseur*. Lors de la compilation d'un fichier en langage C, le compilateur commence toujours par exécuter le préprocesseur. Celui-ci implémente différentes formes de macros qui permettent notamment d'inclure des fichiers (directives `#include`), de compiler de façon conditionnelle certaines lignes ou de définir des constantes. Nous verrons différentes utilisations du préprocesseur C dans le cadre de ce cours. A ce stade, les trois principales fonctions du préprocesseur sont :

- définir des substitutions via la macro `#define`. Cette macro est très fréquemment utilisée pour définir des constantes ou des substitutions qui sont valables dans l'ensemble du programme.

  ```
  #define ZERO  0
  #define STRING "SINF1252"
  ```

- importer (directive `#include`) un fichier. Ce fichier contient généralement des prototypes de fonctions et des constantes. En langage C, ces fichiers qui sont inclus dans un programme sont appelés des *header files* et ont par convention un nom se terminant par `.h`. Le programme `src/hello.c` ci-dessus importe deux fichiers *headers* standards :
  - `<stdio.h>` : contient la définition des principales fonctions de la librairie standard permettant l'interaction avec l'entrée et la sortie standard, et notamment printf(3)
  - `<stdlib.h>` : contient la définition de différentes fonctions et constantes de la librairie standard et notamment `EXIT_SUCCESS` et `EXIT_FAILURE`. Ces constantes sont définies en utilisant la macro `#define` du préprocesseur

    ```
    #define EXIT_FAILURE    1
    #define EXIT_SUCCESS    0
    ```

1. Si cette option n'était pas spécifiée, le compilateur aurait placé le programme compilé dans le fichier baptisé *a.out*.

— inclure du code sur base de la valeur d'une constante définie par un `#define`. Ce contrôle de l'inclusion de code sur base de la valeur de constantes est fréquemment utilisé pour ajouter des lignes qui ne doivent être exécutées que lorsque l'on veut débugger un programme. C'est aussi souvent utilisé pour faciliter la portabilité d'un programme entre différentes variantes de Unix, mais cette utilisation sort du cadre de ce cours.

```
#define DEBUG
/* ... */
#ifdef DEBUG
printf("debug : ...");
#endif /* DEBUG */
```

Il est également possible de définir des macros qui prennent un ou plusieurs paramètres [CPP].

Les headers standards sont placés dans des répertoires bien connus du système. Sur la plupart des variantes de Unix ils se trouvent dans le répertoire `/usr/include/`. Nous aurons l'occasion d'utiliser régulièrement ces fichiers standards dans le cadre du cours.

Le langage Java a été largement inspiré du langage C et de nombreuses constructions syntaxiques sont similaires en Java et en C. Un grand nombre de mots clés en C ont le même rôle qu'en Java. Les principaux types de données primitifs supportés par le C sont :

— `int` et `long` : utilisés lors de la déclaration d'une variable de type entier
— `char` : utilisé lors de la déclaration d'une variable permettant de stocker un caractère
— `double` et `float` pour les variables permettant de stocker un nombre représenté en virgule flottante.

Notez que dans les premières versions du langage C, contrairement à Java, il n'y avait pas de type spécifique permettant de représenter un booléen. Dans de nombreux programmes écrits en C, les booléens sont représentés par des entiers et les valeurs booléenne sont définies [2] comme suit.

```
#define false   0
#define true    1
```

Les compilateurs récents qui supportent le type booléen permettent de déclarer des variables de type `bool` et contiennent les définitions suivantes [2] dans le header standard stdbool.h de [C99].

```
#define false   (bool)0
#define true    (bool)1
```

Au-delà des types de données primitifs, Java et C diffèrent et nous aurons l'occasion d'y revenir dans un prochain chapitre. Le langage C n'est pas un langage orienté objet et il n'est donc pas possible de définir d'objet avec des méthodes spécifiques en C. C permet la définition de structures, d'unions et d'énumérations sur lesquelles nous reviendrons.

En Java, les chaînes de caractères sont représentées grâce à l'objet `String`. En C, une chaîne de caractères est représentée sous la forme d'un tableau de caractères dont le dernier élément contient la valeur `\0`. Alors que Java stocke les chaînes de caractères dans un objet avec une indication de leur longueur, en C il n'y a pas de longueur explicite pour les chaînes de caractères mais le caractère `\0` sert de marqueur de fin de chaîne de caractères. Lorsque le language C a été développé, ce choix semblait pertinent, notamment pour des raisons de performance. Avec le recul, ce choix pose question [Kamp2011] et nous y reviendrons lorsque nous aborderons certains problèmes de sécurité.

```
char string[10];
string[0] = 'j';
string[1] = 'a';
string[2] = 'v';
string[3] = 'a';
string[4] = '\0';
printf("String : %s\n", string);
```

2. Formellement, le standard [C99] ne définit pas de type `bool` mais un type `_Bool` qui est en pratique renommé en type `bool` dans la plupart des compilateurs. La définition précise et complète se trouve dans stdbool.h

L'exemple ci-dessus illustre l'utilisation d'un tableau de caractères pour stocker une chaîne de caractères. Lors de son exécution, ce fragment de code affiche `String : java` sur la sortie standard. Le caractère spécial `\n` indique un passage à la ligne. printf(3) supporte d'autres caractères spéciaux qui sont décrits dans sa page de manuel.

Au niveau des constructions syntaxiques, on retrouve les mêmes boucles et tests en C et en Java :

- test `if (condition) { ... } else { ... }`
- boucle `while (condition) { ... }`
- boucle `do { ... } while (condition);`
- boucle `for (init; condition; incr) { ... }`

En Java, les conditions sont des expressions qui doivent retourner un résultat de type `boolean`. Le langage C est beaucoup plus permissif puisqu'une condition est une expression qui retourne un nombre entier.

La plupart des expressions et conditions en C s'écrivent de la même façon qu'en Java.

Après ce rapide survol du langage C, revenons à notre programme `src/hello.c`. Tout programme C doit contenir une fonction nommée `main` dont la signature [3] est :

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

Lorsque le système d'exploitation exécute un programme C compilé, il démarre son exécution par la fonction `main` et passe à cette fonction les arguments fournis en ligne de commande [4]. Comme l'utilisateur peut passer un nombre quelconque d'arguments, il faut que le programme puisse déterminer combien d'arguments sont utilisés. En Java, la méthode `main` a comme signature `public static void main(String args[])` et l'attribut `args.length` permet de connaître le nombres de paramètres passés en arguments d'un programme. En C, le nombre de paramètres est passé dans la variable entière `argc` et le tableau de chaînes de caractères `char *argv[]` contient tous les arguments. Le programme `src/cmdline.c` illustre comment un programme peut accéder à ses arguments.

```
/************************************
 * cmdline.c
 *
 * Programme affichant ses arguments
 * sur la sortie standard
 *
 ************************************/

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

int main(int argc, char *argv[])
{
  int i;
  printf("Ce programme a %d argument(s)\n", argc);
  for (i = 0; i < argc; i++)
    printf("argument[%d] : %s\n", i, argv[i]);
  return EXIT_SUCCESS;
}
```

Par convention, en C le premier argument (se trouvant à l'indice `0` du tableau `argv`) est le nom du programme qui a été exécuté par l'utilisateur. Une exécution de ce programme est illustrée ci-dessous.

```
Ce programme a 5 argument(s)
argument[0] : ./cmdline
argument[1] : 1
argument[2] : -list
```

3. Il est également possible d'utiliser dans un programme C une fonction `main` qui ne prend pas d'argument. Sa signature sera alors `int main (void)`.

4. En pratique, le système d'exploitation passe également les variables d'environnement à la fonction `main`. Nous verrons plus tard comment ces variables d'environnement sont passées du système au programme et comment celui-ci peut y accéder. Sachez cependant que sous certaines variantes de Unix, et notamment Darwin/MacOS ainsi que sous certaines versions de Windows, le prototype de la fonction `main` inclut explicitement ces variables d'environnement (`int main(int argc, char *argv[], char *envp[])`

```
argument[3] : abcdef
argument[4] : sinf1252
```

Outre le traitement des arguments, une autre différence importante entre Java et C est la valeur de retour de la fonction `main`. En C, la fonction `main` retourne un entier. Cette valeur de retour est passée par le système d'exploitation au programme (typiquemment un *shell* ou interpréteur de commandes) qui a demandé l'exécution du programme. Grâce à cette valeur de retour il est possible à un programme d'indiquer s'il s'est exécuté correctement ou non. Par convention, un programme qui s'exécute sous Unix doit retourner `EXIT_SUCCESS` lorsqu'il se termine correctement et `EXIT_FAILURE` en cas d'échec. La plupart des programmes fournis avec un Unix standard respectent cette convention. Dans certains cas, d'autres valeurs de retour non nulles sont utilisées pour fournir plus d'informations sur la raison de l'échec. En pratique, l'échec d'un programme peut être dû aux arguments incorrects fournis par l'utilisateur ou à des fichiers qui sont inaccessibles.

A titre d'illustration, le programme `src/failure.c` est le programme le plus simple qui échoue lors de son exécution.

```c
/************************************
 * failure.c
 *
 * Programme minimal qui échoue toujours
 *
 ************************************/

#include <stdlib.h>

int main(int argc, char *argv[])
{
  return EXIT_FAILURE;
}
```

Enfin, le dernier point à mentionner concernant notre programme `src/hello.c` est la fonction `printf`. Cette fonction de la librairie standard se retrouve dans la plupart des programmes écrits en C. Elle permet l'affichage de différentes formes de textes sur la sortie standard. Comme toutes les fonctions de la librairie standard, elle est documentée dans sa page de manuel printf(3). printf(3) prend un nombre variable d'arguments. Le premier argument est une chaîne de caractères qui spécifie le format de la chaîne de caractères à afficher. Une présentation détaillée de printf(3) prendrait de nombreuses pages. A titre d'exemple, voici un petit programme utilisant printf(3)

```c
  char weekday[] = "Monday";
  char month[] = "April";
  int day = 1;
  int hour = 12;
  int min = 42;
  char str[] = "SINF1252";
  int i;

  // affichage de la date et l'heure
  printf("%s, %s %d, %d:%d\n", weekday, month, day, hour, min);

  // affichage de la valeur de PI
  printf("PI = %f\n", 4 * atan(1.0));

  // affichage d'un caractère par ligne
  for(i = 0; str[i] != '\0'; i++)
    printf("%c\n", str[i]);
```

Lors de son exécution, ce programme affiche :

```
Monday, April 1, 12:42
PI = 3.141593
S
I
N
```

```
F
1
2
5
2
```

Le langage C permet bien entendu la définition de fonctions. Outre la fonction `main` qui doit être présente dans tout programme, le langage C permet la définition de fonctions qui retournent ou non une valeur. En C, comme en Java, une fonction de type `void` ne retourne aucun résultat tandis qu'une fonction de type `int` retournera un entier. Le programme ci-dessous présente deux fonctions simples. La première, `usage` ne retourne aucun résultat. Elle affiche un message d'erreur sur la sortie d'erreur standard et termine le programme via exit(2) avec un code de retour indiquant un échec. La seconde, `digit` prend comme argument un caractère et retourne `1` si c'est un chiffre et `0` sinon. Le code de cette fonction peut paraître bizarre à un programmeur habitué à Java. En C, les *char* sont représentés par l'entier qui correspond au caractère dans la table des caractères utilisées (voir **RFC 20** pour une table ASCII simple). Toutes les tables de caractères placent les chiffres `0` à `9` à des positions consécutives. En outre, en C une expression a priori booléenne comme `a < b` est définie comme ayant la valeur `1` si elle est vraie et `0` sinon. Il en va de même pour les expressions qui sont combinées en utilisant `&&` ou `||`. Enfin, les fonctions getchar(3) et putchar(3) sont des fonctions de la librairie standard qui permettent respectivement de lire (écrire) un caractère sur l'entrée (la sortie) standard.

```c
/************************************
 * filterdigit.c
 *
 * Programme qui extrait de l'entrée
 * standard les caractères représentant
 * des chiffres
 ************************************/

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

// retourne vrai si c est un chiffre, faux sinon
int digit(char c)
{
  return ((c >= '0') && (c <= '9'));
}

// affiche un message d'erreur
void usage()
{
  fprintf(stderr, "Ce programme ne prend pas d'argument\n");
  exit(EXIT_FAILURE);
}

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

  if (argc > 1)
    usage();

  while ((c = getchar()) != EOF) {
    if (digit(c))
            putchar(c);
  }

  return EXIT_SUCCESS;
}
```

Pages de manuel

Les systèmes d'exploitation de la famille Unix contiennent un grand nombre de librairies, d'appels systèmes et d'utilitaires. Toutes ces fonctions et tous ces programmes sont documentés dans des pages de manuel qui sont accessibles via la commande `man`. Les pages de manuel sont organisées en 8 sections.

- Section 1 : Utilitaires disponibles pour tous les utilisateurs
- Section 2 : Appels systèmes en C
- Section 3 : Fonctions de la librairie
- Section 4 : Fichiers spéciaux
- Section 5 : Formats de fichiers et conventions pour certains types de fichiers
- Section 6 : Jeux
- Section 7 : Utilitaires de manipulation de fichiers textes
- Section 8 : Commandes et procédure de gestion du système

Dans le cadre de ce cours, nous aborderons principalement les fonctionnalités décrites dans les trois premières sections des pages de manuel. L'accès à une page de manuel se fait via la commande `man` avec comme argument le nom de la commande concernée. Vous pouvez par exemple obtenir la page de manuel de `gcc` en tapant `man gcc`. `man` supporte plusieurs paramètres qui sont décrits dans sa page de manuel accessible via `man man`. Dans certains cas, il est nécessaire de spécifier la section du manuel demandée. C'est le cas par exemple pour `printf` qui existe comme utilitaire (section 1) et comme fonction de la librairie (section 3 - accessible via `man 3 printf`).

Outre ces pages de manuel locales, il existe également de nombreux sites web où l'on peut accéder aux pages de manuels de différentes versions de Unix dont notamment :

- les pages de manuel de Debian GNU/Linux
- les pages de manuel de FreeBSD
- les pages de manuel de MacOS

Dans la version online de ces notes, toutes les références vers un programme Unix, un appel système ou une fonction de la librairie pointent vers la page de manuel Linux correspondante.

2.2 Types de données

Durant la première semaine, nous avons abordé quelques types de données de base dont les `int` et les `char`. Pour utiliser ces types de données à bon escient, il est important de comprendre en détails la façon dont ils sont supportés par le compilateur et leurs limitations. Celles-ci dépendent souvent de leur représentation en mémoire et durant cette semaine nous allons commencer à analyser de façon plus détaillée comment la mémoire d'un ordinateur est structurée.

2.2.1 Nombres entiers

Toutes les données stockées sur un ordinateur sont représentées sous la forme de séquences de bits. Ces séquences de bits peuvent d'abord permettre de représenter des nombres entiers. Un système informatique peut travailler avec deux types de nombres entiers :

- les nombres entiers signés (`int` notamment en C)
- les nombres entiers non-signés (`unsigned int` notamment en C)

Une séquence de n bits $b_0...b_i...b_n$ peut représenter le nombre entier $\sum_{i=0}^{n-1} b_i \times 2^i$. Par convention, le bit b_n, associé au facteur du plus grand indice 2^n, est appelé le *bit de poids fort* tandis que le bit b_0, associé à 2^0, est appelé le *bit de poids faible*. Les suites de bits sont communément écrites dans l'ordre descendant des indices $b_n...b_i...b_0$. A titre d'exemple, la suite de bits `0101` correspond à l'entier non signé représentant la valeur cinq. Le bit de poids fort (resp. faible) de cette séquence de quatre bits (ou *nibble*) est `0` (resp. `1`). La table ci-dessous reprend les différentes valeurs décimales correspondant à toutes les séquences de quatre bits consécutifs.

binaire	octal	hexadécimal	décimal
0000	00	0	0
0001	01	1	1
0010	02	2	2
0011	03	3	3
0100	04	4	4
0101	05	5	5
0110	06	6	6
0111	07	7	7
1000	10	8	8
1001	11	9	9
1010	12	A	10
1011	13	B	11
1100	14	C	12
1101	15	D	13
1110	16	E	14
1111	17	F	15

Ecrire une séquence de bits sous la forme d'une suite de `0` et de `1` peut s'avérer fastidieux. La représentation décimale traditionnelle n'est pas pratique non plus car il faut un ou deux chiffres pour représenter une séquence de quatre bits (ou *nibble*) en fonction de la valeur de ces bits. En pratique, de nombreux systèmes informatiques utilisent une représentation hexadécimale pour afficher des séquences de bits. Cette notation hexadécimale est définie sur base de la table ci-dessus en utilisant des lettres pour représenter chaque séquence de quatre bits dont la valeur numérique est supérieure à `9`. Tout comme avec la représentation décimale habituelle, il est possible d'utiliser la représentation hexadécimale pour de longues séquences de bits. La notation octale est parfois utilisée et est supportée par les compilateurs C. Elle utilise un chiffre pour représenter trois bits consécutifs. A titre d'exemple, voici quelques conversions de nombres en notation décimale vers les notations hexadécimales et binaires.

— L'entier décimal `123` s'écrit `0x7b` en notation hexadécimale et `0b00000000000000000000000001111011` en notation binaire
— L'entier décimal `7654321` s'écrit `0x74cbb1` en notation hexadécimale et `0b00000000011101001100101110110001` en notation binaire

Dans ces exemples, nous avons pris la convention de représentation des nombres en langage C. En C, un nombre décimal s'écrit avec la représentation standard. Un nombre entier en notation hexadécimale est par convention préfixé par `0x` tandis qu'un nombre entier en notation binaire est préfixé par `0b`. Ainsi, les déclarations ci-dessous correspondent toutes à la même valeur.

```
int i;
i = 123;   // décimal
i = 0x7b;  // hexadécimal
i = 0173;  // octal !!
//  i = 0b1111011; // binaire, seulement certains compilateurs
```

Certains compilateurs permettent d'entrer des constantes en binaire directement en préfixant le nombre avec `0b` comme dans `i=0b1111011;`. Cependant, cette notation n'est pas portable sur tous les compilateurs. Elle doit donc être utilisée avec précaution, contrairement à la notation hexadécimale qui fait partie du langage.

Note : Notation octale

La notation octale peut poser des surprises désagréables au programmeur C débutant. En effet, pour des raisons historiques, les compilateurs C considèrent qu'un entier dont le premier chiffre est `0` est écrit en représentation octale et non en représentation décimale.

Ainsi, le fragment de code ci-dessous affichera à l'écran le message `65 et 53 sont différents` car le compilateur C interprète la ligne `j=065;` comme contenant un entier en notation octale et non décimale.

```
int i, j;
i = 65;   // décimal
j = 065;  // octal !!!
if (i == j)
  printf("%d et %d sont égaux\n", i, j);
```

```
else
  printf("%d et %d sont différents\n", i, j);
```

Le langage C supporte différents types de données qui permettent de représenter des nombres entiers non signés. Les principaux sont repris dans le tableau ci-dessous.

Type	Explication
`unsigned short`	Nombre entier non signé représenté sur au moins 16 bits
`unsigned int`	Nombre entier non signé représenté sur au moins 16 bits
`unsigned long`	Nombre entier non signé représenté sur au moins 32 bits
`unsigned long long`	Nombre entier non signé représenté sur au moins 64 bits

Le nombre de bits utilisés pour stocker chaque type d'entier non signé peut varier d'une implémentation à l'autre. Le langage C permet de facilement déterminer le nombre de bits utilisés pour stocker un type de données particulier en utilisant l'expression `sizeof`. Appliquée à un type de données, celle-ci retourne le nombre d'octets que ce type occupe. Ainsi, sur de nombreuses plateformes, `sizeof(int)` retournera la valeur `4`.

Les systèmes informatiques doivent également manipuler des nombres entiers négatifs. Cela se fait en utilisant des nombres dits signés. Au niveau binaire, il y a plusieurs approches possibles pour représenter des nombres signés. La première est de réserver le bit de poids fort dans la représentation du nombre pour stocker le signe et stocker la valeur absolue du nombre dans les bits de poids faible. Mathématiquement, un nombre de *n* bits utilisant cette notation pourrait se convertir via la formule $(-1)^{b_{n-1}} \times \sum_{i=0}^{n-2} b_i \times 2^i$.

En pratique, cette notation est rarement utilisée pour les nombres entiers car elle rend l'implémentation des circuits électroniques de calcul plus compliquée. Un autre inconvénient de cette notation est qu'elle utilise deux séquences de bits différentes pour représenter la valeur zéro (`00...0` et `10...0`). La représentation la plus courante pour les nombres entiers signés est la notation en *complément à 2*. Avec cette notation, une séquence de *n* bits correspond au nombre entier $-(b_{n-1}) \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times 2^i$. Avec cette notation, le nombre négatif de 4 bits le plus petit correspond à la valeur `-8`. En notation en complément à deux, il n'y a qu'une seule représentation pour le nombre zéro, la séquence dont tous les bits valent `0`. Par contre, il existe toujours un nombre entier négatif qui n'a pas d'équivalent positif.

binaire	décimal signé
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

En C, les types de données utilisés pour représenter des entiers sont signés par défaut. Ils ont la même taille que leurs équivalents non-signés et sont repris dans la table ci-dessous.

Type	Explication
`short`	Nombre entier signé représenté sur au moins 16 bits
`int`	Nombre entier signé représenté sur au moins 16 bits
`long`	Nombre entier signé représenté sur au moins 32 bits
`long long`	Nombre entier signé représenté sur au moins 64 bits

Dans de nombreux systèmes Unix, on retrouve dans le fichier stdint.h les définitions des types d'entiers supportés par le système avec le nombre de bits et les valeurs minimales et maximales pour chaque type. La table ci-dessous

reprend à titre d'exemple l'information relative aux types `short` (16 bits) et `unsigned int` (32 bits).

Type	Bits	Minimum	Maximum
`short`	16	-32768	32767
`unsigned int`	32	0	4294967295

Le fichier stdint.h contient de nombreuses constantes qui doivent être utilisées lorsque l'on a besoin des valeurs minimales et maximales pour un type donné. Voici à titre d'exemple quelques unes de ces valeurs :

```
#define INT8_MAX         127
#define INT16_MAX        32767
#define INT32_MAX        2147483647
#define INT64_MAX        9223372036854775807LL

#define INT8_MIN          -128
#define INT16_MIN         -32768
#define INT32_MIN        (-INT32_MAX-1)
#define INT64_MIN        (-INT64_MAX-1)

#define UINT8_MAX         255
#define UINT16_MAX        65535
#define UINT32_MAX        4294967295U
#define UINT64_MAX        18446744073709551615ULL
```

L'utilisation d'un nombre fixe de bits pour représenter les entiers peut causer des erreurs dans certains calculs. Par exemple, voici un petit programme qui affiche les 10 premières puissances de cinq et dix.

```
short int i = 1;
unsigned short j = 1;
int n;

printf("\nPuissances de 5 en notation signée\n");
for (n = 1; n < 10; n++) {
  i = i * 5;
  printf("5^%d=%d\n", n, i);
}

printf("\nPuissances de 10 en notation non signée\n");
for (n = 1; n < 10; n++) {
  j = j * 10;
  printf("10^%d=%d\n", n, j);
}
```

Lorsqu'il est exécuté, ce programme affiche la sortie suivante.

```
Puissances de 5 en notation signée
5^1=5
5^2=25
5^3=125
5^4=625
5^5=3125
5^6=15625
5^7=12589
5^8=-2591
5^9=-12955

Puissances de 10 en notation non signée
10^1=10
10^2=100
10^3=1000
10^4=10000
10^5=34464
10^6=16960
10^7=38528
```

```
10^8=57600
10^9=51712
```

Il est important de noter que le langage C ne contient aucun mécanisme d'exception qui permettrait au programmeur de détecter ce problème à l'exécution. Lorsqu'un programmeur choisit une représentation pour stocker des nombres entiers, il est essentiel qu'il ait en tête l'utilisation qui sera faite de cet entier et les limitations qui découlent du nombre de bits utilisés pour représenter le nombre en mémoire. Si dans de nombreuses applications ces limitations ne sont pas pénalisantes, il existe des applications critiques où un calcul erroné peut avoir des conséquences énormes [Bashar1997].

2.2.2 Nombres réels

Outre les nombres entiers, les systèmes informatiques doivent aussi pouvoir manipuler des nombres réels. Ceux-ci sont également représentés sous la forme d'une séquence fixe de bits. Il existe deux formes de représentation pour les nombres réels :

— la représentation en *simple précision* dans laquelle le nombre réel est stocké sous la forme d'une séquence de 32 bits
— la représentation en *double précision* dans laquelle le nombre réel est stocké sous la forme d'une séquence de 64 bits

La plupart des systèmes informatiques qui permettent de manipuler des nombres réels utilisent le standard IEEE-754. Un nombre réel est représenté en virgule flottante et la séquence de bits correspondante est décomposée en trois parties [5] :

— le bit de poids fort indique le signe du nombre. Par convention, `0` est utilisé pour les nombres positifs et `1` pour les nombres négatifs.
— e bits sont réservés pour stocker l'exposant [6]
— les f bits de poids faible servent à stocker la partie fractionnaire du nombre réel

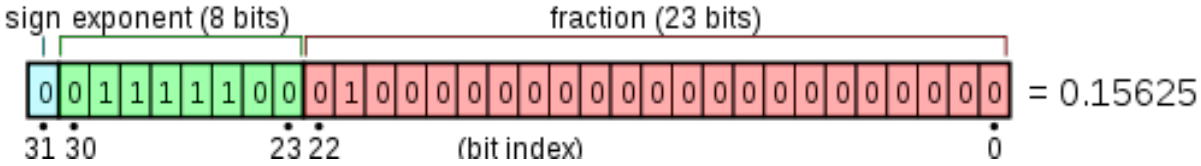


FIGURE 2.1 – Exemple de nombre en virgule flottante (simple précision) (source : wikipedia)

En simple (resp. double) précision, *8* (resp. *11*) bits sont utilisés pour stocker l'exposant et *23* (resp. *52*) bits pour la partie fractionnaire. L'encodage des nombres réels en simple et double précision a un impact sur la précision de divers algorithmes numériques. Une analyse détaillée de ces problèmes sort du cadre de ce cours, mais il est important de noter deux propriétés importantes de la notation en virgule flottante utilisée actuellement. Ces deux propriétés s'appliquent de la même façon à la simple qu'à la double précision.

— une représentation en virgule flottante sur n bits ne permet jamais de représenter plus de 2^n nombres réels différents
— les représentations en virgule flottante privilégient les nombres réels compris dans l'intervalle $[-1, 1]$. On retrouve autant de nombres réels représentables dans cet intervalle que de nombres dont la valeur absolue est supérieure à *1*.

En C, ces nombres en virgule flottante sont représentés en utilisant les types `float` (simple précision) et `double` (double précision). Les fichiers float.h et math.h définissent de nombreuses constantes relatives à ces types. Voici, à titre d'exemple, les valeurs minimales et maximales pour les `float` et les `double` ainsi que les constantes associées. Pour qu'un programme soit portable, il faut utiliser les constantes définies dans float.h et math.h et non leurs valeurs numériques.

```
#define FLT_MIN 1.17549435e-38F
#define FLT_MAX 3.40282347e+38F
```

5. Source : http://en.wikipedia.org/wiki/Single-precision_floating-point_format
6. En pratique, le format binaire contient $127 + exp$ en simple précision et non l'exposant exp. Ce choix facilite certaines comparaisons entre nombres représentés en virgule flottante. Une discussion détaillée de la représentation binaire des nombres en virgule flottante sort du cadre de ce cours dédié aux systèmes informatiques. Une bonne référence à ce sujet est [Goldberg1991].

```
#define DBL_MIN 2.2250738585072014e-308
#define DBL_MAX 1.7976931348623157e+308
```

2.2.3 Les tableaux

En langage C, les tableaux permettent d'agréger des données d'un même type. Il est possible de définir des vecteurs et des matrices en utilisant la syntaxe ci-dessous.

```
#define N 10
int vecteur[N];
float matriceC[N][N];
float matriceR[N][2*N];
```

Les premières versions du langage C ne permettaient que la définition de tableaux dont la taille est connue à la compilation. Cette restriction était nécessaire pour permettre au compilateur de réserver la zone mémoire pour stocker le tableau. Face à cette limitation, de nombreux programmeurs définissaient la taille du tableau via une directive `#define` du pré-processeur comme dans l'exemple ci-dessus. Cette directive permet d'associer une chaîne de caractères quelconque à un symbôle. Dans l'exemple ci-dessus, la chaîne `10` est associée au symbôle `N`. Lors de chaque compilation, le préprocesseur remplace toutes les occurences de `N` par `10`. Cela permet au compilateur de ne traiter que des tableaux de taille fixe.

Un tableau à une dimension peut s'utiliser avec une syntaxe similaire à celle utilisée par Java. Dans un tableau contenant `n` éléments, le premier se trouve à l'indice `0` et le dernier à l'indice `n-1`. L'exemple ci-dessous présente le calcul de la somme des éléments d'un vecteur.

```
int i;
int sum = 0;
for (i = 0; i < N; i++) {
  sum += v[i];
}
```

Le langage C permet aussi la manipulation de matrices carrées ou rectangulaires qui sont composées d'éléments d'un même type. L'exemple ci-dessous calcule l'élément minimum d'une matrice rectangulaire. Il utilise la constante `FLT_MAX` qui correspond au plus grand nombre réel représentable avec un `float` qui est définie dans float.h.

```
#define L 2
#define C 3
float matriceR[L][C] = { {1.0,2.0,3.0},
                         {4.0,5.0,6.0} };
int i, j;
float min = FLT_MAX;
for (i = 0; i < L; i++)
  for (j = 0; j < C; j++)
    if (matriceR[i][j] < min)
      min=matriceR[i][j];
```

Les compilateurs récents qui supportent [C99] permettent l'utilisation de tableaux dont la taille n'est connue qu'à l'exécution. Nous en reparlerons ultérieurement.

2.2.4 Caractères et chaînes de caractères

Historiquement, les caractères ont été représentés avec des séquences de bits de différentes longueurs. Les premiers ordinateurs utilisaient des blocs de cinq bits. Ceux-ci permettaient de représenter 32 valeurs différentes. Cinq bits ne permettent pas facilement de représenter à la fois les chiffres et les lettres et les premiers ordinateurs utilisaient différentes astuces pour supporter ces caractères sur 5 bits. Ensuite, des représentations sur six puis sept et huit bits ont été utilisées. Au début des années septante, le code de caractères ASCII sur `7` et `8` bits s'est imposé sur un grand nombre d'ordinateurs et a été utilisé comme standard pour de nombreuses applications et notamment sur Internet **RFC 20**. La table de caractères ASCII définit une correspondance entre des séquences de bits et des

caractères. **RFC 20** contient la table des caractères ASCII représentés sur 7 bits. A titre d'exemple, le chiffre *0* correspond à l'octet *0b00110000* et le chiffre *9* à l'octet *0b00111001*. La lettre *a* correspond à l'octet *0b01100001* et la lettre *A* à l'octet *0b01000001*.

Les inventeurs du C se sont appuyés sur la table ASCII et ont choisi de représenter un caractère en utilisant un octet. Cela correspond au type `char` que nous avons déjà évoqué.

Concernant le type `char`, il est utile de noter qu'un `char` est considéré en C comme correspondant à un entier. Cela implique qu'il est possible de faire des manipulations numériques sur les caractères. A titre d'exemple, une fonction toupper(3) permettant de transformer un caractère représentant une minuscule dans le caractère représentant la majuscule correspondante peut s'écrire :

```
// conversion de minuscules en majuscules
int toUpper(char c) {
  if (c >= 'a' && c <= 'z')
    return c + ('A' - 'a');
  else
    return c;
}
```

En pratique, l'utilisation de la table ASCII pour représenter des caractères souffre d'une limitation majeure. Avec *7* ou *8* bits il n'est pas possible de représenter exactement tous les caractères écrits de toutes les langues. Une table des caractères sur *7* bits est suffisante pour les langues qui utilisent peu de caractères accentués comme l'anglais. Pour le français et de nombreuses langues d'Europe occidentale, la table sur *8* bits est suffisante et la norme ISO-8859 contient des tables de caractères *8* bits pour de nombreuses langues. La norme Unicode va plus loin en permettant de représenter les caractères écrits de toutes les langues connues sur Terre. Une description détaillée du support de ces types de caractères sort du cadre de ce cours sur les systèmes informatiques. Il est cependant important que vous soyez conscient de cette problématique pour pouvoir la prendre en compte lorsque vous développerez des applications qui doivent traiter du texte dans différentes langues. A titre d'exemple, la fonction toupper(3) qui est implémentée dans les versions actuelles de Linux est nettement plus complexe que celle que nous avons vue ci-dessus. Tout d'abord, la fonction toupper(3) prend comme argument un `int` et non un `char`. Cela lui permet d'accepter des caractères dans n'importe quel encodage. Ensuite, le traitement qu'elle effectue dépend du type d'encodage qui a été défini via setlocale(3) (voir locale(7)).

Dans la suite du cours, nous supposerons qu'un caractère est toujours représentable en utilisant le type `char` permettant de stocker un octet.

En C, les chaînes de caractères sont représentées sous la forme d'un tableau de caractères. Une chaîne de caractères peut s'initialiser de différentes façons reprises ci-dessous.

```
char name1[] = { 'U', 'n', 'i', 'x' };
char name2[] = { "Unix" };
char name3[] = "Unix";
```

Lorsque la taille de la chaîne de caractères n'est pas indiquée à l'initialisation (c'est-à-dire dans les deux dernières lignes ci-dessus), le compilateur C la calcule et alloue un tableau permettant de stocker la chaîne de caractères suivie du caractère `\0` qui par convention termine *toujours* les chaînes de caractères en C. En mémoire, la chaîne de caractères correspondant à `name3` occupe donc cinq octets. Les quatre premiers contiennent les caractères *U*, *n*, *i* et *x* et le cinquième le caractère `\0`. Il est important de bien se rappeler cette particularité du langage C car comme nous le verrons plus tard ce choix a de nombreuses conséquences.

Cette particularité permet d'implémenter facilement des fonctions de manipulation de chaînes de caractères. A titre d'exemple, la fonction ci-dessous calcule la longueur d'une chaîne de caractères.

```
int length(char str[])
{
  int i = 0;
  while (str[i] != 0) // '\0' et 0 sont égaux
    i++;
  return i;
}
```

Contrairement à des langages comme Java, C ne fait aucune vérification sur la façon dont un programme manipule un tableau. En C, il est tout à fait légal d'écrire le programme suivant :

```
char name[5] = "Unix";
printf("%c", name[6]);
printf("%c", name[12345]);
printf("%c", name[-1]);
```

En Java, tous les accès au tableau `name` en dehors de la zone mémoire réservée provoqueraient une `ArrayOutOfBoundException`. En C, il n'y a pas de mécanisme d'exception et le langage présuppose que lorsqu'un programmeur écrit `name[i]`, il a la garantie que la valeur `i` sera telle qu'il accédera bien à un élément valide du tableau `name`. Ce choix de conception du C permet d'obtenir du code plus efficace qu'avec Java puisque l'interpréteur Java doit vérifier tous les accès à chaque tableau lorsqu'ils sont exécutés. Malheureusement, ce choix de conception du C est aussi à l'origine d'un très grand nombre de problèmes qui affectent la sécurité de nombreux logiciels. Ces problèmes sont connus sous la dénomination *buffer overflow*. Nous aurons l'occasion d'y revenir plus tard.

2.2.5 Les pointeurs

Une différence majeure entre le C et la plupart des langages de programmation actuels est que le C est proche de la machine et permet au programmeur d'interagir directement avec la mémoire où les données qu'un programme manipule sont stockées. En Java, un programme peut créer autant d'objets qu'il souhaite (ou presque [7]). Ceux-ci sont stockés en mémoire et le *garbage collector* retire de la mémoire les objets qui ne sont plus utilisés. En C, un programme peut aussi réserver des zones pour stocker de l'information en mémoire. Cependant, comme nous le verrons plus tard, c'est le programmeur qui doit explicitement allouer et désallouer la mémoire.

Les *pointeurs* sont une des caractéristiques principales du langage C par rapport à de nombreux autres langages. Un *pointeur* est défini comme étant une variable contenant l'adresse d'une autre variable. Pour bien comprendre le fonctionnement des pointeurs, il est important d'avoir en tête la façon dont la mémoire est organisée sur un ordinateur. D'un point de vue abstrait, la mémoire d'un ordinateur peut être vue sous la forme d'une zone de stockage dans laquelle il est possible de lire ou d'écrire de l'information. Chaque zone permettant de stocker de l'information est identifiée par une *adresse*. La mémoire peut être vue comme l'implémentation de deux fonctions C :

- `data read(addr)` est une fonction qui sur base d'une adresse retourne la valeur stockée à cette adresse
- `void write(addr, data)` est une fonction qui écrit la donnée `data` à l'adresse `addr` en mémoire.

Ces adresses sont stockées sur un nombre fixe de bits qui dépend en général de l'architecture du microprocesseur. Les valeurs les plus courantes aujourd'hui sont *32* et *64*. Par convention, les adresses sont représentées sous la forme d'entiers non-signés. Sur la plupart des architectures de processeurs, une adresse correspond à une zone mémoire permettant de stocker un octet. Lorsque nous utiliserons une représentation graphique de la mémoire, nous placerons toujours les adresses numériquement basses en bas de la figure et elles croîtront vers le haut.

Considérons l'initialisation ci-dessous et supposons qu'elle est stockée dans une mémoire où les adresses sont encodées sur *3* bits. Une telle mémoire dispose de huit slots permettant chacun de stocker un octet.

```
char name[] = "Unix";
char c = 'Z';
```

Après exécution de cette initialisation et en supposant que rien d'autre n'est stocké dans cette mémoire, celle-ci contiendra les informations reprises dans la table ci-dessous.

7. En pratique, l'espace mémoire accessible à un programme Java est limité par différents facteurs. Voir notamment le paramètre `-Xm` de la machine virtuelle Java http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/java.html

Adresse	Contenu
111	0
110	0
101	Z
100	0
011	x
010	i
001	n
000	U

En langage C, l'expression `&var` permet de récupérer l'adresse à laquelle une variable a été stockée. Appliquée à l'exemple ci-dessus, l'expression `&(name[0])` retournerait la valeur `0b000` tandis que `&c` retournerait la valeur `0b101`.

L'expression `&` peut s'utiliser avec n'importe quel type de donnée. Les adresses de données en mémoire sont rarement affichées, mais quand c'est le cas, on utilise la notation hexadécimale comme dans l'exemple ci-dessous.

```
int i = 1252;
char str[] = "sinf1252";
char c = 'c';

printf("i vaut %d, occupe %ld bytes et est stocké à l'adresse : %p\n",
       i, sizeof(i), &i);
printf("c vaut %c, occupe %ld bytes et est stocké à l'adresse : %p\n",
       c, sizeof(c), &c);
printf("str contient \"%s\" et est stocké à partir de l'adresse : %p\n",
       str, &str);
```

L'exécution de ce fragment de programme produit la sortie suivante.

```
i vaut 1252, occupe 4 bytes et est stocké à l'adresse : 0x7fff89f99cbc
c vaut c, occupe 1 bytes et est stocké à l'adresse : 0x7fff89f99caf
str contient "sinf1252" et est stocké à partir de l'adresse : 0x7fff89f99cb0
```

L'intérêt des pointeurs en C réside dans la possibilité de les utiliser pour accéder et manipuler des données se trouvant en mémoire de façon efficace. En C, chaque pointeur a un type et le type du pointeur indique le type de la donnée qui est stockée dans une zone mémoire particulière. Le type est associé au pointeur lors de la déclaration de celui-ci.

```
int i = 1;       // entier
int *ptr_i;      // pointeur vers un entier
char c = 'Z';    // caractère
char *ptr_c;     // pointeur vers un char
```

Grâce aux pointeurs, il est possible non seulement d'accéder à l'adresse où une donnée est stockée, mais aussi d'accéder à la valeur qui est stockée dans la zone mémoire pointée par le pointeur en utilisant l'expression `*ptr`. Il est également possible d'effectuer des calculs sur les pointeurs comme représenté dans l'exemple ci-dessous.

```
int i = 1;            // entier
int *ptr_i;           // pointeur vers un entier
char str[] = "Unix";
char *s;              // pointeur vers un char

ptr_i = &i;
printf("valeur de i : %d, valeur pointée par ptr_i : %d\n", i, *ptr_i);
*ptr_i = *ptr_i + 1252;
printf("valeur de i : %d, valeur pointée par ptr_i : %d\n", i, *ptr_i);
s = str;
for (i = 0; i < strlen(str); i++){
  printf("valeur de str[%d] : %c, valeur pointée par *(s+%d) : %c\n",
         i, str[i], i, *(s+i));
}
```

L'exécution de ce fragment de programme produit la sortie suivante.

```
valeur de i : 1, valeur pointée par ptr_i : 1
valeur de i : 1253, valeur pointée par ptr_i : 1253
valeur de str[0] : U, valeur pointée par *(s+0) : U
valeur de str[1] : n, valeur pointée par *(s+1) : n
valeur de str[2] : i, valeur pointée par *(s+2) : i
valeur de str[3] : x, valeur pointée par *(s+3) : x
```

En pratique en C, les notations `char*` et `char[]` sont équivalentes et l'une peut s'utiliser à la place de l'autre. En utilisant les pointeurs, la fonction de calcul de la longueur d'une chaîne de caractères peut se réécrire comme suit.

```
int length(char *s)
{
  int i = 0;
  while (*(s+i) != '\0')
    i++;
  return i;
}
```

Les pointeurs sont fréquemment utilisés dans les programmes écrits en langage C et il est important de bien comprendre leur fonctionnement. Un point important à bien comprendre est ce que l'on appelle l'*arithmétique des pointeurs*, c'est-à-dire la façon dont les opérations sur les pointeurs sont exécutées en langage C. Pour cela, il est intéressant de considérer la manipulation d'un tableau d'entiers à travers des pointeurs.

```
#define SIZE 3
unsigned int tab[3];
tab[0] = 0x01020304;
tab[1] = 0x05060708;
tab[2] = 0x090A0B0C;
```

En mémoire, ce tableau est stocké en utilisant trois mots consécutifs de 32 bits comme le montre l'exécution du programme ci-dessous :

```
int i;
for (i = 0; i < SIZE; i++) {
  printf("%X est a l'adresse %p\n", tab[i], &(tab[i]));
}
```

```
1020304 est a l'adresse 0x7fff5fbff750
5060708 est a l'adresse 0x7fff5fbff754
90A0B0C est a l'adresse 0x7fff5fbff758
```

La même sortie est produite avec le fragment de programme suivant qui utilise un pointeur.

```
unsigned int* ptr = tab;
for (i = 0; i < SIZE; i++) {
  printf("%X est a l'adresse %p\n", *ptr, ptr);
  ptr++;
}
```

Ce fragment de programme est l'occasion de réfléchir sur la façon dont le C évalue les expressions qui contiennent des pointeurs. La première est l'assignation `ptr=tab`. Lorsque `tab` est déclaré par la ligne `unsigned int tab[3]`, le compilateur considère que `tab` est une constante qui contiendra toujours l'adresse du premier élément du tableau. Il faut noter que puisque `tab` est considéré comme une constante, il est interdit d'en modifier la valeur en utilisant une assignation comme `tab=tab+1`. Le pointeur `ptr`, par contre correspond à une zone mémoire qui contient une adresse. Il est tout à fait possible d'en modifier la valeur. Ainsi, l'assignation `ptr=tab` (ou `ptr=&(tab[0])`) place dans `ptr` l'adresse du premier élément du tableau. Les pointeurs peuvent aussi être modifiés en utilisant des expressions arithmétiques.

```
ptr = ptr + 1;  // ligne 1
ptr++;          // ligne 2
ptr = ptr - 2;  // ligne 3
```

Après l'exécution de la première ligne, `ptr` va contenir l'adresse de l'élément `1` du tableau `tab` (c'est-à-dire `&(tab[1])`). Ce résultat peut surprendre car si l'élément `tab[0]` se trouve à l'adresse `0x7fff5fbff750` c'est cette adresse qui est stocké dans la zone mémoire correspondant au pointeur `ptr`. On pourrait donc s'attendre à ce que l'expression `ptr+1` retourne plutôt la valeur `0x7fff5fbff751`. Il n'est en rien. En C, lorsque l'on utilise des calculs qui font intervenir des pointeurs, le compilateur prend en compte le type du pointeur qui est utilisé. Comme `ptr` est de type `unsigned int*`, il pointe toujours vers une zone mémoire permettant de stocker un entier non-signé sur 32 bits. L'expression `ptr+1` revient en fait à calculer la valeur `ptr+sizeof(unsigned int)` et donc `ptr+1` correspondra à l'adresse `0x7fff5fbff754`. Pour la même raison, l'exécution de la deuxième ligne placera l'adresse `0x7fff5fbff758` dans `ptr`. Enfin, la dernière ligne calculera `0x7fff5fbff758-2*sizeof(unsigned int)` ce qui correspond à `0x7fff5fbff750`.

Il est intéressant pour terminer cette première discussion de l'arithmétique des pointeurs, de considérer l'exécution du fragment de code ci-dessous.

```
unsigned char* ptr_char = (unsigned char *) tab;
printf("ptr_char contient %p\n", ptr_char);
for (i = 0; i < SIZE + 1; i++) {
  printf("%X est a l'adresse %p\n", *ptr_char, ptr_char);
  ptr_char++;
}
```

L'exécution de ce fragment de code produit une sortie qu'il est intéressant d'analyser.

```
ptr_char contient 0x7fff5fbff750
4 est a l'adresse 0x7fff5fbff750
3 est a l'adresse 0x7fff5fbff751
2 est a l'adresse 0x7fff5fbff752
1 est a l'adresse 0x7fff5fbff753
```

Tout d'abord, l'initialisation du pointeur `ptr_char` a bien stocké dans ce pointeur l'adresse en mémoire du premier élément du tableau. Ensuite, comme `ptr_char` est un pointeur de type `unsigned char *`, l'expression `*ptr_char` a retourné la valeur de l'octet se trouvant à l'adresse `0x7fff5fbff750`. L'incrémentation du pointeur `ptr_char` s'est faite en respectant l'arithmétique des pointeurs. Comme `sizeof(unsigned char)` retourne `1`, la valeur stockée dans `ptr_char` a été incrémentée d'une seule unité par l'instruction `ptr_char++`. En analysant les quatre `unsigned char` se trouvant aux adresses `0x7fff5fbff750` à `0x7fff5fbff753`, on retrouve bien l'entier `0x01020304` qui avait été placé dans `tab[0]`.

2.2.6 Les structures

Outre les types de données décrits ci-dessus, les programmes informatiques doivent souvent pouvoir manipuler des données plus complexes. A titre d'exemples, un programme de calcul doit pouvoir traiter des nombres complexes, un programme de gestion des étudiants doit traiter des fiches d'étudiants avec nom, prénom, numéro de matricule, ... Dans les langages orientés objets comme Java, cela se fait en encapsulant des données de différents types avec les méthodes permettant leur traitement. C n'étant pas un langage orienté objet, il ne permet pas la création d'objets et de méthodes directement associées. Par contre, C permet de construire des types de données potentiellement complexes.

C permet la définition de structures qui combinent différents types de données simples ou structurés. Contrairement aux langages orientés objet, il n'y a pas de méthode directement associée aux structures qui sont définies. Une structure est uniquement un type de données. Voici quelques exemples de structures simples en C.

```
// structure pour stocker une coordonnée 3D
struct coord {
  int x;
  int y;
  int z;
```

```
};

struct coord point = {1, 2, 3};
struct coord p;

// structure pour stocker une fraction
struct fraction {
  int numerator;
  int denominator;
};

struct fraction demi = {1, 2};
struct fraction f;

// structure pour représenter un étudiant
struct student {
  int matricule;
  char prenom[20];
  char nom[30];
};

struct student s = {1, "Linus", "Torvalds"};
```

Le premier bloc définit une structure dénommée `coord` qui contient trois entiers baptisés `x`, `y` et `z`. Dans une structure, chaque élément est identifié par son nom et il est possible d'y accéder directement. La variable `point` est de type `struct coord` et son élément `x` est initialisé à la valeur `1` tandis que son élément `z` est initialisé à la valeur `3`. La variable `p` est également de type `struct coord` mais elle n'est pas explicitement initialisée lors de sa déclaration.

La structure `struct fract` définit une fraction qui est composée de deux entiers qui sont respectivement le numérateur et le dénominateur. La structure `struct student` définit elle un type de données qui comprend un numéro de matricule et deux chaînes de caractères.

Les structures permettent de facilement regrouper des données qui sont logiquement reliées entre elles et doivent être manipulées en même temps. C permet d'accéder facilement à un élément d'une structure en utilisant l'opérateur '`.`'. Ainsi, la structure `point` dont nous avons parlé ci-dessus aurait pu être initialisée par les trois expressions ci-dessous :

```
point.x = 1;
point.y = 2;
point.z = 3;
```

Dans les premières versions du langage C, une structure devait nécessairement contenir uniquement des données qui ont une taille fixe, c'est-à-dire des nombres, des caractères, des pointeurs ou des tableaux de taille fixe. Il n'était pas possible de stocker des tableaux de taille variable comme une chaîne de caractères `char []`. Les compilateurs récents [C99] permettent de supporter des tableaux flexibles à l'intérieur de structures. Nous ne les utiliserons cependant pas dans le cadre de ce cours.

Les structures sont utilisées dans différentes librairies et appels systèmes sous Unix et Linux. Un exemple classique est la gestion du temps sur un système Unix. Un système informatique contient généralement une horloge dite *temps-réel* qui est en pratique construite autour d'un cristal qui oscille à une fréquence fixée. Ce cristal est piloté par un circuit électronique qui compte ses oscillations, ce qui permet de mesurer le passage du temps. Le système d'exploitation utilise cette horloge *temps réel* pour diverses fonctions et notamment la mesure du temps du niveau des applications.

Un système de type Unix maintient différentes structures qui sont associées à la mesure du temps [8]. La première sert à mesurer le nombre de secondes et de microsecondes qui se sont écoulées depuis le 1er janvier 1970. Cette structure, baptisée `struct timeval` est définie dans sys/time.h comme suit :

8. Une description plus détaillée des différentes possibilités de mesure du temps via les fonctions de la librairie standard est disponible dans le chapitre 21 du manuel de la *libc*.

```
struct timeval {
  time_t       tv_sec;   /* seconds since Jan. 1, 1970 */
  suseconds_t  tv_usec;  /* and microseconds */
};
```

Cette structure est utilisée par des appels systèmes tels que gettimeofday(2) pour notamment récupérer l'heure courante ou les appels de manipulation de timers tels que getitimer(2) / setitimer(2). Elle est aussi utilisée par la fonction time(3posix) de la librairie standard et est très utile pour mesurer les performances d'un programme.

Les structures sont également fréquemment utilisées pour représenter des formats de données spéciaux sur disque comme le format des répertoires [9] ou les formats de paquets qui sont échangés sur le réseau [10].

La définition de `struct timeval` utilise une fonctionnalité fréquemment utilisée du C : la possibilité de définir des alias pour des noms de type de données existants. Cela se fait en utilisant l'opérateur `typedef`. En C, il est possible de renommer des types de données existants. Ainsi, l'exemple ci-dessous utilise `typedef` pour définir l'alias `Entier` pour le type `int` et l'alias `Fraction` pour la structure `struct fraction`.

```
// structure pour stocker une fraction
typedef struct fraction {
  double numerator;
  double denominator;
} Fraction ;

typedef int Entier;

int main(int argc, char *argv[])
{
  Fraction demi = {1, 2};
  Entier i = 2;
  // ...
  return EXIT_SUCCESS;
}
```

Les types `Entier` et `int` peuvent être utilisés de façon interchangeable à l'intérieur du programme une fois qu'ils ont été définis.

Note : `typedef` en pratique

Le renommage de types de données a des avantages et des inconvénients dont il faut être conscient pour pouvoir l'utiliser à bon escient. L'utilisation de `typedef` peut faciliter la lecture et la portabilité de certains programmes. Lorsqu'un `typedef` est associé à une structure, cela facilite la déclaration de variables de ce type et permet le cas échéant de modifier la structure de données ultérieurement sans pour autant devoir modifier l'ensemble du programme. Cependant, contrairement aux langages orientés objet, des méthodes ne sont pas directement associées aux structures et la modification d'une structure oblige souvent à vérifier toutes les fonctions qui utilisent cette structure. L'utilisation de `typedef` permet de clarifier le rôle de certains types de données ou valeurs de retour de fonctions. A titre d'exemple, l'appel système read(2) qui permet notamment de lire des données dans un fichier retourne le nombre d'octets qui ont été lus après chaque appel. Cette valeur de retour est de type `ssize_t`. L'utilisation de ces types permet au compilateur de vérifier que les bons types de données sont utilisés lors des appels de fonctions.

`typedef` est souvent utilisé pour avoir des identifiants de type de données plus court. Par exemple, il est très courant d'abrévier les types `unsigned` comme ci-dessous.

```
typedef unsigned int u_int_t;
typedef unsigned long u_long_t;
```

Soyez prudent si vous utilisez des `typedef` pour redéfinir des pointeurs. En C, il est tout à fait valide d'écrire les lignes suivantes.

9. Voir notamment fs(5) pour des exemples relatifs aux systèmes de fichiers. Une analyse détaillée des systèmes de fichiers sort du cadre de ce cours.
10. Parmi les exemples simples, on peut citer la structure `struct ipv6hdr` qui correspond à l'entête IPv6 et est définie dans linux/ipv6.h

```
typedef int * int_ptr;
typedef char * string;
```

Malheureusement, il y a un risque dans un grand programme que le développeur oublie que ces types de données correspondent à des pointeurs qui doivent être manipulés avec soin. Le Linux kernel coding style contient une discussion intéressante sur l'utilisation des `typedef`.

Les pointeurs sont fréquemment utilisés lors de la manipulation de structures. Lorsqu'un pointeur pointe vers une structure, il est utile de pouvoir accéder facilement aux éléments de la structure. Le langage C supporte deux notations pour représenter ces accès aux éléments d'une structure. La première notation est `(*ptr).elem` où `ptr` est un pointeur et `elem` l'identifiant d'un des éléments de la structure pointée par `ptr`. Cette notation est en pratique assez peu utilisée. La notation la plus fréquente est `ptr->elem` dans laquelle `ptr` et `->elem` sont respectivement un pointeur et un identifiant d'élément. L'exemple ci-dessous illustre l'initialisation de deux fractions en utilisant ces notations.

```
struct fraction demi, quart;
struct fraction *demi_ptr;
struct fraction *quart_ptr;

demi_ptr = &demi;
quart_ptr = &quart;

(*demi_ptr).num = 1;
(*demi_ptr).den = 2;

quart_ptr->num = 1;
quart_ptr->den = 4;
```

Les pointeurs sont fréquemment utilisés en combinaison avec des structures et on retrouve très souvent la seconde notation dans des programmes écrits en C.

2.2.7 Les fonctions

Comme la plupart des langages, le C permet de modulariser un programme en le découpant en de nombreuses fonctions. Chacune réalise une tâche simple. Tout comme Java, C permet la définition de fonctions qui ne retournent aucun résultat. Celles-ci sont de type `void` comme l'exemple trivial ci-dessous.

```
void usage()
{
  printf("Usage : ...\n");
}
```

La plupart des fonctions utiles retournent un résultat qui peut être une donnée d'un des types standard ou une structure. Cette utilisation est similaire à ce que l'on trouve dans des langages comme Java. Il faut cependant être attentif à la façon dont le langage C traite les arguments des fonctions. Le langage C utilise le *passage par valeur* des arguments. Lorsqu'une fonction est exécutée, elle reçoit les valeurs de ces arguments. Ces valeurs sont stockées dans une zone mémoire qui est locale à la fonction. Toute modification faite sur la valeur d'une variable à l'intérieur d'une fonction est donc locale à cette fonction. Les deux fonctions ci-dessous ont le même résultat et aucune des deux n'a d'effet de bord.

```
int twotimes(int n)
{
  return 2 * n;
}

int two_times(int n)
{
  n = 2 * n;
  return n;
}
```

Il faut être nettement plus attentif lorsque l'on écrit des fonctions qui utilisent des pointeurs comme arguments. Lorsqu'une fonction a un argument de type pointeur, celui-ci est passé par valeur, mais connaissant la valeur du pointeur, il est possible à la fonction de modifier le contenu de la zone mémoire pointée par le pointeur. Ceci est illustré par l'exemple ci-dessous.

```
int times_two(int *n)
{
  return (*n) + (*n);
}

int timestwo(int *n)
{
  *n = (*n) + (*n);
  return *n;
}

void f()
{
  int i = 1252;
  printf("i:%d\n", i);
  printf("times_two(&i)=%d\n", times_two(&i));
  printf("après times_two, i:%d\n", i);
  printf("timestwo(&i)=%d\n", timestwo(&i));
  printf("après timestwo, i:%d\n", i);
}
```

Lors de l'exécution de la fonction `f`, le programme ci-dessus affiche à la console la sortie suivante :

```
i:1252
times_two(&i)=2504
après times_two, i:1252
timestwo(&i)=2504
après timestwo, i:2504
```

Cet exemple illustre aussi une contrainte imposée par le langage C sur l'ordre de définition des fonctions. Pour que les fonctions `times_two` et `timestwo` puissent être utilisées à l'intérieur de la fonction `f`, il faut qu'elles aient été préalablement définies. Dans l'exemple ci-dessus, cela s'est fait en plaçant la définition des deux fonctions avant leur utilisation. C'est une règle de bonne pratique utilisable pour de petits programmes composés de quelques fonctions. Pour des programmes plus larges, il est préférable de placer au début du code source la signature des fonctions qui y sont définies. La signature d'une fonction comprend le type de valeur de retour de la fonction, son nom et les types de ses arguments. Généralement, ces déclarations sont regroupées à l'intérieur d'un *fichier header* dont le nom se termine par `.h`.

```
int times_two(int *);
int timestwo(int *);
```

Les fonctions peuvent évidemment recevoir également des tableaux comme arguments. Cela permet par exemple d'implémenter une fonction qui calcule la longueur d'une chaîne de caractères en itérant dessus jusqu'à trouver le caractère de fin de chaîne.

```
int length(char *s)
{
  int i = 0;
  while (*(s+i) != '\0')
    i++;
  return i;
}
```

Tout comme cette fonction peut accéder au ième caractère de la chaîne passée en argument, elle peut également et sans aucune restriction modifier chacun des caractères de cette chaîne. Par contre, comme le pointeur vers la chaîne de caractères est passé par valeur, la fonction ne peut pas modifier la zone mémoire qui est pointée par l'argument.

Un autre exemple de fonctions qui manipulent les tableaux sont des fonctions mathématiques qui traitent des vecteurs par exemple.

```
void plusun(int size, int *v)
{
  int i;
  for (i = 0; i < size; i++)
    v[i]++;
}

void print_vecteur(int size, int*v) {
  int i;
  printf("v={");
  for (i = 0;i < size - 1; i++)
    printf("%d,", v[i]);

  if (size > 0)
    printf("%d}", v[size - 1]);
  else
    printf("}");
}
```

Ces deux fonctions peuvent être utilisées par le fragment de code ci-dessous :

```
int vecteur[N] = {1, 2, 3, 4, 5};
plusun(N, vecteur);
print_vecteur(N, vecteur);
```

Note : Attention à la permissivité du compilateur C

Certains langages comme Java sont fortement typés et le compilateur contient de nombreuses vérifications, notamment sur les types de données utilisés, qui permettent d'éviter un grand nombre d'erreurs. Le langage C est lui nettement plus libéral. Les premiers compilateurs C étaient très permissifs notamment sur les types de données passés en arguments. Ainsi, un ancien compilateur C accepterait probablement sans broncher les appels suivants :

```
plusun(vecteur,N);
print_vecteur(N,vecteur);
```

Dans ce fragment de programme, l'appel à `print_vecteur` est tout à fait valide. Par contre, l'appel à `plusun` est lui erroné puisque le premier argument est un tableau d'entiers (ou plus précisément un pointeur vers le premier élément d'un tableau d'entiers) alors que la fonction `plusun` attend un entier. Inversement, le second argument est un entier à la place d'un tableau d'entiers. Cette erreur n'empêche pas le compilateur gcc(1) de compiler le programme correspondant. Il émet cependant le *warning* suivant :

```
warning: passing argument 1 of 'plusun' makes integer from pointer without a cast
warning: passing argument 2 of 'plusun' makes pointer from integer without a cast
```

De nombreux programmeurs débutants ignorent souvent les warnings émis par le compilateur et se contentent d'avoir un programme compilable. C'est la source de nombreuses erreurs et de nombreux problèmes. Dans l'exemple ci-dessus, l'exécution de l'appel `plusun(vecteur,N)` provoquera une tentative d'accès à la mémoire dans une zone qui n'est pas allouée au processus. Dans ce cas, la tentative d'accès est bloquée par le système et provoque l'arrêt immédiat du programme sur une *segmentation fault*. Dans d'autres cas, des erreurs plus subtiles mais du même type ont provoqué des problèmes graves de sécurité dans des programmes écrits en langage C. Nous y reviendrons ultérieurement.

Pour terminer, mentionnons que les fonctions écrites en C peuvent utiliser des structures et des pointeurs vers des structures comme arguments. Elles peuvent aussi retourner des structures comme résultat. Ceci est illustré par deux variantes de fonctions permettant d'initialiser une fraction et de déterminer si deux fractions sont égales [11]

11. Cette définition de l'égalité entre fractions suppose que les fractions à comparer sont sous forme irréductible. Le lecteur est invité à écrire la fonction générale permettant de tester l'égalité entre fractions réductibles.

```
struct fraction init(int num, int den)
{
  struct fraction f;
  f.numerator = num;
  f.denominator = den;
  return f;
}

int equal(struct fraction f1, struct fraction f2)
{
  return ((f1.numerator == f2.numerator)
          && (f1.denominator == f2.denominator));
}

int equalptr(struct fraction *f1, struct fraction *f2)
{
  return ((f1->numerator==f2->numerator)
          && (f1->denominator==f2->denominator));
}

void initptr(struct fraction *f, int num, int den)
{
  f->numerator = num;
  f->denominator = den;
}
```

Considérons d'abord les fonctions `init` et `equal`. `init` est une fonction qui construit une structure sur base d'arguments entiers et retourne la valeur construite. `equal` quant à elle reçoit comme arguments les valeurs de deux structures. Elle peut accéder à tous les éléments des structures passées en argument. Comme ces structures sont passées par valeur, toute modification aux éléments de la structure est locale à la fonction `equal` et n'est pas répercutée sur le code qui a appelé la fonction.

Les fonctions `initptr` et `equalptr` utilisent toutes les deux des pointeurs vers des `struct fraction` comme arguments. Ce faisant, elles ne peuvent modifier la valeur de ces pointeurs puisqu'ils sont passés comme valeur. Par contre, les deux fonctions peuvent bien entendu modifier les éléments de la structure qui se trouvent dans la zone de mémoire pointée par le pointeur. C'est ce que `initptr` fait pour initialiser la structure. `equalptr` par contre se contente d'accéder aux éléments des structures passées en argument sans les modifier. Le fragment de code ci-dessous illustre comment ces fonctions peuvent être utilisées en pratique.

```
struct fraction quart;
struct fraction tiers;
quart = init(1, 4);
initptr(&tiers, 1, 3);
printf("equal(tiers,quart)=%d\n", equal(tiers, quart));
printf("equalptr(&tiers,&quart)=%d\n", equalptr(&tiers, &quart));
```

2.2.8 Les expressions de manipulation de bits

La plupart des langages de programmation sont spécialisés dans la manipulation des types de données classiques comme les entiers, les réels et les chaînes de caractères. Comme nous l'avons vu, le langage C permet de traiter ces types de données. En outre, il permet au programmeur de pouvoir facilement manipuler les bits qui se trouvent en mémoire. Pour cela, le langage C définit des expressions qui correspondent à la plupart des opérations de manipulation de bits que l'on retrouve dans les langages d'assemblage. Les premières opérations sont les opérations logiques.

La première opération logique est la négation *négation* (*NOT* en anglais). Elle prend comme argument un bit et retourne le bit inverse. Comme toutes les opérations logiques, elle peut se définir simplement sous la forme d'une table de vérité. Dans des formules mathématiques, la négation est souvent représentée sous la forme $\neg A$.

A	NOT(A)
0	1
1	0

La deuxième opération est la *conjonction logique* (*AND* en anglais). Cette opération prend deux arguments binaires et retourne un résultat binaire. Dans des formules mathématiques, la conjonction logique est souvent représentée sous la forme $A \wedge B$. Elle se définit par la table de vérité suivante :

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

La troisième opération est la *disjonction logique* (*OR* en anglais). Cette opération prend deux arguments binaires. Dans des formules mathématiques, la disjonction logique est souvent représentée sous la forme $A \vee B$. Elle se définit par la table de vérité suivante.

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Enfin, une dernière opération logique intéressante est le *ou exclusif* (*XOR* en anglais). Celle-ci se définit par la table de vérité ci-dessous. Cette opération est parfois représentée mathématiquement comme $A \oplus B$.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Ces opérations peuvent être combinées entre elles. Pour des raisons technologiques, les circuits logiques implémentent plutôt les opérations NAND (qui équivaut à AND suivi de NOT) ou NOR (qui équivaut à OR suivi de NOT). Il est également important de mentionner les lois formulées par De Morgan qui peuvent se résumer par les équations suivantes :

— $\neg(A \wedge B) = \neg A \vee \neg B$
— $\neg(A \vee B) = \neg A \wedge \neg B$

Ces opérations binaires peuvent s'étendre à des séquences de bits. Voici quelques exemples qui permettent d'illustrer ces opérations sur des octets.

```
~ 00000000 = 11111111
11111010 & 01011111 = 01011010
11111010 | 01011111 = 11111111
11111010 ^ 01011111 = 10100101
```

En C, ces expressions logiques s'utilisent comme dans le fragment de code suivant. En général, elles s'utilisent sur des representations non signées, souvent des `unsigned char` ou des `unsigned int`.

```
r = ~a;     // négation bit à bit
r = a & b; // conjonction bit à bit
r = a | b; // disjonction bit à bit
r = a ^ b; // xor bit à bit
```

En pratique, les opérations logiques sont utiles pour effectuer des manipulations au niveau des bits de données stockées en mémoire. Une utilisation fréquente dans certaines applications réseaux ou systèmes est de forcer certains bits à la prendre la valeur `0` ou `1`. La conjonction logique permet de forcer facilement un bit à zéro tandis que la disjonction logique permet de forcer facilement un bit à un. L'exemple ci-dessous montre comment forcer les valeurs de certains bits dans un `unsigned char`. Il peut évidemment se généraliser à des séquences de bits plus longues.

```
r = c & 0x7E; // 0b01111110 force les bits de poids faible et fort à 0
r = d | 0x18; // 0b00011000 force les bits 4 et 3 à 1
```

L'opération XOR joue un rôle important dans certaines applications. La plupart des méthodes de chiffrement et de déchiffrement utilisent de façon extensive cette opération. Une des propriétés intéressantes de l'opération XOR est que $(A \oplus B) \oplus B = A$. Cette propriété est largement utilisée par les méthodes de chiffrement. La méthode développée par Vernam au début du vingtième siècle s'appuie sur l'opération XOR. Pour transmettre un message M de façon sûre, elle applique l'opération XOR bit à bit entre tous les bits du message M et une clé K doit avoir au moins le même nombre de bits que M. Si cette clé K est totalement aléatoire et n'est utilisée qu'une seule fois, alors on parle de *one-time-pad*. On peut montrer que dans ce cas, la méthode de chiffrement est totalement sûre. En pratique, il est malheureusement difficile d'avoir une clé totalement aléatoire qui soit aussi longue que le message à transmettre. Le programme ci-dessous implémente cette méthode de façon triviale. La fonction memfrob(3) de la librairie *GNU* utilise également un chiffrement via un XOR.

```
int main(int argc, char* argv[])
{
  if (argc != 2)
    usage("ce programme prend une clé comme argument");

  char *key = argv[1];
  char c;
  int i = 0;
  while (((c = getchar()) != EOF) && (i < strlen(key))) {
    putchar(c ^ *(key + i));
    i++;
  }
  return EXIT_SUCCESS;
}
```

Note : Ne pas confondre expressions logiques et opérateurs binaires

En C, les symboles utilisés pour les expressions logiques (`||` et `&&`) sont très proches de ceux utilisés pour représenter les opérateurs binaires (| et &). Il arrive parfois qu'un développeur confonde `&` avec `&&`. Malheureusement, le compilateur ne peut pas détecter une telle erreur car dans les deux cas le résultat attendu est généralement du même type.

```
0b0100 & 0b0101 = 0b0100
0b0100 && 0b0101 = 0b0001
0b0100 | 0b0101 = 0b0101
0b0100 || 0b0101 = 0b0001
```

Un autre point important à mentionner concernant les expressions logiques est qu'en C celles-ci sont évaluées de gauche à droite. Cela implique que dans l'expression `( expr1 && expr2 )`, le compilateur C va d'abord évaluer l'expression `expr1`. Si celle-ci est évaluée à la valeur `0`, la seconde expression ne sera pas évaluée. Cela peut être très utile lorsque l'on doit exécuter du code si un pointeur est non `NULL` et qu'il pointe vers une valeur donnée. Dans ce cas, la condition sera du type `((ptr != NULL) && (ptr->den > 0))`.

Pour terminer, le langage C supporte des expressions permettant le décalage à gauche ou à droite à l'intérieur d'une suite de bits non signée.

- `a = n >> B` décale les bits représentants `n` de `B` bits vers la droite et place le résultat dans la variable `a`
- `a = n << B` décale les bits représentants `n` de `B` bits vers la gauche et place le résultat dans la variable `a`

Ces opérations de décalage permettent différentes manipulations de bits. A titre d'exemple, la fonction `int2bin` utilise à la fois des décalages et des masques pour calculer la représentation binaire d'un entier non-signé et la placer dans une chaîne de caractères.

```
#define BITS_INT 32
// str[BITS_INT]
void int2bin(unsigned int num, char *str)
{
```

```
  int i;
  str[BITS_INT] = '\0';
  for (i = BITS_INT - 1; i >= 0; i--) {
    if ((num & 1) == 1)
      str[i] = '1';
    else
      str[i] = '0';
    num = num >> 1;
  }
}
```

2.3 Déclarations

Durant les chapitres précédents, nous avons principalement utilisé des variables locales. Celles-ci sont déclarées à l'intérieur des fonctions où elles sont utilisées. La façon dont les variables sont déclarées est importante dans un programme écrit en langage C. Dans cette section nous nous concentrerons sur des programmes C qui sont écrits sous la forme d'un seul fichier source. Nous verrons plus tard comment découper un programme en plusieurs modules qui sont répartis dans des fichiers différents et comment les variables peuvent y être déclarées.

La première notion importante concernant la déclaration des variables est leur *portée*. La portée d'une variable peut être définie comme étant la partie du programme où la variable est accesible et où sa valeur peut être modifiée. Le langage C définit deux types de portée à l'intérieur d'un fichier C. La première est la *portée globale*. Une variable qui est définie en dehors de toute définition de fonction a une portée globale. Une telle variable est accessible dans toutes les fonctions présentes dans le fichier. La variable `g` dans l'exemple ci-dessous a une portée globale.

```
float g;   // variable globale

int f(int i) {
int n;   // variable locale
// ...
for(int j=0;j<n;j++) {  // variable locale
  // ...
  }
//...
for(int j=0;j<n;j++) {  // variable locale
  // ...
  }
}
```

Dans un fichier donné, il ne peut évidemment pas y avoir deux variables globales qui ont le même identifiant. Lorsqu'une variable est définie dans un *bloc*, la portée de cette variable est locale à ce bloc. On parle dans ce cas de *portée locale*. La variable locale n'existe pas avant le début du bloc et n'existe plus à la fin du bloc. Contrairement aux identifiants de variables globales qui doivent être uniques à l'intérieur d'un fichier, il est possible d'avoir plusieurs variables locales qui ont le même identifiant à l'intérieur d'un fichier. C'est fréquent notamment pour les définitions d'arguments de fonction et les variables de boucles. Dans l'exemple ci-dessus, les variables `n` et `j` ont une portée locale. La variable `j` est définie dans deux blocs différents à l'intérieur de la fonction `f`.

Le programme `/C/S3-src/portee.c` illustre la façon dont le compilateur C gère la portée de différentes variables.

```
int g1;
int g2=1;

void f(int i) {
  int loc;    //def1a
  int loc2=2; //def2a
  int g2=-i*i;
  g1++;

  printf("[f-%da] \t\t %d \t %d \t %d \t %d\n",i,g1,g2,loc,loc2);
```

```
  loc=i*i;
  g1++;
  g2++;
  printf("[f-%db] \t\t %d \t %d \t %d \t %d\n",i,g1,g2,loc,loc2);

}


int main(int argc, char *argv[]) {
  int loc; //def1b
  int loc2=1; //def2b

  printf("Valeurs de : \t g1 \t g2\t loc\t loc2\n");
  printf("=================================================\n");

  printf("[main1] \t %d \t %d \t %d \t %d\n",g1,g2,loc,loc2);

  loc=1252;
  loc2=1234;
  g1=g1+1;
  g1=g1+2;

  printf("[main2] \t %d \t %d \t %d \t %d\n",g1,g2,loc,loc2);

  for(int i=1;i<3;i++) {
    int loc=i; //def1c
    int g2=-i;
    loc++;
    g1=g1*2;
    f(i);
    printf("[main-for-%d] \t %d \t %d \t %d \t %d\n",i,g1,g2,loc,loc2);
  }
  f(7);
  g1=g1*3;
  g2=g2+2;
  printf("[main3] \t %d \t %d \t %d \t %d\n",g1,g2,loc,loc2);

  return(EXIT_SUCCESS);
}
```

Ce programme contient deux variables qui ont une portée globale : `g1` et `g2`. Ces deux variables sont définies en dehors de tout bloc. En pratique, elles sont généralement déclarées au début du fichier, même si le compilateur C accepte un définition en dehors de tout bloc et donc par exemple en fin de fichier. La variable globale `g1` n'est définie qu'une seule fois. Par contre, la variable `g2` est définie avec une portée globale et est redéfinie à l'intérieur de la fonction `f` ainsi que dans la boucle `for` de la fonction `main`. Redéfinir une variable globale de cette façon n'est pas du tout une bonne pratique, mais cela peut arriver lorsque par mégarde on importe un fichier header qui contient une définition de variable globale. Dans ce cas, le compilateur C utilise la variable qui est définie dans le bloc le plus proche. Pour la variable `g2`, c'est donc la variable locale `g2` qui est utilisée à l'intérieur de la boucle `for` ou de la fonction `f`.

Lorsqu'un identifiant de variable locale est utilisé à plusieurs endroits dans un fichier, c'est la définition la plus proche qui est utilisée. L'exécution du programme ci-dessus illustre cette utilisation des variables globales et locales.

```
Valeurs de :          g1          g2          loc          loc2
=================================================
[main1]          0          1          0          1
[main2]          3          1          1252          1234
[f-1a]                  7          -1          0          2
[f-1b]                  8          0          1          2
[main-for-1]          8          -1          2          1234
[f-2a]                  17          -4          0          2
```

```
[f-2b]                  18          -3          4          2
[main-for-2]          18          -2          3          1234
[f-7a]                  19          -49         0          2
[f-7b]                  20          -48         49         2
[main3]          60          3          1252          1234
```

Note : Utilisation des variables

En pratique, les variables globales doivent être utilisées de façon parcimonieuse et il faut limiter leur utilisation aux données qui doivent être partagées par plusieurs fonctions à l'intérieur d'un programme. Lorsqu'une variable globale a été définie, il est préférable de ne pas réutiliser son identifiant pour une variable locale. Au niveau des variables locales, les premières versions du langage C imposaient leur définition au début des blocs. Les standards récents [C99] autorisent la déclaration de variables juste avant leur première utilisation un peu comme en Java.

Les versions récentes de C [C99] permettent également de définir des variables dont la valeur sera constante durant toute l'exécution du programme. Ces déclarations de ces constants sont préfixées par le mot-clé `const` qui joue le même rôle que le mot clé `final` en Java.

```
// extrait de <math.h>
#define M_PI   3.14159265358979323846264338327950288;

const double pi=3.14159265358979323846264338327950288;

const struct fraction {
  int num;
  int denom;
}  demi={1,2};
```

Il y a deux façons de définir des constantes dans les versions récentes de C [C99]. La première est via la macro `#define` du préprocesseur. Cette macro permet de remplacer une chaîne de caractères (par exemple `M_PI` qui provient de math.h) par un nombre ou une autre chaîne de caractères. Ce remplacement s'effectue avant la compilation. Dans le cas de `M_PI` ci-dessus, le préprocesseur remplace toute les occurences de cette chaîne de caractères par la valeur numérique de π. Lorsqu'une variable `const` est utilisée, la situation est un peu différente. Le préprocesseur n'intervient pas. Par contre, le compilateur réserve une zone mémoire pour la variable qui a été définie comme constante. Cela a deux avantages par rapport à l'utilisation de `#define`. Premièrement, il est possible de définir comme constante n'importe quel type de données en C, y compris des structures ou des pointeurs alors qu'avec un `#define` on ne peut définir que des nombres ou des chaînes de caractères. Ensuite, comme une `const` est stockée en mémoire, il est possible d'obtenir son adresse et de l'examiner via un *debugger*.

2.4 Unions et énumérations

Les structures que nous avons présentées précédemment permettent de combiner plusieurs données de types primitifs différents entre elles. Outre ces structures (`struct`), le langage C supporte également les `enum` et les `union`. Le mot-clé `enum` est utilisé pour définir un type énuméré, c'est-à-dire un type de donnée qui permet de stocker un nombre fixe de valeurs. Quelques exemples classiques sont repris dans le fragment de programme ci-dessous :

```
// les jours de la semaine
typedef enum  {
      monday, tuesday, wednesday, thursday, friday, saturday, sunday
   } day;

// jeu de carte
typedef enum {
  coeur, carreau, trefle, pique
} carte;

// bits
typedef enum {
```

```
        BITRESET = 0,
        BITSET = 1
} bit_t;
```

Le premier `enum` permet de définir le type de données `day` qui contient une valeur énumérée pour chaque jour de la semaine. L'utilisation d'un type énuméré rend le code plus lisible que simplement l'utilisation de constantes définies via le préprocesseur.

```
  bit_t bit=BITRESET;
  day jour=monday;
  if(jour==saturday||jour==sunday)
    printf("Congé\n");
```

En pratique, lors de la définition d'un type énuméré, le compilateur C associe une valeur entière à chacune des valeurs énumérées. Une variable permettant de stocker la valeur d'un type énuméré occupe la même zone mémoire qu'un entier.

Outre les structures, le langage C supporte également les unions. Alors qu'une structure permet de stocker plusieurs données dans une même zone mémoire, une `union` permet de réserver une zone mémoire pour stocker une données parmi plusieurs types possibles. Une `union` est parfois utilisée pour minimiser la quantité de mémoire utilisée pour une structure de données qui peut contenir des données de plusieurs types. Pour bien comprendre la différence entre une `union` et une `struct`, considérons l'exemple ci-dessous.

```
struct s_t {
  int i;
  char c;
} s;

union u_t {
  int i;
  char c;
} u;
```

Une union, `u` et une structure, `s` sont déclarées dans ce fragment de programme.

```
  // initialisation
  s.i=1252;
  s.c='A';
  u.i=1252;
  // u contient un int
  u.c='Z';
  // u contient maintenant un char (et u.i est perdu)
```

La structure `s` peut contenir à la fois un entier et un caractère. Par contre, l'`union u`, peut elle contenir un entier (`u.i`) *ou* un caractère (`u.c`), mais jamais les deux en même temps. Le compilateur C alloue la taille pour l'`union` de façon à ce qu'elle puisse contenir le type de donnée se trouvant dans l'`union` nécessitant le plus de mémoire. Si les unions sont utiles dans certains cas très particulier, il faut faire très attention à leur utilisation. Lorsqu'une `union` est utilisée, le compilateur C fait encore moins de vérifications sur les types de données et le code ci-dessous est considéré comme valide par le compilateur :

```
  u.i=1252;
  printf("char : %c\n", u.c);
```

Lors de son exécution, la zone mémoire correspondant à l'union `u` sera simplement interprétée comme contenant un `char`, même si on vient d'y stocker un entier. En pratique, lorsqu'une `union` est vraiment nécessaire pour des raisons d'économie de mémoire, on l'encapsulera dans une `struct` en utilisant un type énuméré qui permet de spécifier le type de données qui est présent dans l'`union`.

```
typedef enum { INTEGER, CHAR } Type;

typedef struct
{
```

```
  Type type;
  union {
  int i;
  char c;
  } x;
} Value;
```

Le programmeur pourra alors utiliser cette structure en indiquant explicitement le type de données qui y est actuellement stocké comme suit.

```
Value v;
v.type=INTEGER;
v.x.i=1252;
```

2.5 Organisation de la mémoire

Lors de l'exécution d'un programme en mémoire, le système d'exploitation charge depuis le système de fichier le programme en langage machine et le place à un endroit convenu en mémoire. Lorsqu'un programme s'exécute sur un système Unix, la mémoire peut être vue comme étant divisée en six zones principales. Ces zones sont représentées schématiquement dans la figure ci-dessous.

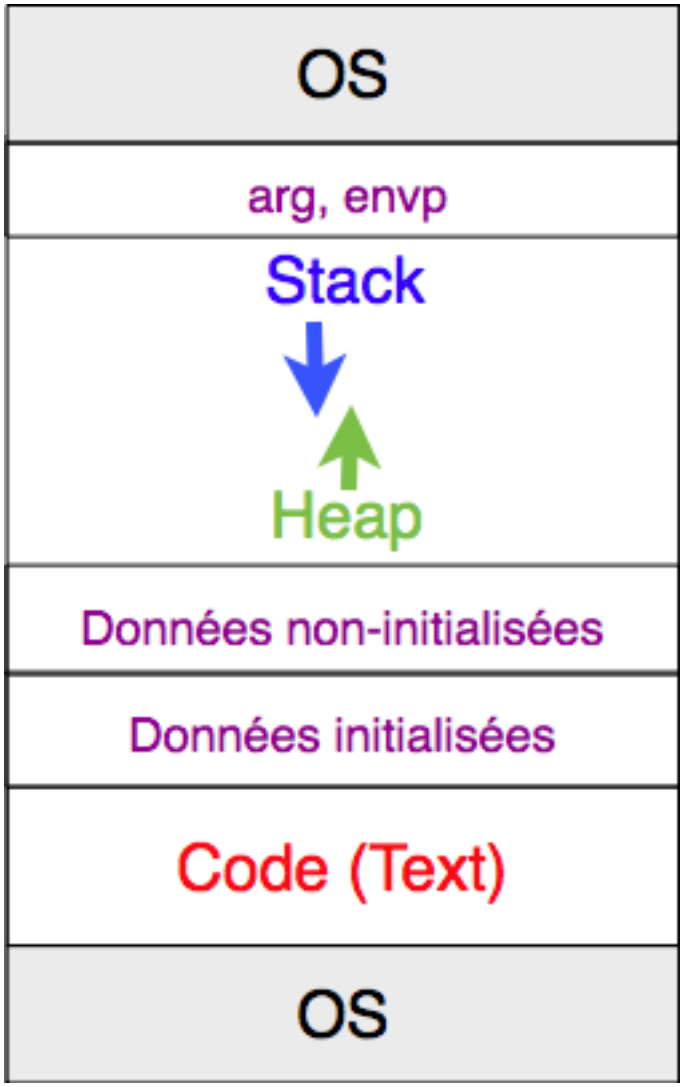


FIGURE 2.2 – Organisation d'un programme Linux en mémoire

La figure ci-dessus présente une vision schématique de la façon dont un processus Linux est organisé en mémoire centrale. Il y a d'abord une partie de la mémoire qui est réservée au système d'exploitation (OS dans la figure). Cette zone est représentée en grisé dans la figure.

2.5.1 Le segment text

La première zone est appelée par convention le *segment text*. Cette zone se situe dans la partie basse de la mémoire [12]. C'est dans cette zone que sont stockées toutes les instructions qui sont exécutées par le micro-processeur. Elle est généralement considérée par le système d'exploitation comme étant uniquement accessible en lecture. Si un programme tente de modifier son *segment text*, il sera immédiatement interrompu par le système d'exploitation. C'est dans le segment text que l'on retrouvera les instructions de langage machine correspondant aux fonctions de calcul et d'affichage du programme. Nous en reparlerons lorsque nous présenterons le fonctionnement du langage d'assemblage.

2.5.2 Le segment des données initialisées

La deuxième zone, baptisée *segment des données initialisées*, contient l'ensemble des données et chaînes de caractères qui sont utilisées dans le programme. Ce segment contient deux types de données. Tout d'abord, il comprend l'ensemble des variables globales explicitement initialisées par le programme (dans le cas contraire, elles sont initialisées à zéro par le compilateur et appartiennent alors au *segment des données non-initialisées*). Ensuite, les constantes et les chaînes de caractères utilisées par le programme.

```
#define MSG_LEN 10
int g;   // initialisé par le compilateur
int g_init=1252;
const int un=1;
int tab[3]={1,2,3};
int array[10000];
char cours[]="SINF1252";
char msg[MSG_LEN]; // initialisé par le compilateur

int main(int argc, char *argv[]) {
  int i;
  printf("g est à l'adresse %p et initialisée à %d\n",&g,g);
  printf("msg est à l'adresse %p contient les caractères :",msg);
  for(i=0;i<MSG_LEN;i++)
    printf(" %x",msg[i]);
  printf("\n");
  printf("Cours est à l'adresse %p et contient : %s\n",&cours,cours);
  return(EXIT_SUCCESS);
}
```

Dans le programme ci-dessus, la variable `g_init`, la constante `un` et les tableaux `tab` et `cours` sont dans la zone réservée aux variables initialisées. En pratique, leur valeur d'initialisation sera chargée depuis le fichier exécutable lors de son chargement en mémoire. Il en va de même pour toutes les chaînes de caractères qui sont utilisées comme arguments aux appels à `printf(3)`.

L'exécution de ce programme produit la sortie standard suivante.

```
g est à l'adresse 0x60aeac et initialisée à 0
msg est à l'adresse 0x60aea0 contient les caractères : 0 0 0 0 0 0 0 0 0 0
Cours est à l'adresse 0x601220 et contient : SINF1252
```

Cette sortie illustre bien les adresses où les variables globales sont stockées. La variable globale `msg` fait notamment partie du *segment des données non-initialisées*.

12. Dans de nombreuses variantes de Unix, il est possible de connaître le sommet du segment *text* d'un processus grâce à la variable *etext*. Cette variable, de type `char` est initialisée par le système au chargement du processus. Elle doit être déclarée comme variable de type `extern char etext` et son adresse (`&etext`) correspond au sommet du segment text.

2.5.3 Le segment des données non-initialisées

La troisième zone est le *segment des données non-initialisées*, réservée aux variables non-initialisées. Cette zone mémoire est initialisée à zéro par le système d'exploitation lors du démarrage du programme. Dans l'exemple ci-dessus, c'est dans cette zone que l'on stockera les valeurs de la variable `g` et des tableaux `array` et `msg`.

Note : Initialisation des variables

Un point important auquel tout programmeur C doit faire attention est l'initialisation correcte de l'ensemble des variables utilisées dans un programme. Le compilateur C est nettement plus permissif qu'un compilateur Java et il autorisera l'utilisation de variables avant qu'elles n'aient été explicitement initialisées, ce qui peut donner lieu à des erreurs parfois très difficiles à corriger.

En C, par défaut les variables globales qui ne sont pas explicitement initialisées dans un programme sont initialisées à la valeur zéro par le compilateur. Plus précisément, la zone mémoire qui correspond à chaque variable globale non-explicitement initialisée contiendra des bits valant 0. Pour les variables locales, le langage C n'impose aucune initialisation par défaut au compilateur. Par souci de performance et sachant qu'un programmeur ne devrait jamais utiliser de variable locale non explicitement initialisée, le compilateur C n'initialise pas par défaut la valeur de ces variables. Cela peut avoir des conséquences ennuyeuses comme le montre l'exemple ci-dessous.

```
#define ARRAY_SIZE 1000

// initialise un tableau local
void init(void) {
  long a[ARRAY_SIZE];
  for(int i=0;i<ARRAY_SIZE;i++) {
    a[i]=i;
  }
}

// retourne la somme des éléments
// d'un tableau local
long read(void) {
  long b[ARRAY_SIZE];
  long sum=0;
  for(int i=0;i<ARRAY_SIZE;i++) {
    sum+=b[i];
  }
  return sum;
}
```

Cet extrait de programme contient deux fonctions erronées. La seconde, baptisée `read(void)` déclare un tableau local et retourne la somme des éléments de ce tableau sans l'initialiser. En Java, une telle utilisation d'un tableau non-initialisé serait détectée par le compilateur. En C, elle est malheureusement valide (mais fortement découragée évidemment). La première fonction, `init(void)` se contente d'initialiser un tableau local mais ne retourne aucun résultat. Cette fonction ne sert a priori à rien puisqu'elle n'a aucun effet sur les variables globales et ne retourne aucun résultat. L'exécution de ces fonctions via le fragment de code ci-dessous donne cependant un résultat interpellant.

```
  printf("Résultat de read() avant init(): %ld\n",read());
  init();
  printf("Résultat de read() après init() : %ld\n",read());

Résultat de read() avant init(): 7392321044987950589
Résultat de read() après init() : 499500
```

2.5.4 Le tas (ou heap)

La quatrième zone de la mémoire est le *tas* (ou *heap* en anglais). Vu l'importance pratique de la terminologie anglaise, c'est celle-ci que nous utiliserons dans le cadre de ce document. C'est une des deux zones dans laquelle

un programme peut obtenir de la mémoire supplémentaire pour stocker de l'information. Un programme peut y réserver une zone permettant de stocker des données et y associer un pointeur.

Le système d'exploitation mémorise, pour chaque processus en cours d'exécution, la limite supérieure de son *heap*. Le système d'exploitation permet à un processus de modifier la taille de son heap via les appels systèmes brk(2) et sbrk(2). Malheureusement, ces deux appels systèmes se contentent de modifier la limite supérieure du *heap* sans fournir une API permettant au processus d'y allouer efficacement des blocs de mémoire. Rares sont les processus qui utilisent directement brk(2) si ce n'est sous la forme d'un appel à `sbrk(0)` de façon à connaître la limite supérieure actuelle du *heap*.

En C, la plupart des processus allouent et libèrent de la mémoire en utilisant les fonctions malloc(3) et free(3) qui font partie de la librairie standard.

La fonction malloc(3) prend comme argument la taille (en bytes) de la zone mémoire à allouer. La signature de la fonction malloc(3) demande que cette taille soit de type `size_t`, c'est-à-dire le type retourné par l'expression `sizeof`. Il est important de toujours utiliser `sizeof` lors du calcul de la taille d'une zone mémoire à allouer. malloc(3) retourne normalement un pointeur de type `(void *)`. Ce type de pointeur est le type par défaut pour représenter dans un programme C une zone mémoire qui ne pointe pas vers un type de données particulier. En pratique, un programme va généralement utiliser malloc(3) pour allouer de la mémoire pour stocker différents types de données et le pointeur retourné par malloc(3) sera *casté* dans un pointeur du bon type.

Note : typecast en langage C

Comme le langage Java, le langage C supporte des conversions implicites et explicites entre les différents types de données. Ces conversions sont possibles entre les types primitifs et les pointeurs. Nous les rencontrerons régulièrement, par exemple lorsqu'il faut récupérer un pointeur alloué par malloc(3) ou le résultat de `sizeof`. Contrairement au compilateur Java, le compilateur C n'émet pas toujours de message de *warning* lors de l'utilisation de typecast qui risque d'engendrer une perte de précision. Ce problème est illustré par l'exemple suivant avec les nombres.

```
int i=1;
float f=1e20;
double d=1e100;

printf("i [int]: %d, [float]:%f, [double]:%f\n",i,(float)i,(double)i);
printf("f [int]: %d, [float]:%f, [double]:%f\n",(int)f,f,(double)f);
printf("d [int]: %d, [float]:%f, [double]:%f\n",(int)d,(float)d,d);
printf("sizeof -> int:%d float:%d double:%d\n",(int)sizeof(int), (int)sizeof(float), (int)s
```

La fonction de la librairie free(3) est le pendant de malloc(3). Elle permet de libérer la mémoire qui a été allouée par malloc(3). Elle prend comme argument un pointeur dont la valeur a été initialisée par malloc(3) et libère la zone mémoire qui avait été allouée par malloc(3) pour ce pointeur. La valeur du pointeur n'est pas modifiée, mais après libération de la mémoire il n'est évidemment plus possible [13] d'accéder aux données qui étaient stockées dans cette zone.

Le programme ci-dessous illustre l'utilisation de malloc(3) et free(3).

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


typedef struct fraction {
  int num;
  int den;
} Fraction;

void error(char *msg) {
```

13. Pour des raisons de performance, le compilateur C ne génère pas de code permettant de vérifier automatiquement qu'un accès via un pointeur pointe vers une zone de mémoire qui est libre. Il est donc parfois possible d'accéder à une zone mémoire qui a été libérée, mais le programme n'a aucune garantie sur la valeur qu'il y trouvera. Ce genre d'accès à des zones mémoires libérées doit bien entendu être complètement proscrit.

```
  fprintf(stderr,"Erreur :%s\n",msg);
  exit(EXIT_FAILURE);
}

int main(int argc, char *argv[]) {
  int size=1;
  if(argc==2)
    size=atoi(argv[1]);

  char * string;
  printf("Valeur du pointeur string avant malloc : %p\n",string);
  string=(char *) malloc((size+1)*sizeof(char));
  if(string==NULL)
    error("malloc(string)");

  printf("Valeur du pointeur string après malloc : %p\n",string);
  int *vector;
  vector=(int *)malloc(size*sizeof(int));
  if(vector==NULL)
    error("malloc(vector)");

  Fraction * fract_vect;
  fract_vect=(Fraction *) malloc(size*sizeof(Fraction));
  if(fract_vect==NULL)
    error("malloc(fract_vect)");

  free(string);
  printf("Valeur du pointeur string après free   : %p\n",string);
  string=NULL;
  free(vector);
  vector=NULL;
  free(fract_vect);
  fract_vect=NULL;

  return(EXIT_SUCCESS);
}
```

Ce programme alloue trois zones mémoires. Le pointeur vers la première est sauvé dans le pointeur `string`. Elle est destinée à contenir une chaîne de `size` caractères (avec un caractère supplémentaire pour stocker le caractère `\0` de fin de chaîne). Il y a deux points à remarquer concernant cette allocation. Tout d'abord, le pointeur retourné par malloc(3) est casté en un `char *`. Cela indique au compilateur que `string` va bien contenir un pointeur vers une chaîne de caractères. Ce cast explicite rend le programme plus clair. Ensuite, la valeur de retour de malloc(3) est systématiquement testée. malloc(3) peut en effet retourner `NULL` lorsque la mémoire est remplie. Cela a peu de chance d'arriver dans un programme de test tel que celui-ci, mais tester les valeurs de retour des fonctions de la librairie est une bonne habitude à prendre lorsque l'on programme sous Unix. Le second pointeur, `vector` pointe vers une zone destiné à contenir un tableau d'entiers. Le dernier pointeur, `fract_vect` pointe vers une zone qui pourra stocker un tableau de `Fraction`. Lors de son exécution, le programme affiche la sortie suivante.

```
Valeur du pointeur string avant malloc : 0x7fff5fbfe1d8
Valeur du pointeur string après malloc : 0x100100080
Valeur du pointeur string après free   : 0x100100080
```

Dans cette sortie, on remarque que l'appel à fonction free(3) libère la zone mémoire, mais ne modifie pas la valeur du pointeur correspondant. Le programmeur doit explicitement remettre le pointeur d'une zone mémoire libérée à `NULL`.

Un autre exemple d'utilisation de malloc(3) est la fonction `duplicate` ci-dessous qui permet de retourner une copie d'une chaîne de caractères. Il est important de noter qu'en C la fonction strlen(3) retourne la longueur de la chaîne de caractères passée en argument sans prendre en compte le caractère `\0` qui marque sa fin. C'est la raison pour laquelle malloc(3) doit réserver un bloc de mémoire en plus. Même si généralement les `char` occupent un octet en mémoire, il est préférable d'utiliser explicitement `sizeof(char)` lors du calcul de l'espace mémoire nécessaire pour un type de données.

```
#include <string.h>

char *duplicate(char * str) {
  int i;
  size_t len=strlen(str);
  char *ptr=(char *)malloc(sizeof(char)*(len+1));
  if(ptr!=NULL) {
    for(i=0;i<len+1;i++) {
        *(ptr+i)=*(str+i);
      }
  }
  return ptr;
}
```

malloc(3) et free(3) sont fréquemment utilisés dans des programmes qui manipulent des structures de données dont la taille varie dans le temps. C'est le cas pour les différents sortes de listes chaînées, les piles, les queues, les arbres, ... L'exemple ci-dessous (`/C/S3-src/stack.c`) illustre l'implémentation d'une pile simple en C. Le pointeur vers le sommet de la pile est défini comme une variable globale. Chaque élément de la pile est représenté comme un pointeur vers une structure qui contient un pointeur vers la donnée stockée (dans cet exemple des fractions) et l'élément suivant sur la pile. Les fonctions `push` et `pop` permettent respectivement d'ajouter un élément et de retirer un élément au sommet de la pile. La fonction `push` alloue la mémoire nécessaire avec malloc(3) tandis que la fonction `pop` utilise free(3) pour libérer la mémoire dès qu'un élément est retiré.

```
typedef struct node_t
{
  struct fraction_t *data;
  struct node_t *next;
} node;

struct node_t *stack; // sommet de la pile

// ajoute un élément au sommet de la pile
void push(struct fraction_t *f)
{
  struct node_t *n;
  n=(struct node_t *)malloc(sizeof(struct node_t));
  if(n==NULL)
    exit(EXIT_FAILURE);
  n->data = f;
  n->next = stack;
  stack = n;
}
// retire l'élément au sommet de la pile
struct fraction_t * pop()
{
  if(stack==NULL)
    return NULL;
  // else
  struct fraction_t *r;
  struct node_t *removed=stack;
  r=stack->data;
  stack=stack->next;
  free(removed);
  return (r);
}
```

Ces fonctions peuvent être utilisées pour empiler et dépiler des fractions sur une pile comme dans l'exemple ci-dessous. La fonction `display` permet d'afficher sur *stdout* le contenu de la pile.

```
// affiche le contenu de la pile
void display()
{
```

```
  struct node_t *t;
  t = stack;
  while(t!=NULL) {
    if(t->data!=NULL) {
      printf("Item at addr %p  :  Fraction %d/%d   Next %p\n",t,t->data->num,t->data->den,t->next
    }
    else {
      printf("Bas du stack %p\n",t);
    }
    t=t->next;
  }
}

// exemple
int main(int argc, char *argv[]) {

  struct fraction_t demi={1,2};
  struct fraction_t tiers={1,3};
  struct fraction_t quart={1,4};
  struct fraction_t zero={0,1};

  // initialisation
  stack = (struct node_t *)malloc(sizeof(struct node_t));
  stack->next=NULL;
  stack->data=NULL;

  display();
  push(&zero);
  display();
  push(&demi);
  push(&tiers);
  push(&quart);
  display();

  struct fraction_t *f=pop();
  if(f!=NULL)
    printf("Popped : %d/%d\n",f->num,f->den);

  return(EXIT_SUCCESS);
}
```

Lors de son exécution le programme `/C/S3-src/stack.c` présenté ci-dessus affiche les lignes suivantes sur sa sortie standard.

```
Bas du stack 0x100100080
Item at addr 0x100100090  :  Fraction 0/1   Next 0x100100080
Bas du stack 0x100100080
Item at addr 0x1001000c0  :  Fraction 1/4   Next 0x1001000b0
Item at addr 0x1001000b0  :  Fraction 1/3   Next 0x1001000a0
Item at addr 0x1001000a0  :  Fraction 1/2   Next 0x100100090
Item at addr 0x100100090  :  Fraction 0/1   Next 0x100100080
Bas du stack 0x100100080
Popped : 1/4
```

Le tas (ou *heap*) joue un rôle très important dans les programmes C. Les données qui sont stockées dans cette zone de la mémoire sont accessibles depuis toute fonction qui possède un pointeur vers la zone correspondante

---

Note : Ne comptez jamais sur les free(3) implicites

Un programmeur débutant qui expérimente avec malloc(3) pourrait écrire le code ci-dessous et conclure que comme celui-ci s'exécute correctement, il n'est pas nécessaire d'utiliser free(3). Lors de l'exécution d'un programme, le système d'exploitation réserve de la mémoire pour les différents segments du programme et ajuste si nécessaire cette allocation durant l'exécution du programme. Lorsque le programme se termine, via `return` dans

la fonction `main` ou par un appel explicite à exit(2), le système d'exploitation libère tous les segments utilisés par le programme, le text, les données, le tas et la pile. Cela implique que le système d'exploitation effectue un appel implicite à free(3) à la terminaison d'un programme.

```
#define LEN 1024
int main(int argc, char *argv[]) {

  char *str=(char *) malloc(sizeof(char)*LEN);
  for(int i=0;i<LEN-1;i++) {
    *(str+i)='A';
  }
  *(str+LEN)='\0';  // fin de chaîne
  return(EXIT_SUCCESS);
}
```

Un programmeur ne doit cependant *jamais* compter sur cet appel implicite à free(3). Ne pas libérer la mémoire lorsqu'elle n'est plus utilisée est un problème courant qui est généralement baptisé *memory leak*. Ce problème est particulièrement gênant pour les processus tels que les serveurs Internet qui ne se terminent pas ou des processus qui s'exécutent longtemps. Une petite erreur de programmation peut causer un *memory leak* qui peut après quelque temps consommer une grande partie de l'espace mémoire inutilement. Il est important d'être bien attentif à l'utilisation correcte de malloc(3) et de free(3) pour toutes les opérations d'allocation et de libération de la mémoire.

---

malloc(3) est la fonction d'allocation de mémoire la plus fréquemment utilisée [14]. La librairie standard contient cependant d'autres fonctions permettant l'allocation et la réallocation de mémoire. calloc(3) est nettement moins utilisée que malloc(3). Elle a pourtant un avantage majeur par rapport à malloc(3) puisqu'elle initialise à zéro la zone de mémoire allouée. malloc(3) se contente d'allouer la zone de mémoire mais n'effectue aucune initialisation. Cela permet à malloc(3) d'être plus rapide, mais le programmeur ne doit jamais oublier qu'il ne peut pas utiliser malloc(3) sans initialiser la zone mémoire allouée. Cela peut s'observer en pratique avec le programme ci-dessous. Il alloue une zone mémoire pour `v1`, l'initialise puis la libère. Ensuite, le programme alloue une nouvelle zone mémoire pour `v2` et y retrouve les valeurs qu'il avait stocké pour `v1` précédemment. En pratique, n'importe quelle valeur pourrait se trouver dans la zone retournée par *malloc(3)*.

```
#define LEN 1024

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

  int *v1;
  int *v2;
  int sum=0;
  v1=(int *)malloc(sizeof(int)*LEN);
  for(int i=0;i<LEN;i++) {
    sum+=*(v1+i);
    *(v1+i)=1252;
  }
  printf("Somme des éléments de v1 : %d\n", sum);
  sum=0;
  free(v1);
  v2=(int *)malloc(sizeof(int)*LEN);
  for(int i=0;i<LEN;i++) {
    sum+=*(v2+i);
  }

  printf("Somme des éléments de v2 : %d\n", sum);
  free(v2);

  return(EXIT_SUCCESS);
}
```

14. Il existe différentes alternatives à l'utilisation de malloc(3) pour l'allocation de mémoire comme Hoard ou gperftools

L'exécution du programme ci-dessus affiche le résultat suivant sur la sortie standard. Ceci illustre bien que la fonction malloc(3) n'initialise pas les zones de mémoire qu'elle alloue.

```
Somme des éléments de v1 : 0
Somme des éléments de v2 : 1282048
```

Lors de l'exécution du programme, on remarque que la première zone mémoire retournée par malloc(3) a été initialisée à zéro. C'est souvent le cas en pratique pour des raisons de sécurité, mais ce serait une erreur de faire cette hypothèse dans un programme. Si la zone de mémoire doit être initialisée, la mémoire doit être allouée par calloc(3) ou via une initialisation explicite ou en utilisant des fonctions telles que bzero(3) ou memset(3).

2.5.5 Les arguments et variables d'environnement

Lorsque le système d'exploitation charge un programme Unix en mémoire, il initialise dans le haut de la mémoire une zone qui contient deux types de variables. Cette zone contient tout d'abord les arguments qui ont été passés via la ligne de commande. Le système d'exploitation met dans `argc` le nombre d'arguments et place dans `char *argv[]` tous les arguments passés avec dans `argv[0]` le nom du programme qui est exécuté.

Cette zone contient également les variables d'environnement. Ces variables sont généralement relatives à la configuration du système. Leurs valeurs sont définies par l'administrateur système ou l'utilisateur. De nombreuses variables d'environnement sont utilisées dans les systèmes Unix. Elles servent à modifier le comportement de certains programmes. Une liste exhaustive de toutes les variables d'environnement est impossible, mais en voici quelques unes qui sont utiles en pratique [15] :

- `HOSTNAME` : le nom de la machine sur laquelle le programme s'exécute. Ce nom est fixé par l'administrateur système via la commande hostname(1)
- `SHELL` : l'interpréteur de commande utilisé par défaut pour l'utilisateur courant. Cet interpréteur est lancé par le système au démarrage d'une session de l'utilisateur. Il est stocké dans le fichier des mots de passe et peut être modifié par l'utilisateur via la commande passwd(1)
- `USER` : le nom de l'utilisateur courant. Sous Unix, chaque utilisateur est identifié par un numéro d'utilisateur et un nom uniques. Ces identifiants sont fixés par l'administrateur système via la commande passwd(1)
- `HOME` : le répertoire d'accueil de l'utilisateur courant. Ce répertoire d'accueil appartient à l'utilisateur. C'est dans ce répertoire qu'il peut stocker tous ses fichiers.
- `PRINTER` : le nom de l'imprimante par défaut qui est utilisée par la commande lp(1posix)
- `PATH` : cette variable d'environnement contient la liste ordonnée des répertoires que le système parcourt pour trouver un programme à exécuter. Cette liste contient généralement les répertoires dans lesquels le système stocke les exécutables standards, comme `/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:` ainsi que des répertoires relatifs à des programmes spécialisés comme `/usr/lib/mozart/bin:/opt/python3/bin`. L'utilisateur peut ajouter des répertoires à son `PATH` avec bash(1) en incluant par exemple la commande `PATH=$PATH:$HOME/local/bin:.` dans son fichier `.profile`. Cette commande ajoute au `PATH` par défaut le répertoire `$HOME/local/bin` et le répertoire courant. Par convention, Unix utilise le caractère `.` pour représenter ce répertoire courant.

La librairie standard contient plusieurs fonctions qui permettent de manipuler les variables d'environnement d'un processus. La fonction getenv(3) permet de récupérer la valeur associée à une variable d'environnement. La fonction unsetenv(3) permet de supprimer une variable de l'environnement du programme courant. La fonction setenv(3) permet elle de modifier la valeur d'une variable d'environnement. Cette fonction alloue de la mémoire pour stocker la nouvelle variable d'environnement et peut échouer si il n'y a pas assez de mémoire disponible pour stocker de nouvelles variables d'environnement. Ces fonctions sont utilisées notamment par l'interpréteur de commande mais parfois par des programmes dont le comportement dépend de la valeur de certaines variables d'environnement. Par exemple, la commande man(1) utilise différentes variables d'environnement pour déterminer par exemple où les pages de manuel sont stockées et la langue (variable `LANG`) dans laquelle il faut afficher les pages de manuel.

15. Il possible de lister les définitions actuelles des variables d'environnement via la commande printenv(1). Les interpréteurs de commande tels que bash(1) permettent de facilement modifier les valeurs de ces variables. La plupart d'entre elles sont initialisées par le système ou via les fichiers qui sont chargés automatiquement au démarrage de l'interpréteur comme `/etc/profile` qui contient les variables fixées par l'administrateur système ou le fichier `.profile` du répertoire d'accueil de l'utilisateur qui contient les variables d'environnement propres à cet utilisateur.

Le programme ci-dessous illustre brièvement l'utilisation de getenv(3), unsetenv(3) et setenv(3). Outre ces fonctions, il existe également clearenv(3) qui permet d'effacer complètement toutes les variables d'environnement du programme courant et putenv(3) qui était utilisé avant setenv(3).

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

// affiche la valeur de la variable d'environnement var
void print_var(char *var) {
  char *val=getenv(var);
  if(val!=NULL)
    printf("La variable %s a la valeur : %s\n",var,val);
  else
    printf("La variable %s n'a pas été assignée\n",var);
}

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

  char *old_path=getenv("PATH");

  print_var("PATH");

  if(unsetenv("PATH")!=0) {
    fprintf(stderr,"Erreur unsetenv\n");
    exit(EXIT_FAILURE);
  }

  print_var("PATH");

  if(setenv("PATH",old_path,1)!=0) {
    fprintf(stderr,"Erreur setenv\n");
    exit(EXIT_FAILURE);
  }

  print_var("PATH");

  return(EXIT_SUCCESS);
}
```

2.5.6 La pile (ou stack)

La *pile* ou *stack* en anglais est la dernière zone de mémoire utilisée par un processus. C'est une zone très importante car c'est dans cette zone que le processus va stocker l'ensemble des variables locales mais également les valeurs de retour de toutes les fonctions qui sont appelées. Cette zone est gérée comme une pile, d'où son nom. Pour comprendre son fonctionnement, nous utiliserons le programme `/C/S3-src/fact.c` qui permet de calculer une factorielle de façon récursive.

```
// retourne i*j
int times(int i, int j) {
  int m;
  m=i*j;
  printf("\t[times(%d,%d)] : return(%d)\n",i,j,m);
  return m;
}
// calcul récursif de factorielle
// n>0
int fact(int n) {
  printf("[fact(%d)]: Valeur de n:%d, adresse: %p\n",n,n,&n);
  int f;
  if(n==1) {
    printf("[fact(%d)]: return(1)\n",n);
```

```
    return(n);
  }
  printf("[fact(%d)]: appel à fact(%d)\n",n,n-1);
  f=fact(n-1);
  printf("[fact(%d)]: calcul de times(%d,%d)\n",n,n,f);
  f=times(n,f);
  printf("[fact(%d)]: return(%d)\n",n,f);
  return(f);
}

void compute() {
  int nombre=3;
  int f;
  printf("La fonction fact est à l'adresse : %p\n",fact);
  printf("La fonction times est à l'adresse : %p\n",times);
  printf("La variable nombre vaut %d et est à l'adresse %p\n",nombre,&nombre);
  f=fact(nombre);
  printf("La factorielle de %d vaut %d\n",nombre,f);
}
```

Lors de l'exécution de la fonction `compute()`, le programme ci-dessus produit la sortie suivante.

```
La fonction fact est à l'adresse : 0x100000a0f
La fonction times est à l'adresse : 0x1000009d8
La variable nombre vaut 3 et est à l'adresse 0x7fff5fbfe1ac
[fact(3)]: Valeur de n:3, adresse: 0x7fff5fbfe17c
[fact(3)]: appel à fact(2)
[fact(2)]: Valeur de n:2, adresse: 0x7fff5fbfe14c
[fact(2)]: appel à fact(1)
[fact(1)]: Valeur de n:1, adresse: 0x7fff5fbfe11c
[fact(1)]: return(1)
[fact(2)]: calcul de times(2,1)
        [times(2,1)] : return(2)
[fact(2)]: return(2)
[fact(3)]: calcul de times(3,2)
        [times(3,2)] : return(6)
[fact(3)]: return(6)
La factorielle de 3 vaut 6
```

Il est intéressant d'analyser en détails ce calcul récursif de la factorielle car il illustre bien le fonctionnement du stack et son utilisation.

Tout d'abord, il faut noter que les fonctions `fact` et `times` se trouvent, comme toutes les fonctions définies dans le programme à l'intérieur du *segment text*. La variable `nombre` quant à elle se trouve sur la pile en haut de la mémoire. Il s'agit d'une variable locale qui est allouée lors de l'exécution de la fonction `compute`. Il en va de même des arguments qui sont passés aux fonctions. Ceux-ci sont également stockés sur la pile. C'est le cas par exemple de l'argument `n` de la fonction `fact`. Lors de l'exécution de l'appel à `fact(3)`, la valeur `3` est stockée sur la pile pour permettre à la fonction `fact` d'y accéder. Ces accès sont relatifs au sommet de la pile comme nous aurons l'occasion de le voir dans la présentation du langage d'assemblage. Le premier appel récursif se fait en calculant la valeur de l'argument (`2`) et en appelant la fonction. L'argument est placé sur la pile, mais à une autre adresse que celle utilisée pour `fact(3)`. Durant son exécution, la fonction `fact(2)` accède à ses variables locales sur la pile sans interférer avec les variables locales de l'exécution de `fact(3)` qui attend le résultat de `fact(2)`. Lorsque `fact(2)` fait l'appel récursif, la valeur de son argument (`1`) est placée sur la pile et l'exécution de `fact(1)` démarre. Celle-ci a comme environnement d'exécution le sommet de la pile qui contient la valeur `1` comme argument et la fonction retourne la valeur `1` à l'exécution de `fact(2)` qui l'avait lancée. Dès la fin de `fact(1)`, `fact(2)` reprend son exécution où elle avait été interrompue et applique la fonction `times` avec `2` et `1` comme arguments. Ces deux arguments sont placés sur le pile et `times` peut y accéder au début de son exécution pour calculer la valeur `2` et retourner le résultat à la fonction qui l'a appelé, c'est-à-dire `fact(2)`. Cette dernière retrouve son environnement d'exécution sur la pile. Elle peut maintenant retourner son résultat à la fonction `fact(3)` qui l'avait appelée. Celle-ci va appeler la fonction `times` avec `3` et `2` comme arguments et finira par retourner la valeur `6`.

La pile joue un rôle essentiel lors de l'exécution de programmes en C puisque toutes les variables locales, y compris celles de la fonction `main` y sont stockées. Comme nous le verrons lorsque nous aborderons le langage assembleur, la pile sert aussi à stocker l'adresse de retour des fonctions. C'est ce qui permet à l'exécution de `fact(2)` de se poursuivre correctement après avoir récupéré la valeur calculée par l'appel à `fact(1)`. L'utilisation de la pile pour stocker les variables locales et les arguments de fonctions a une conséquence importante. Lorsqu'une variable est définie comme argument ou localement à une fonction `f`, elle n'est accessible que durant l'exécution de la fonction `f`. Avant l'exécution de `f` cette variable n'existe pas et si `f` appelle la fonction `g`, la variable définie dans `f` n'est plus accessible à partir de la fonction `g`.

En outre, comme le langage C utilise le passage par valeur, les valeurs des arguments d'une fonction sont copiés sur la pile avant de démarrer l'exécution de cette fonction. Lorsque la fonction prend comme argument un entier, cette copie prend un temps très faible. Par contre, lorsque la fonction prend comme argument une ou plusieurs structures de grand taille, celles-ci doivent être entièrement copiées sur la pile. A titre d'exemple, le programme ci-dessous définit une très grande structure contenant un entier et une zone permettant de stocker un million de caractères. Lors de l'appel à la fonction `sum`, les structures `one` et `two` sont entièrement copiées sur la pile. Comme chaque structure occupe plus d'un million d'octets, cela prend plusieurs centaines de microsecondes. Cette copie est nécessaire pour respecter le passage par valeur des structures à la fonction `sum`. Celle-ci ne peut pas modifier le contenu des structures qui lui sont passées en argument. Par comparaison, lors de l'appel à `sumptr`, seules les adresses de ces deux structures sont copiées sur la pile. Un appel à `sumptr` prend moins d'une microseconde, mais bien entendu la fonction `sumptr` a accès via les pointeurs passés en argument à toute la zone de mémoire qui leur est associée.

```
#define MILLION 1000000

struct large_t {
  int i;
  char str[MILLION];
};

int sum(struct large_t s1, struct large_t s2) {
  return (s1.i+s2.i);
}

int sumptr(struct large_t *s1, struct large_t *s2) {
  return (s1->i+s2->i);
}

int main(int argc, char *argv[]) {
  struct timeval tvStart, tvEnd;
  int err;
  int n;
  struct large_t one={1,"one"};
  struct large_t two={1,"two"};

  n=sum(one,two);
  n=sumptr(&one,&two);
```

Certaines variantes de Unix et certains compilateurs permettent l'allocation de mémoire sur la pile via la fonction alloca(3). Contrairement à la mémoire allouée par malloc(3) qui doit être explicitement libérée en utilisant free(3), la mémoire allouée par alloca(3) est libérée automatiquement à la fin de l'exécution de la fonction dans laquelle la mémoire a été allouée. Cette façon d'allouer de la mémoire sur la pile n'est pas portable et il est préférable de n'allouer de la mémoire que sur le tas en utilisant malloc(3).

Les versions récentes du C et notamment [C99] permettent d'allouer de façon dynamique un tableau sur la pile. Cette fonctionnalité peut être utile dans certains cas, mais elle peut aussi être la source de nombreuses erreurs et difficultés. Pour bien comprendre ce problème, considérons à nouveau la fonction `duplicate` qui a été définie précédemment en utilisant malloc(3) et des pointeurs.

```
#include <string.h>

char *duplicate(char * str) {
  int i;
```

```
  size_t len=strlen(str);
  char *ptr=(char *)malloc(sizeof(char)*(len+1));
  if(ptr!=NULL) {
    for(i=0;i<len+1;i++) {
        *(ptr+i)=*(str+i);
      }
  }
  return ptr;
}
```

Un étudiant pourrait vouloir éviter d'utiliser malloc(3) et écrire plutôt la fonction suivante.

```
char *duplicate2(char * str) {
  int i;
  size_t len=strlen(str);
  char str2[len+1];
  for(i=0;i<len+1;i++) {
    str2[i]=*(str+i);
  }
  return str2;
}
```

Lors de la compilation, gcc(1) affiche le *warning* `In function 'duplicate2': warning: function returns address of local variable`. Ce warning indique que la ligne `return str2;` retourne l'adresse d'une variable locale qui n'est plus accessible à la fin de la fonction `duplicate2`. En effet, l'utilisation de tableaux alloués dynamiquement sur la pile est équivalent à une utilisation implicite de alloca(3). La déclaration `char str2[len];` est équivalente à `char *str2 =(char *)alloca(len*sizeof(char));` et la zone mémoire allouée sur la pile pour `str2` est libérée lors de l'exécution de `return str2;` puisque toute mémoire allouée sur la pile est implicitement libérée à la fin de l'exécution de la fonction durant laquelle elle a été allouée. Donc, une fonction qui appelle `duplicate2` ne peut pas récupérer les données se trouvant dans la zone mémoire qui a été allouée par `duplicate2`.

2.6 Compléments de C

Dans les sections précédentes, nous n'avons pas pu couvrir l'ensemble des concepts avancés qui sont relatifs à une bonne utilisation du langage C. Cette section contient quelques notions plus avancées qui sont importantes en pratique.

2.6.1 Pointeurs

Les pointeurs sont très largement utilisés dans les programmes écrits en langage C. Nous avons utilisé des pointeurs vers des types de données primitifs tel que les `int`, `char` ou `float` et des pointeurs vers des structures. En pratique, il est possible en C de définir des pointeurs vers n'importe quel type d'information qui est manipulée par un programme C.

Un premier exemple sont les pointeurs vers des fonctions. Comme nous l'avons vu dans le chapitre précédent, une fonction est une séquence d'instructions assembleur qui sont stockées à un endroit bien précis de la mémoire. Cette localisation précise des instructions qui implémentent la fonction permet d'appeler une fonction avec l'instruction `calll`. En C, il est parfois aussi souhaitable de pouvoir appeler une fonction via un pointeur vers cette fonction plutôt qu'en nommant la fonction directement. Cela peut rendre le code plus flexible et plus facile à adapter. Nous avons déjà utilisé des pointeurs vers des fonctions sans le savoir lorsque nous avons utilisé `printf("fct : %p\n",f)` où `f` est un nom de fonction. L'exemple ci-dessous montre une autre utilisation intéressante des pointeurs vers des fonctions. Lorsque l'on écrit du code C, il est parfois utile d'ajouter des commandes qui permettent d'afficher à l'écran des informations de debugging. L'exemple ci-dessous est une application qui supporte trois niveaux de debugging. Rien n'est affiché au niveau `0`, une ligne s'affiche au niveau `1` et des informations plus détaillées sont affichées au niveau `2`. Lors de son exécution , l'application affiche la sortie suivante.

```
$./fctptr 0
fct debug_print : 0x100000d28
$ ./fctptr 1
fct debug_print : 0x100000d32
debug: Hello
$ ./fctptr 2
fct debug_print : 0x100000d5f
debug: Hello
g=1
```

Cette application qui supporte plusieurs niveaux de debugging utilise pourtant toujours le même appel pour afficher l'information de debugging : `(debug_print[debug_level])(...);`. Cet appel profite des pointeurs vers les fonctions. Le tableau `debug_print` est un tableau de pointeurs vers des fonctions qui chacune prend comme argument un `char *`. La variable globale `debug_level` est initialisée sur base de l'argument passé au programme.

```
int g=1;
int debug_level;

void empty (char *str) {
  return;
}

void oneline(char *str) {
  fprintf(stderr,"debug: %s\n",str);
}

void detailed(char *str) {
  fprintf(stderr, "debug: %s\n",str);
  fprintf(stderr,"g=%d\n",g);
}

void (* debug_print[])(char *) = { empty,
                                   oneline,
                                   detailed };

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

  if(argc!=2)
    return(EXIT_FAILURE);

  debug_level=atoi(argv[1]);
  if((debug_level<0) || (debug_level>2) )
    return(EXIT_FAILURE);
  printf("fct debug_print : %p\n",debug_print[debug_level]);
  (debug_print[debug_level])("Hello");

   return(EXIT_SUCCESS);
}
```

Ce n'est pas la seule utilisation des pointeurs vers des fonctions. Il y a notamment la fonction de la librairie qsort(3) qui permet de trier un tableau contenant n'importe quel type d'information. Cette fonction prend plusieurs arguments :

```
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *));
```

Le premier est un pointeur vers le début de la zone mémoire à trier. Le second est le nombre d'éléments à trier. Le troisième contient la taille des éléments stockés dans le tableau. Le quatrième argument est un pointeur vers la fonction qui permet de comparer deux éléments du tableau. Cette fonction retourne un entier négatif si son premier argument est inférieur au second et positif ou nul sinon. Un exemple de fonction de comparaison est la fonction

strcmp(3) de la librairie standard. Un autre exemple est repris ci-dessous avec une fonction de comparaison simple qui permet d'utiliser qsort(3) pour trier un tableau de `double`.

```
#define SIZE 5
double array[SIZE]= { 1.0, 7.32, -3.43, 8.7, 9.99 };

void print_array() {
  for(int i=0;i<SIZE;i++)
    printf("array[i]:%f\n",array[i]);
}

int cmp(const void *ptr1, const void *ptr2) {
  const double *a=ptr1;
  const double *b=ptr2;
  if(*a==*b)
    return 0;
  else
    if(*a<*b)
      return -1;
    else
      return +1;
}

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

  printf("Avant qsort\n\n");
  print_array();
  qsort(array,SIZE,sizeof(double),cmp);
  printf("Après qsort\n\n");
  print_array();

  return(EXIT_SUCCESS);
}
```

Il est utile d'analyser en détails les arguments de la fonction de comparaison utilisée par qsort(3). Celle-ci prend deux arguments de type `const void *`. L'utilisation de pointeurs `void *` est nécessaire car la fonction doit être générique et pouvoir traiter n'importe quel type de pointeurs. `void *` est un pointeur vers une zone quelconque de mémoire qui peut être casté vers n'importe quel type de pointeur par la fonction de comparaison. Le qualificatif `const` indique que la fonction n'a pas le droit de modifier la donnée référencée par ce pointeur, même si elle reçoit un pointeur vers cette donnée. On retrouvera régulièrement cette utilisation de `const` dans les signatures des fonctions de la librairie pour spécifier des contraintes sur les arguments passés à une fonction [16].

Le second type de pointeurs que nous n'avons pas encore abordé en détails sont les pointeurs vers des pointeurs. En fait, nous les avons utilisés sans vraiment le savoir dans la fonction `main`. En effet, le second argument de cette fonction est un tableau de pointeurs qui pointent chacun vers des chaînes de caractères différentes. La notation `char *argv[]` est équivalente à la notation `char **argv`. `**argv` est donc un pointeur vers une zone qui contient des pointeurs vers des chaînes de caractères. Ce pointeur vers un pointeur doit être utilisé avec précaution. `argv[0]` est un pointeur vers une chaîne de caractères. La construction `&(argv[0])` permet donc d'obtenir un pointeur vers un pointeur vers une chaîne de caractères, ce qui correspond bien à la déclaration `char **`. Ensuite, l'utilisation de `*p` pourrait surprendre. `*p` est un pointeur vers une chaîne de caractères. Il peut donc être comparé à `NULL` qui est aussi un pointeur, incrémenté et la chaîne de caractères qu'il référencie peut être affichée par printf(3).

```
int main(int argc, char **argv) {

  char **p;
  p=argv;
  printf("Arguments :");
  while(*p!=NULL) {
    printf(" %s",*p);
```

16. La qualificateur `restrict` est également parfois utilisé pour indiquer des contraintes sur les pointeurs passés en argument à une fonction [Walls2006].

```
    p++;
  }
  printf("\n");
  return(EXIT_SUCCESS);
}
```

En pratique, ces pointeurs vers des pointeurs se retrouveront lorsque l'on doit manipuler des structures multidimensionnelles, mais aussi lorsqu'il faut qu'une fonction puisse modifier une adresse qu'elle a reçue en argument.

Un autre exemple d'utilisation de pointeurs vers des pointeurs est la fonction strtol(3) de la librairie standard. Cette fonction est une généralisation des fonctions comme atoi(3). Elle permet de convertir une chaîne de caractères en un nombre. La fonction strtol(3) prend trois arguments et retourne un `long`. Le premier argument est un pointeur vers la chaîne de caractères à convertir. Le troisième argument est la base utilisée pour cette conversion.

```
#include <stdlib.h>
long
strtol(const char *restrict str, char **restrict endptr, int base);
```

L'utilisation principale de strtol(3) est de convertir une chaîne de caractères en un nombre. La fonction atoi(3) fait de même et l'expression `atoi("1252")` retourne l'entier `1252`. Malheureusement, la fonction atoi(3) ne traite pas correctement les chaînes de caractères qui ne contiennent pas un nombre. Elle ne retourne pas de code d'erreur et ne permet pas savoir quelle partie de la chaîne de caractères passée en argument était en erreur.

strtol(3) est un exemple de fonction qui doit retourner deux types d'informations. Tout d'abord, strtol(3) retourne un résultat (dans ce cas un nombre). Si la chaîne de caractères à convertir est erronée, strtol(3) convertit le début de la chaîne et retourne un pointeur indiquant le premier caractère en erreur. Pour bien comprendre le fonctionnement de strtol(3), considérons l'exemple ci-dessous.

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

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

  char *p, *s;
  long li;
  s = "1252";
  li = strtol(s,&p,10);
  if(*p != '\0') {
    printf("Caractère erronné : %c\n",*p);
      // p pointe vers le caractère en erreur
  }
  printf("Valeur convertie : %s -> %ld\n",s,li);

  s = "12m52";
  li = strtol(s,&p,10);
  if(*p != '\0') {
    printf("Caractère erronné : %c\n",*p);
  }
  printf("Valeur convertie : %s -> %ld\n",s,li);

  return(EXIT_SUCCESS);
}
```

Lors de son exécution, ce programme affiche la sortie suivante.

```
Valeur convertie : 1252 -> 1252
Caractère erronné : m
Valeur convertie : 12m52 -> 12
```

L'appel à strtol(3) prend trois arguments. Tout d'abord un pointeur vers la chaîne de caractères à convertir. Ensuite l'adresse d'un pointeur vers une chaîne de caractères. Enfin la base de conversion. La première chaîne de caractères est correcte. Elle est convertie directement. La seconde par contre contient un caractère erroné. Lors de son exécution, strtol(3) va détecter la présence du caractère `m` et placera un pointeur vers ce caractère dans `*p`.

Pour que la fonction strtol(3) puisse retourner un pointeur de cette façon, il est nécessaire que son second argument soit de type `char **`. Si le second argument était de type `char *`, la fonction strtol(3) recevrait l'adresse d'une zone mémoire contenant un caractère. Comme le langage C utilise la passage par valeur, strtol(3) pourrait modifier la caractère pointé par ce pointeur mais pas son adresse. En utilisant un second argument de type `char **`, strtol(3) a la possibilité de modifier la valeur pointée par ce pointeur.

Une implémentation partielle de strtol(3) pourrait être la suivante.

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <stdbool.h>

int mystrtol(const char *restrict str,
             char **restrict endptr,
             int base) {

  int val;
  int i=0;
  int err=false;
  while(!err && *(str+i)!='\0')
    {
      if(!isdigit(*(str+i))) {
        err=true;
        *endptr=(char *)(str+i);
      }
      i++;
    }
  // ...
  return val;
}
```

Cette partie de code utilise la fonction isdigit(3) pour vérifier si les caractères présents dans la chaîne de caractères sont des chiffres. Sinon, elle fixe via son second argument la valeur du pointeur vers le caractère en erreur. Cela est réalisé par l'expression `*endptr=(char *)(str+i);`. Il faut noter que `*endptr` est bien une zone mémoire pointée par le pointeur `endptr` reçu comme second argument. Cette valeur peut donc être modifiée.

Il existe d'autres fonctions de la librairie standard qui utilisent des pointeurs vers des pointeurs comme arguments dont notamment strsep(3) et strtok_r(3).

2.6.2 De grands programmes en C

Lorsque l'on développe de grands programmes en C, il est préférable de découper le programme en modules. Chaque module contient des fonctions qui traitent d'un même type de problème et sont fortement couplées. A titre d'exemple, un module `stack` pourrait regrouper différentes fonctions de manipulation d'une pile. Un autre module pourrait regrouper les fonctions relatives au dialogue avec l'utilisateur, un autre les fonctions de gestion des fichiers, ...

Pour comprendre l'utilisation de ces modules, considérons d'abord un programme trivial composé de deux modules. Le premier module est celui qui contient la fonction `main`. Tout programme C doit contenir une fonction `main` pour pouvoir être exécuté. C'est en général l'interface avec l'utilisateur. Le second module contient une fonction générique qui est utilisée par le module principal.

```
/*************************************
 * main.c
 *
 * Programme d'exemple pour le linker
 *
 *************************************/

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

```c
#include <stdlib.h>

int main(int argc, char *argv[]) {
  float f1=3.45;
  float f2=-4.12;
  printf("Minimum(%f,%f)=%f\n",f1,f2,min(f1,f2));
  return(EXIT_SUCCESS);
}
```

Un module d'un programme C est en général décomposé en deux parties. Tout d'abord, le fichier *fichier header* contient les définitions de certaines constantes et les signatures des fonctions exportées par ce module. Ce fichier est en quelque sorte un résumé du module, ou plus précisément de son interface externe. Il doit être inclus dans tout fichier qui utilise les fonctions du module correspondant. Dans un tel fichier *fichier header*, on retrouve généralement trois types d'informations :

- les signatures des fonctions qui sont définies dans le module. En général, seules les fonctions qui sont destinées à être utilisées par des modules extérieures sont reprises dans le *fichier header*
- les constantes qui sont utilisées à l'intérieur du module et doivent être visibles en dehors de celui-ci, notamment par les modules qui utilisent les fonctions du module. Ces constantes peuvent être définies en utilisant des directives `#define` du préprocesseur
- les variables globales qui sont utilisées par les fonctions du module et doivent être accessibles en dehors de celui-ci

```c
/**************************************
 * min.h
 *
 **************************************/

#ifndef _MIN_H_
#define _MIN_H_

float min(float, float);

#endif /* _MIN_H */
```

Note : Un *fichier header* ne doit être inclus qu'une seule fois

L'exemple de *fichier header* ci-dessus illustre une convention courante dans l'écriture de ces fichiers. Parfois, il est nécessaire d'inclure un *fichier header* dans un autre fichier header. Suite à cela, il est possible que les mêmes définitions d'un *fichier header* soient incluses deux fois ou plus dans le même module. Cela peut causer des erreurs de compilation qui risquent de perturber certains programmeurs. Une règle de bonne pratique pour éviter ce problème est d'inclure le contenu du *fichier header* de façon conditionnelle comme présenté ci-dessus. Une constante, dans ce cas `_MIN_H_`, est définie pour le *fichier header* concerné. Cette constante est définie dans la première ligne effective du *fichier header*. Celui-ci n'est inclus dans un module que si cette constante n'a pas été préalablement définie. Si cette constante est connue par le préprocesseur, cela indique qu'un autre *fichier header* a déjà inclus les définitions de ce fichier et qu'elles ne doivent pas être incluses une seconde fois.

```c
/**************************************
 * min.c
 *
 * Programme d'exemple pour le linker
 *
 **************************************/

#include "min.h"

float min(float a, float b) {
  if(a<b)
    return a;
  else
    return b;
}
```

Note : Localisation des fichiers header

Un programmeur C peut utiliser deux types de fichiers header. Il y a tout d'abord les fichiers headers standards qui sont fournis avec le système. Ce sont ceux que nous avons utilisés jusque maintenant. Ces headers standards se reconnaissent car ils sont entourés des caractères < et > dans la directive `#include`. Ceux-ci se trouvent dans des répertoires connus par le compilateur, normalement `/usr/include`. Les fichiers headers qui accompagnent un module se trouvent eux généralement dans le même répertoire que le module. Dans l'exemple ci-dessus, le header `min.h` est inclus via la directive `#include "min.h"`. Lorsque le préprocesseur rencontre une telle directive, il cherche le fichier dans le répertoire courant. Il est possible de spécifier des répertoires qui contiennent des fichiers headers via l'argument `-I` de gcc(1) ou en utilisant les variables d'environnement `GCC_INCLUDE_DIR` ou `CPATH`.

Lorsque l'on doit compiler un programme qui fait appel à plusieurs modules, quelle que soit sa taille, il est préférable d'utiliser make(1) pour automatiser sa compilation. Le fichier ci-dessous est un exemple minimaliste de *Makefile* utilisable pour un tel projet.

```
myprog: main.o min.o
        gcc -std=c99 -o myprog main.o min.o

main.o: main.c min.h
        gcc -std=c99 -c main.c

min.o: min.c min.h
        gcc -std=c99 -c min.c
```

La compilation d'un tel programme se déroule en plusieurs étapes. La première étape est celle du préprocesseur. Celui-ci est directement appelé par le compilateur gcc(1) mais il est également possible de l'invoquer directement via cpp(1). Le préprocesseur remplace toutes les macros telles que les `#define` et insère les fichiers headers sur base des directives `#include`. La sortie du préprocesseur est utilisée directement par le compilateur. Celui-ci transforme le module en langage C en langage assembleur. Ce module en assembleur est ensuite assemblé par as(1) pour produire un *fichier objet*. Ce *fichier objet* n'est pas directement exécutable. Il contient les instructions en langage machine pour les différentes fonctions définies dans le module, les définitions des constantes et variables globales ainsi qu'une table reprenant tous les symboles (noms de fonction, noms de variables globales, ...) définis dans ce module. Ces phases sont exécutées pour chaque module utilisé. Par convention, les fichiers objets ont en général l'extension `.o`. Ces fichiers objet sont créés par les deux dernières cibles du fichier `Makefile` ci-dessus. L'option `-c` passée à gcc(1) indique à gcc(1) qu'il doit générer un fichier objet sans le transformer en exécutable. Cette dernière opération est réalisée par la première cible du `Makefile`. Dans cette cible, gcc(1) fait office d'éditeur de liens ou de *linker* en anglais. Le *linker* combine différents fichiers objets en faisant les liens nécessaires entre les fichiers. Dans notre exemple, le fichier `main.o` contient une référence vers la fonction `min` qui n'est pas connue lors de la compilation de `main.c`. Par contre, cette référence est connue dans le fichier `min.o`. L'éditeur de liens va combiner ces références de façon à permettre aux fonctions d'un module d'exécuter n'importe quelle fonction définie dans un autre module.

La figure ci-dessous représente graphiquement les différentes étapes de compilation des modules `min.c` et `main.c`.

Lorsque plusieurs modules, potentiellement développés par des programmeurs qui ne se sont pas concertés, doivent être intégrés dans un grand programme, il y a des risques de conflits entre des variables et fonctions qui pourraient être définies dans plusieurs modules différents. Ainsi, deux modules pourraient définir la fonction `int min(int, int)` ou la variable globale `float dist`. Le langage C intègre des facilités qui permettent d'éviter ou de contrôler ces problèmes.

Tout d'abord, les variables locales sont locales au bloc dans lequel elles sont définies. Ce principe permet d'utiliser le même nom de variable dans plusieurs blocs d'un même fichier. Il s'étend naturellement à l'utilisation de variables locales dans des fichiers différents.

Pour les variables globales, la situation est différente. Si une variable est définie en dehors d'un bloc dans un fichier, cette variable est considérée comme étant globale. Par défaut, elle est donc accessible depuis tous les modules qui composent le programme. Cela peut en pratique poser des difficultés si le même nom de variable est utilisé dans deux modules différents. Pour contourner ce problème, le langage C utilise le qualificateur `static`.

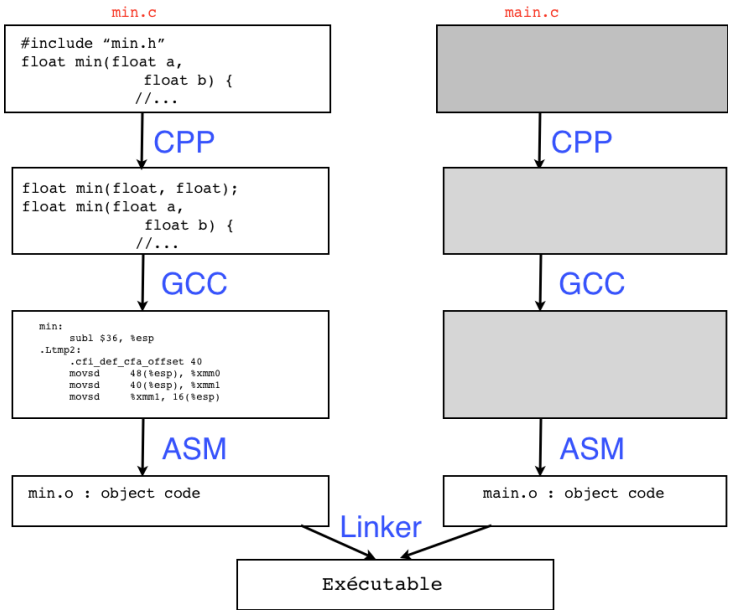


FIGURE 2.3 – Etapes de compilation

Lorsque ce qualificateur est placé devant une déclaration de variable en dehors d'un bloc dans un module, il indique que la variable doit être accessible à toutes les fonctions du module mais pas en dehors du module. Lorsqu'un module utilise des variables qui sont communes à plusieurs fonctions mais ne doivent pas être visibles en dehors du module, il est important de les déclarer comme étant `static`. Le deuxième qualificateur relatif aux variables globales est `extern`. Lorsqu'une déclaration de variable globale est préfixée par `extern`, cela indique au compilateur que la variable est définie dans un autre module qui sera linké ultérieurement. Le compilateur réserve une place pour cette variable dans la table des symboles du fichier objet, mais cette place ne pourra être liée à la zone mémoire qui correspond à cette variable que lorsque l'éditeur de liens combinera les différents fichiers objet entre eux.

Note : Les deux utilisations de `static` pour des variables

La qualificateur `static` peut être utilisé à la fois pour des variables qui sont définies en dehors d'un bloc et dans un bloc. Lorsqu'une variable est définie comme étant `static` hors d'un bloc dans un module, elle n'est accessible qu'aux fonctions de ce module. Par contre, lorsqu'une variable est définie comme étant `static` à l'intérieur d'un bloc, par exemple dans une fonction, le qualificatif indique que cette variable doit toujours se trouver à la même localisation en mémoire, quel que soit le moment où elle est appelée. Ces variables `static` sont placées par le compilateur dans le bas de la mémoire, avec les variables globales. Contrairement aux variables locales traditionnelles, une variable locale `static` garde sa valeur d'une invocation de la fonction à l'autre. En pratique, les variables locales `static` doivent être utilisées avec précaution et bien documentées. Un de leurs intérêt est qu'elles ne sont initialisées qu'au lancement du programme et pas à chaque invocation de la fonction où elles sont définies.

La qualificateur `static` peut aussi précéder des déclarations de fonctions. Dans ce cas, il indique que la fonction ne doit pas être visible en dehors du module dans lequel elle est définie. Sans le qualificateur `static`, une fonction déclarée dans un module est accessible depuis n'importe quel autre module.

Afin d'illustrer l'utilisation de `static` et `extern`, considérons le programme `prog.c` ci-dessous qui inclut le module `module.c` et également le module `min.c` présenté plus haut.

```
/*************************************
 * module.h
 *
 *************************************/
#ifndef _MODULE_H_
#define _MODULE_H_
```

```
float vmin(int, float *);

#endif /* _MODULE_H */

/*************************************
 * module.c
 *
 *************************************/

#include "module.h"

static float min(float, float);

int num1=0;  // accessible hors de module.c
extern int num2; // définie dans un autre module
static int num3=1252; // accessible uniquement dans ce module

float vmin(int n, float *ptr) {
  float *p=ptr;
  float m=*ptr;
  for(int i=1;i<n;i++) {
    m=min(m,*p);
    p++;
  }
  return m;
}

static float min(float a, float b) {
  if(a<b)
    return a;
  else
    return b;
}
```

Ce module contient deux fonctions, `vmin` et `min`. `vmin` est déclarée sans qualificatif. Elle est donc accessible depuis n'importe quel module. Sa signature est reprise dans le *fichier header* `module.h`. La fonction `min` par contre est déclarée avec le qualificatif `static`. Cela implique qu'elle n'est utilisable qu'à l'intérieur de ce module et invisible de tout autre module. La variable globale `num1` est accessible depuis n'importe quel module. La variable `num2` également, mais elle est initialisée dans un autre module. Enfin, la variable `num3` n'est accessible qu'à l'intérieur de ce module.

```
#include "min.h"
#include "module.h"

#define SIZE 4

extern int num1; // définie dans un autre module
int num2=1252;   // accessible depuis un autre module
static int num3=-1; // accessible uniquement dans ce module

void f() {
  static int n=0;
  int loc=2;
  if(n==0)
    printf("n est à  l'adresse %p et loc à l'adresse %p\n",&n,&loc);
  printf("f, n=%d\n",n);
  n++;
}

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

  float v[SIZE]={1.0, 3.4, -2.4, 9.9};
```

```
  printf("Minimum: %f\n",vmin(SIZE,v));
  f();
  f();
  printf("Minimum(0.0,1.1)=%f\n",min(0.0,1.1));
  return(EXIT_SUCCESS);
}
```

Ce module inclus les fichiers `min.h` et `module.h` qui contiennent les signatures des fonctions se trouvant dans ces deux modules. Trois variables globales sont utilisées par ce module. `num1` est définie dans un autre module (dans ce cas `module.c`). `num2` est initialisée dans ce module mais accessible depuis n'importe quel autre module. `num3` est une variable globale qui est accessible uniquement depuis le module `prog.c`. Même si cette variable porte le même nom qu'une autre variable déclarée dans `module.c`, il n'y aura pas de conflit puisque ces deux variables sont `static`.

La fonction `f` mérite que l'on s'y attarde un peu. Cette fonction contient la définition de la variable `static n`. Même si cette variable est locale à la fonction `f` et donc invisible en dehors de cette fonction, le compilateur va lui réserver une place dans la même zone que les variables globales. La valeur de cette variable `static` sera initialisée une seule fois : au démarrage du programme. Même si cette variable parait être locale, elle ne sera jamais réinitialisée lors d'un appel à la fonction `f`. Comme cette variable est stockée en dehors de la pile, elle conserve sa valeur d'une invocation à l'autre de la fonction `f`. Ceci est illustré par l'exécution du programme qui produit la sortie suivante.

```
Minimum: -2.400000
n est à  l'adresse 0x100001078 et loc à l'adresse 0x7fff5fbfe1cc
f, n=0
f, n=1
Minimum(0.0,1.1)=0.000000
```

Le dernier point à mentionner concernant cet exemple est relatif à la fonction `min` qui est utilisée dans la fonction `main`. Le module `prog.c` étant linké avec `module.c` et `min.c`, le linker associe à ce nom de fonction la déclaration qui se trouve dans le fichier `min.c`. La déclaration de la fonction `min` qui se trouve dans `module.c` est `static`, elle ne peut donc pas être utilisée en dehors de ce module.

2.6.3 Traitement des erreurs

Certaines fonctions de la librairie et certains appels systèmes réussissent toujours. C'est le cas par exemple pour getpid(2). D'autres fonctions peuvent échouer et il est important de tester la valeur de retour de chaque fonction/appel système utilisé pour pouvoir réagir correctement à toute erreur. Pour certaines fonctions ou appels systèmes, il est parfois nécessaire de fournir à l'utilisateur plus d'information sur l'erreur qui s'est produite. La valeur de retour utilisée pour la plupart des fonctions de la libraire et appels systèmes (souvent un `int` ou un pointeur), ne permet pas de fournir de l'information précise sur l'erreur qui s'est produite.

Les systèmes Unix utilisent la variable globale *errno* pour résoudre ce problème et permettre à une fonction de la librairie ou un appel système qui a échoué de donner plus de détails sur les raisons de l'échec. Cette variable globale est définie dans errno.h qui doit être inclus par tout programme voulant tester ces codes d'erreur. Cette variable est de type `int` et *errno.h* contient les définitions des constantes correspondants aux cas d'erreurs possibles. Il faut noter que la librairie standard fournit également les fonctions perror(3) et strerror(3) qui facilitent l'écriture de messages d'erreur compréhensibles pour l'utilisateur.

A titre d'exemple, le programme ci-dessous utilise strerror(3) pour afficher un message d'erreur plus parlant lors d'appels erronés à la fonction setenv(3).

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

  if(setenv(NULL,NULL,1)!=0) {
    fprintf(stderr,"Erreur : errno=%d %s\n",errno,strerror(errno));
```

```
  }
  if(setenv("PATH=","/usr/bin",1)!=0) {
    fprintf(stderr,"Erreur : errno=%d %s\n",errno,strerror(errno));
   }


}
```

Note : La valeur de `errno` n'indique pas la réussite ou l'échec d'une fonction

Il faut noter que la variable `errno` n'est modifiée par les fonctions de la librairie ou les appels systèmes que si l'appel échoue. Si l'appel réussit, la valeur de `errno` n'est pas modifiée. Cela implique qu'il ne faut surtout pas tester la valeur de `errno` pour déterminer si une fonction de la librairie a échoué ou réussi. Il ne faut surtout par utiliser le pattern suivant :

```
setenv("PATH","/usr/bin",1);
if(errno!=0) {
  fprintf(stderr,"Erreur : errno=%d %s\n",errno,strerror(errno);
}
```

Le code ci-dessus est erroné car il ne teste pas la valeur de retour de setenv(3). Comme les fonctions de la librairie et les appels systèmes ne modifient `errno` que lorsqu'une erreur survient, le code ci-dessus pourrait afficher un message d'erreur relatif à un appel système précédent qui n'a absolument rien à voir avec l'appel à la fonction setenv(3). Le code correct est évidemment de tester la valeur de retour de setenv(3) :

```
err=setenv("PATH","/usr/bin",1);
if(err!=0) {
  fprintf(stderr,"Erreur : errno=%d %s\n",errno,strerror(errno));
}
```

CHAPITRE 3

Structure des ordinateurs

3.1 Organisation des ordinateurs

Pour bien comprendre la façon dont les programmes s'exécutent sur un ordinateur, il est nécessaire de connaitre quelques principes de base sur l'architecture des ordinateurs et de leur organisation.

Un des premiers principes fondateurs est le modèle d'architecture de *von Neumann*. Ce modèle d'architecture a été introduit durant le développement des premiers ordinateurs pendant la seconde guerre mondiale mais reste tout à fait valide aujourd'hui [Krakowiak2011]. La partie la plus intéressante de ce modèle concerne les fonctions de calcul d'un ordinateur. Il postule qu'un ordinateur est organisé autour de deux types de dispositifs :

— L'unité centrale ou *processeur*. Cette unité centrale peut être décomposée en deux parties : l'unité de commande et l'unité arithmétique et logique. L'unité arithmétique et logique regroupe les circuits électroniques qui permettent d'effectuer les opérations arithmétiques (addition, soustraction, division, ...) et logiques. C'est cette unité qui réalise les calculs proprement dits. L'unité de commande permet quant à elle de charger, décoder et exécuter les instructions du programme qui sont stockées en mémoire.

— La *mémoire*. Celle-ci joue un double rôle. Elle stocke à la fois les données qui sont traitées par le programme mais aussi les instructions qui composent celui-ci. Cette utilisation de la mémoire pour stocker les données et une représentation binaire du programme à exécuter sont un des principes fondamentaux du fonctionnement des ordinateurs actuels.

La figure ci-dessous illustre les principaux éléments du modèle de von Neumann.

Les technologies utilisées pour construire les processeurs et la mémoire ont fortement évolué depuis les premiers ordinateurs, mais les principes fondamentaux restent applicables. En première approximation, on peut considérer la mémoire comme étant un dispositif qui permet de stocker des données binaires. La mémoire est découpée en blocs d'un octet. Chacun de ces blocs est identifié par une adresse, qui est elle aussi représentée sous la forme d'un nombre binaire. Une mémoire qui permet de stocker 2^k bytes de données utilisera au minimum k bits pour représenter l'adresse d'une zone mémoire. Ainsi, une mémoire pouvant stocker 64 millions de bytes doit utiliser au moins 26 bits d'adresse. En pratique, les processeurs des ordinateurs de bureau utilisent 32 ou 64 bits pour représenter les adresses en mémoire. D'anciens processeurs utilisaient 16 ou 20 bits d'adresse. Le nombre de bits utilisés pour représenter une adresse en mémoire limite la capacité totale de mémoire adressable par un processeur. Ainsi, un processeur qui utilise des adresses sur 32 bits n'est pas capable physiquement d'adresser plus de 4 GBytes de mémoire.

En pratique, l'organisation physique d'un ordinateur actuel est plus complexe que le modèle de von Neumann. Schématiquement, on peut considérer l'organisation présentée dans la figure ci-dessous. Le processeur est directement connecté à la mémoire via un *bus* de communication rapide. Ce bus permet des échanges de données et d'instructions efficaces entre la mémoire et le processeur. Outre le processeur et la mémoire, un troisième dispositif, souvent baptisé adaptateur de bus est connecté au bus processeur-mémoire. Cet adaptateur permet au processeur d'accéder aux dispositifs de stockage ou aux dispositifs d'entrées-sorties tels que le clavier, la souris ou les cartes réseau. En pratique, cela se réalise en connectant les différents dispositifs à un autre bus de communication (PCI, SCSI, ...) et en utilisant un adaptateur de bus qui est capable de traduire les commandes venant du processeur.

Différentes technologies ont été mises en oeuvre pour construire les mémoires utilisées dans les ordinateurs. Aujourd'hui, les technologies les plus courantes sont les mémoires de type *SRAM* et les mémoires de type *DRAM*. Dans une *SRAM*, l'information est stockée sous la forme d'un courant électrique qui passe ou ne passe pas à

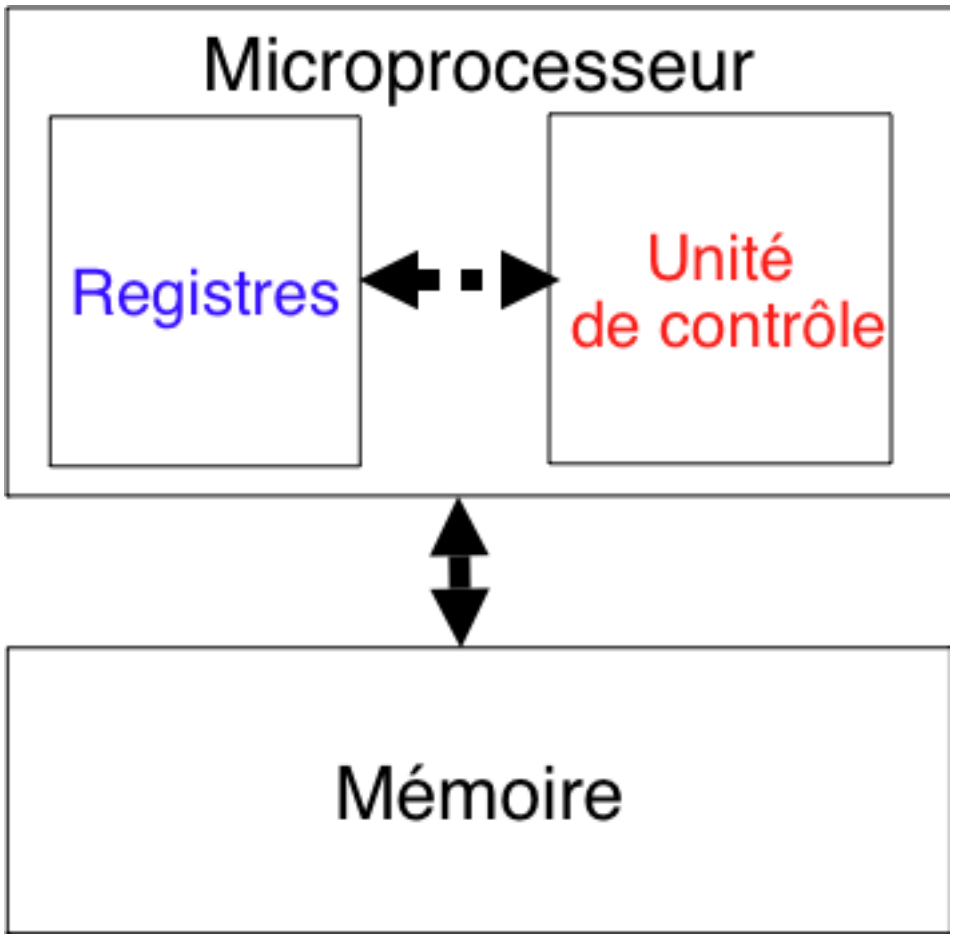


FIGURE 3.1 – Modèle de von Neumann

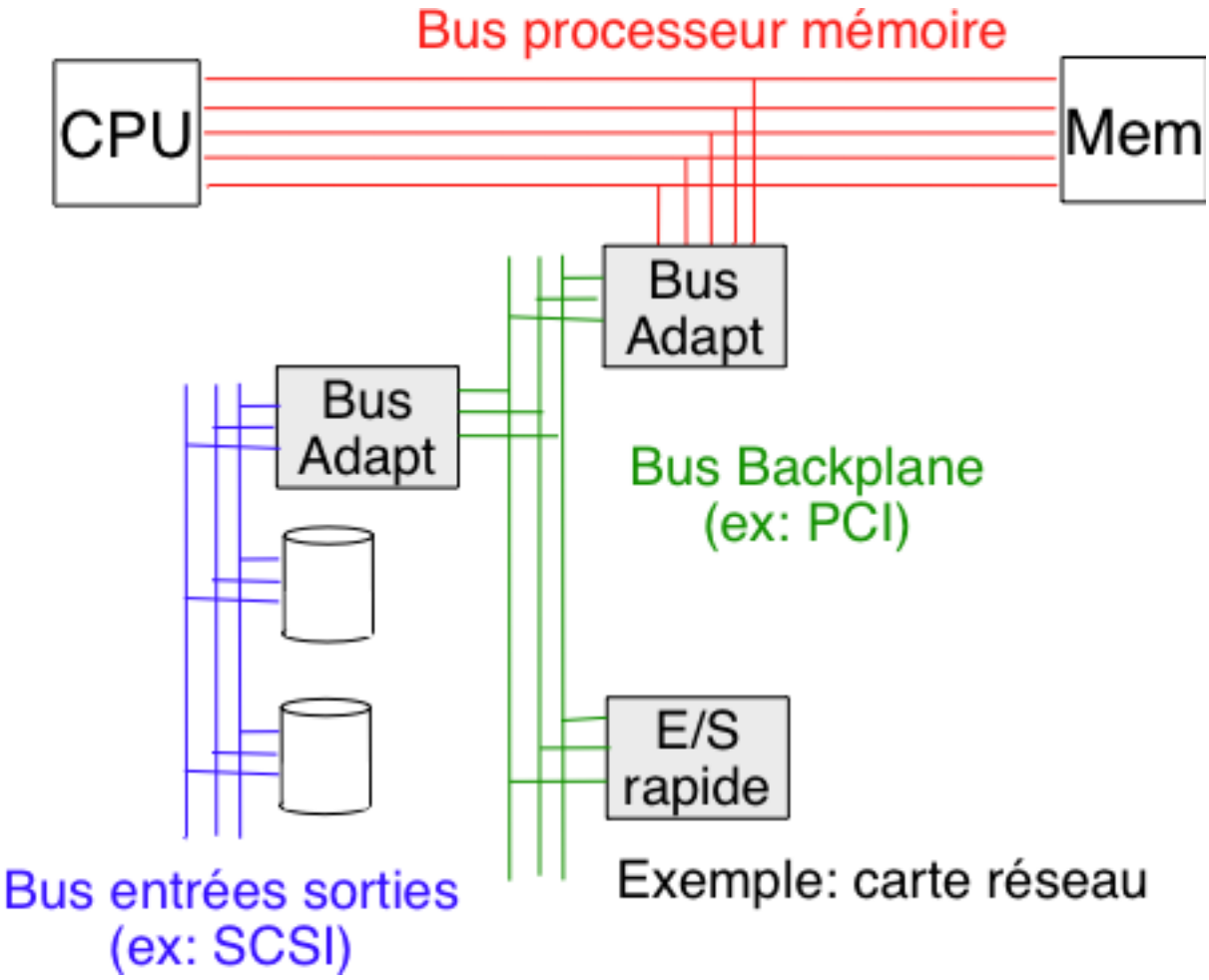


FIGURE 3.2 – Architecture d'un ordinateur actuel

un endroit donné. L'avantage de cette technologie est que le temps d'accès à une donnée stockée en *SRAM* est assez faible. Malheureusement, leur inconvénient majeur est leur grande consommation électrique qui empêche de développer des mémoires de grande capacité. Aujourd'hui, les *SRAM* les plus grandes ont une capacité de seulement 12 MBytes [HennessyPatterson].

Les *DRAM* sont totalement différentes des *SRAM* d'un point de vue électronique. Dans une mémoire de type *DRAM*, c'est la présence ou l'absence d'une charge (de quelques électrons à quelques dizaines d'électrons) dans un condensateur qui représente la valeur `0` ou `1`. Il est possible de construire des *DRAM* de très grande taille, jusqu'à 1 GByte par chip [HennessyPatterson]. C'est la raison pour laquelle on retrouve très largement des mémoires de type *DRAM* dans les ordinateurs. Malheureusement, leurs performances sont nettement moins bonnes que les mémoires de type *SRAM*. En pratique, une mémoire *DRAM* actuelle peut être vue comme étant équivalente à une grille [Drepper2007]. Les adresses peuvent être vues comme étant composées d'un numéro de colonne et d'un numéro de ligne. Pour lire ou écrire une donnée en mémoire *DRAM*, le processeur doit d'abord indiquer la ligne qu'il souhaite lire et ensuite la colonne. Ces deux opérations sont successives. Lorsque la mémoire a reçu la ligne et la colonne demandées, elle peut commencer le transfert de la donnée. En pratique, les mémoires *DRAM* sont optimisées pour fournir un débit de transfert élevé, mais elles ont une latence élevée. Cela implique que dans une mémoire *DRAM*, il est plus rapide de lire ou d'écrire un bloc de 128 bits successifs que quatre blocs de 32 bits à des endroits différents en mémoire. A titre d'exemple, le tableau ci-dessous, extrait de [HP] fournit le taux de transfert maximum de différentes technologies de *DRAM*.

Technologie	Fréquence [MHz]	Débit [MB/sec]
SDRAM	200	1064
RDRAM	400	1600
DDR-1	266	2656
DDR-2	333	5328
DDR-2	400	6400
DDR-3	400	6400
DDR-3	533	8500
DDR-3	667	10600

Le processeur interagit en permanence avec la mémoire, que ce soit pour charger des données à traiter ou pour charger les instructions à exécuter. Tant les données que les instructions sont représentées sous la forme de nombres en notation binaire. Certains processeurs utilisent des instructions de taille fixe. Par exemple, chaque instruction est encodée sous la forme d'un mot de 32 bits. D'autres processeurs, comme ceux qui implémentent l'architecture [IA32], utilisent des instructions qui sont encodées sous la forme d'un nombre variable de bytes. Ces choix d'encodage des instructions influencent la façon dont les processeurs sont implémentés d'un point de vue micro-électronique, mais ont assez peu d'impact sur le développeur de programmes. L'élément qui est important de bien comprendre est que le processeur doit en permanence charger des données et des instructions depuis la mémoire lors de l'exécution d'un programme.

Outre des unités de calcul, un processeur contient plusieurs registres. Un *registre* est une zone de mémoire très rapide se trouvant sur le processeur. Sur les processeurs actuels, cette zone de mémoire permet de stocker un mot de 32 bits ou un long mot de 64 bits. Les premiers processeurs disposaient d'un registre unique baptisé l'*accumulateur*. Les processeurs actuels en contiennent généralement une ou quelques dizaines. Chaque registre est identifié par un nom ou un numéro et les instructions du processeur permettent d'accéder directement aux données se trouvant dans un registre particulier. Les registres sont les mémoires les plus rapides qui sont disponibles sur un ordinateur. Malheureusement, ils sont en nombre très limité et il est impossible de faire fonctionner un programme non trivial en utilisant uniquement des registres.

Du point de vue des performances, il serait préférable de pouvoir construire un ordinateur équipé uniquement de *SRAM*. Malheureusement, au niveau de la capacité et du prix, c'est impossible sauf pour de rares applications bien spécifiques qui nécessitent de hautes performances et se contentent d'une capacité limitée. Les ordinateurs actuels utilisent en même temps de la mémoire *SRAM* et de la mémoire *DRAM*. Avec les registres, les *SRAM* et les *DRAM* composent les trois premiers niveaux de la *hiérarchie de mémoire*.

Le tableau ci-dessous, extrait de [BryantOHallaron2011] , compare les temps d'accès entre les mémoires *SRAM* et les mémoires *DRAM* à différentes périodes.

Année	Accès SRAM	Accès DRAM
1980	300 ns	375 ns
1985	150 ns	200 ns
1990	35 ns	100 ns
1995	15 ns	70 ns
2000	3 ns	60 ns
2005	2 ns	50 ns
2010	1.5 ns	40 ns

Cette évolution des temps d'accès doit être mise en parallèle avec l'évolution des performances des processeurs. En 1980, le processeur Intel 8080 fonctionnait avec une horloge de 1 MHz et accédait à la mémoire toutes les 1000 ns. A cette époque, la mémoire était nettement plus rapide que le processeur. En 1990, par contre, le processeur Intel 80386 accédait à la mémoire en moyenne toutes les 50 ns. Couplé à une mémoire uniquement de type DRAM, il était ralenti par cette mémoire. En 2000, le Pentium-III avait un cycle de 1.6 ns, plus rapide que les meilleures mémoires disponibles à l'époque. Il en va de même aujourd'hui où les temps de cycle sont inférieurs au temps d'accès des mémoires. Même s'il existe des solutions techniques pour mitiger ce problème, la différence de performance croissante entre la mémoire et le processeur est un des facteurs qui limitent les améliorations des performances de nombreux programmes.

Une première solution pour combiner la *SRAM* et la *DRAM* serait de réserver par exemple les adresses basses à la *SRAM* qui est plus performante et d'utiliser la *DRAM* pour les adresses hautes. Avec cette solution, le programme stocké dans la *SRAM* pourrait s'exécuter plus rapidement que le programme stocké en *DRAM*. Afin de permettre à tous les programmes de pouvoir utiliser la *SRAM*, on pourrait imaginer que le système d'exploitation fournisse des fonctions qui permettent aux applications de demander la taille de la mémoire *SRAM* disponible et de déplacer des parties d'un programme et des données en *SRAM*. Ce genre de solution obligerait chaque application à pouvoir déterminer quelles sont les instructions à exécuter et quelles données doivent être placées en mémoire *SRAM* pour obtenir les meilleures performances. Même si en théorie, ce genre de solution est envisageable, en pratique, elle a très peu de chances de pouvoir fonctionner.

La deuxième solution est d'utiliser le principe de la *mémoire cache*. Une *mémoire cache* est une mémoire de faible capacité mais rapide. La mémoire cache peut stocker des données provenant de mémoires de plus grande capacité mais plus lentes. Cette mémoire cache sert d'interface entre le processeur et la mémoire principale. Toutes les demandes d'accès à la mémoire principale passent par la mémoire cache comme illustré dans la figure ci-dessous.

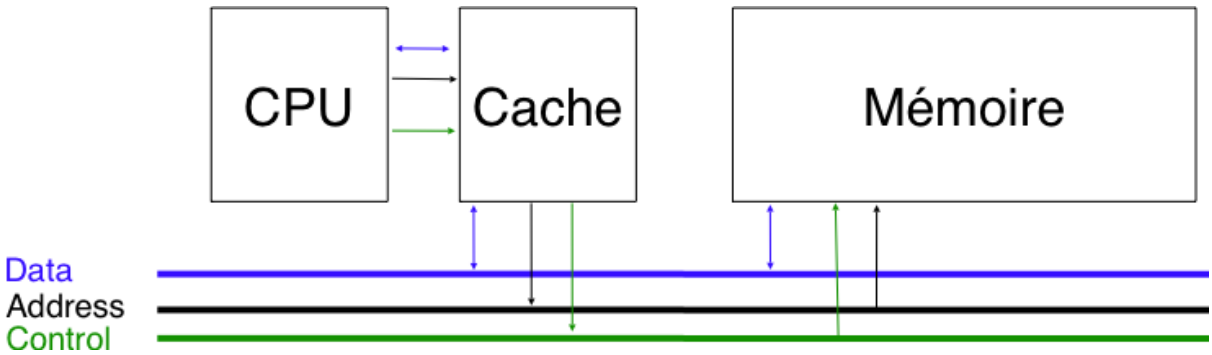


FIGURE 3.3 – Le processeur, la mémoire cache et la mémoire principale

On utilise des mémoires caches dans de nombreux systèmes informatiques de façon à améliorer leurs performances. Ces mémoires caches utilisent en fait le *principe de localité*. En pratique, deux types de localité doivent être considérés. Tout d'abord, il y a la *localité temporelle*. Si un processeur a accédé à la mémoire à l'adresse *A* à l'instant *t*, il est fort probable qu'il accédera encore à cette adresse dans les instants qui suivent. La localité temporelle apparaît notamment lors de l'exécution de longues boucles qui exécutent à de nombreuses reprises les mêmes instructions. Le second type de localité est la *localité spatiale*. Celle-ci implique que si un programme a accédé à l'adresse *A* à l'instant *t*, il est fort probable qu'il accédera aux adresses proches de *A* comme *A+4*, *A-4* dans les instants qui suivent. Cette localité apparaît par exemple lorsqu'un programme traite un vecteur stocké en mémoire.

Les mémoires caches exploitent ces principes de localité en stockant de façon transparente les instructions et

les données les plus récemment utilisées. D'un point de vue physique, on peut voir le processeur comme étant connecté à la (ou parfois les) mémoire cache qui est elle-même connectée à la mémoire *RAM*. Les opérations de lecture en mémoire se déroulent généralement comme suit. Chaque fois que le processeur a besoin de lire une donnée se trouvant à une adresse, il fournit l'adresse demandée à la mémoire cache. Si la donnée correspondant à cette adresse est présente en mémoire cache, celle-ci répond directement au processeur. Sinon, la mémoire cache interroge la mémoire *RAM*, se met à jour et ensuite fournit la donnée demandée au processeur. Ce mode de fonctionnement permet à la mémoire cache de se mettre à jour au fur et à mesure des demandes faites par le processeur afin de profiter de la localité temporelle. Pour profiter de la localité spatiale, la plupart des caches se mettent à jour en chargeant directement une *ligne de cache* qui peut compter jusqu'à quelques dizaines d'adresses consécutives en mémoire. Ce chargement d'une ligne complète de cache permet également de profiter des mémoires *DRAM* récentes qui sont souvent optimisées pour fournir des débits de transfert élevés pour de longs blocs consécutifs en mémoire. La figure ci-dessous illustre graphiquement la hiérarchie de mémoires dans un ordinateur.

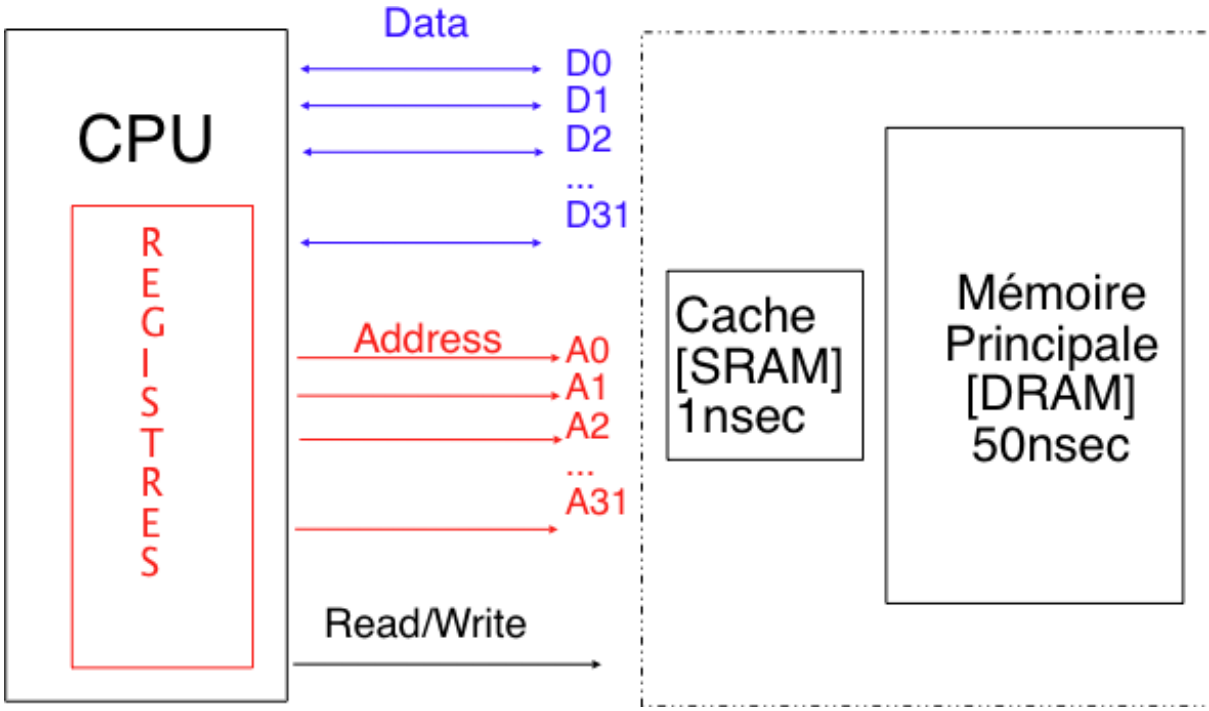


FIGURE 3.4 – La hiérarchie de mémoires

Pour les opérations d'écriture, la situation est plus compliquée. Si le processeur écrit l'information x à l'adresse A en mémoire, il faudrait idéalement que cette valeur soit écrite simultanément en mémoire cache et en mémoire *RAM* de façon à s'assurer que la mémoire *RAM* contienne toujours des données à jour. La stratégie d'écriture la plus simple est baptisée *write through*. Avec cette stratégie, toute demande d'écriture venant du processeur donne lieu à une écriture en mémoire cache et une écriture en mémoire *RAM*. Cette stratégie garantit qu'à tout moment la mémoire cache et la mémoire *RAM* contiennent la même information. Malheureusement, d'un point de vue des performances, cette technique rabaisse les performances de la mémoire cache à celles de la mémoire *RAM*. Vu la différence de performance entre les deux types de mémoires, cette stratégie n'est plus acceptable aujourd'hui. L'alternative est d'utiliser la technique du *write back*. Avec cette technique, toute écriture est faite en *mémoire cache* directement. Cela permet d'obtenir de très bonnes performances pour les écritures. Une donnée modifiée n'est réécrite en mémoire *RAM* que lorsqu'elle doit être retirée de la mémoire cache. Cette écriture est faite automatiquement par la mémoire cache. Pour la plupart des programmes, la gestion des opérations d'écriture est transparente. Il faut cependant être attentif à la technique d'écriture utilisée lorsque plusieurs dispositifs peuvent accéder directement à la mémoire *RAM* sans passer par le processeur. C'est le cas par exemple pour certaines cartes réseaux ou certains contrôleurs de disque dur. Pour des raisons de performances, ces dispositifs peuvent copier des données directement de la mémoire *RAM* vers le réseau ou un disque dur. Si une écriture de type *write-back* est utilisée, le système d'exploitation doit veiller à ce que les données écrites par le processeur en cache aient bien été écrites également en mémoire *RAM* avant d'autoriser la carte réseau ou le contrôleur de disque à effectuer un transfert.

3.2 Etude de cas : Architecture [IA32]

Pour comprendre le fonctionnement d'un microprocesseur, la solution la plus efficace est de considérer une architecture en particulier et de voir comment fonctionnent les processeurs qui l'implémentent. Dans cette section, nous analysons brièvement le fonctionnement des processeurs [1] de la famille [IA32].

Cette architecture recouvre un grand nombre de variantes qui ont leur spécificités propre. Une descriptions détaillée de cette architecture est disponible dans [IA32]. Nous nous limiterons à un très petit sous-ensemble de cette architecture dans le cadre de ce cours. Une analyse complète de l'architecture [IA32] occupe plusieurs centaines de pages dans des livres de référence [BryantOHallaron2011] [Hyde2010].

L'architecture [IA32] est supportée par différents types de processeurs. Certains utilisent des registres et des bus de données de 32 bits. D'autres, plus récents utilisent des registres de 64 bits. Il y a des différences importantes entre ces deux architectures. Comme les processeurs récents supportent à la fois les modes 32 bits et 64 bits, nous nous limiterons à l'architecture 32 bits.

Un des éléments importants d'un processeur tel que ceux de l'architecture [IA32] sont ses registres. Un processeur [IA32] dispose de huit registres génériques. Ceux-ci ont été baptisés `EAX`, `EBX`, `ECX`, `EDX`, `EBP`, `ESI`, `EDI` et `ESP`. Ces registres peuvent stocker des données sous forme binaire. Dans l'architecture [IA32], ils ont une taille de 32 bits. Cela implique que chaque registre peut contenir un nombre ou une adresse puisque les entiers (`int` en C) et les adresses (pointeurs `*` en C sur [IA32]) sont tous les deux encodés sur 32 bits dans l'architecture [IA32]. Cette capacité à stocker des données ou des adresses à l'intérieur d'un même registre est un des points clés de la flexibilité des microprocesseurs.

Deux de ces registres, `EBP` et `ESP` sont utilisés dans la gestion de la pile comme nous le verrons plus tard. Les autres registres peuvent être utilisés directement par le programmeur. En outre, tout processeur contient un registre spécial qui stocke à tout moment l'adresse de l'instruction courante en mémoire. Ce registre est souvent dénommé le *compteur de programme* ou *program counter* (*PC*) en anglais. Dans l'architecture [IA32], c'est le registre `EIP` qui stocke l'*Instruction Pointer* qui joue ce rôle. Ce registre ne peut pas être utilisé pour effectuer des opérations arithmétiques. Il peut cependant être modifié par les instructions de saut comme nous le verrons plus tard et joue un rôle essentiel dans l'implémentation des instructions de contrôle.

Outre ces registres génériques, les processeurs de la famille [IA32] contiennent aussi des registres spécialisés pour manipuler les nombres en virgule flottante (`float` et `double`). Nous ne les analyserons pas dans le cadre de ce cours. Par contre, les processeurs [IA32] contiennent également des drapeaux regroupés dans le registre `eflags`. Ceux-ci sont utilisés pour implémenter différents tests et comparaisons.

Les processeurs qui implémentent les spécifications [IA32] supportent les types de données repris dans le tableau ci-dessous.

Type	Taille (bytes)	Suffixe assembleur
`char`	1	b
`short`	2	w
`int`	4	l
`long int`	4	l
`void *`	4	l

Dans les sections qui suivent, nous analysons quelques instructions de l'architecture [IA32] qui permettent de manipuler des nombres entiers en commençant par les instructions de transfert entre la mémoire et les registres.

3.2.1 Les instructions `mov`

Les instructions de la famille `mov` [2] permettent de déplacer des données entre registres ou depuis la mémoire vers un registre ou enfin d'un registre vers une zone mémoire. Ces instructions sont essentielles car elles permettent au processeur de récupérer les données qui sont stockées en mémoire mais aussi de sauvegarder en mémoire le résultat d'un calcul effectué par le processeur. Une instruction `mov` contient toujours deux arguments. Le premier

1. Pour une liste détaillée des processeurs de cette famille produits par intel, voir notamment http://www.intel.com/pressroom/kits/quickreffam.htm D'autres fabricants produisent également des processeurs compatibles avec l'architecture [IA32].
2. On parle de famille d'instructions car il existe de nombreuses instructions de déplacement en mémoire. Les plus simples sont suffixées par un caractère qui indique le type de données transféré. Ainsi, `movb` permet le transfert d'un byte tandis que `movl` permet le transfert d'un mot de 32 bits. Des détails sur ces instructions peuvent être obtenus dans [IA32]

spécifie la donnée à déplacer ou son adresse et la seconde l'endroit où il faut sauvegarder cette donnée ou la valeur stockée à cette adresse.

```
mov src, dest  ; déplacement de src vers dest
```

Il existe une instruction de la famille `mov` qui correspond à chaque type de donnée pouvant être déplacé. L'instruction `movb` est utilisée pour déplacer un byte, `movw` pour déplacer un mot de 16 bits et `movl` lorsqu'il faut déplacer un mot de 32 bits.

En pratique, il y a plusieurs façons de spécifier chaque argument d'une instruction `mov`. Certains auteurs utilisent le terme *mode d'adressage* pour représenter ces différents types d'arguments même si il ne s'agit pas toujours d'adresses. Le premier mode est le mode *registre*. La source et la destination d'une opération `mov` peuvent être un nom de registre. Ceux-ci sont en général préfixés avec le caractère `%`. Ainsi, `%eax` correspond au registre `EAX`. La première instruction ci-dessous déplace le mot de 32 bits stocké dans le registre `%eax` vers le registre `%ebx`. La seconde instruction elle n'a aucun effet puisqu'elle déplace le contenu du registre `%ecx` vers ce même registre.

```
movl %eax, %ebx   ; déplacement de %eax vers %ebx
movl %ecx, %ecx   ; aucun effet
```

Le deuxième mode d'adressage est le mode *immédiat*. Celui-ci ne peut être utilisé que pour l'argument *source*. Il permet de placer une constante dans un registre, par exemple pour initialiser sa valeur comme dans les exemples ci-dessous. Il se reconnaît à l'utilisation du symbole `$` comme préfixe de la constante.

```
movl $0, %eax      ; initialisation de %eax à 0
movl $1252, %ecx   ; initialisation de %ecx à 1252
```

Le troisième mode d'adressage est le mode *absolu*. Dans ce mode, l'un des arguments de l'instruction `mov` est une adresse en mémoire. Si la source est une adresse, alors l'instruction `mov` transfère le mot de 32 bits stocké à l'adresse spécifiée comme source vers le registre spécifié comme destination. Si la destination est une adresse, alors l'instruction `mov` sauvegarde la donnée source à cette adresse en mémoire. Pour illustrer cette utilisation de l'instruction `mov`, considérons la mémoire illustrée ci-dessous.

Adresse	Valeur
0x10	0x04
0x0C	0x10
0x08	0xFF
0x04	0x00
0x00	0x04

Les instructions ci-dessous sont un exemple de déplacement de données entre la mémoire et un registre et d'un registre vers la mémoire.

```
movl 0x04, %eax    ; place la valeur 0x00 (qui se trouve à l'adresse 0x04) dans %eax
movl $1252, %ecx   ; initialisation de %ecx à 1252
movl %ecx, 0x08    ; remplace 0xFF par le contenu de %ecx (1252) à l'adresse 0x08
```

Le quatrième mode d'adressage est le mode *indirect*. Plutôt que de spécifier directement une adresse, avec le mode indirect, on spécifie un registre dont la valeur est une adresse en mémoire. Ce mode indirect est équivalent à l'utilisation des pointeurs en langage C. Il se reconnait à l'utilisation de parenthèses autour du nom du registre source ou destination. L'exemple ci-dessous illustre l'utilisation de l'adressage indirect en considérant la mémoire présentée plus haut.

```
movl $0x08, %eax    ; place la valeur 0x08 dans %eax
movl (%eax), %ecx   ; place la valeur se trouvant à l'adresse qui est
                    ; dans %eax dans le registre %ecx %ecx=0xFF
movl 0x10, %eax     ; place la valeur se trouvant à l'adresse 0x10 dans %eax
movl %ecx, (%eax)   ; place le contenu de %ecx, c'est-à-dire 0xFF à l'adresse qui est contenue da
```

Le cinquième mode d'adressage est le mode avec une *base* et un *déplacement*. Ce mode peut être vu comme une extension du mode *indirect*. Il permet de lire en mémoire à une adresse qui est obtenue en additionnant un entier, positif ou négatif, à une adresse stockée dans un registre. Ce mode d'adressage joue un rôle important dans le fonctionnement de la pile comme nous le verrons d'ici peu.

```
movl $0x08, %eax      ; place la valeur 0x08 dans %eax
movl 0(%eax), %ecx    ; place la valeur (0xFF) se trouvant à l'adresse
                      ; 0x08= (0x08+0) dans le registre %ecx
movl 4(%eax), %ecx    ; place la valeur (0x10) se trouvant à l'adresse
                      ; 0x0C (0x08+4) dans le registre %ecx
movl -8(%eax), %ecx   ; place la valeur (0x04) se trouvant à l'adresse
                      ; 0x00 (0x08-8) dans le registre %ecx
```

L'architecture [IA32] supporte encore d'autres modes d'adressage. Ceux sont décrits dans [IA32] ou [BryantO-Hallaron2011]. Une autre instruction permettant de déplacer de l'information est l'instruction `leal` (load effective address). Cette instruction est parfois utilisée par les compilateurs. Elle place dans le registre destination l'adresse de son argument source plutôt que sa valeur. Ainsi `leal 4(%esp) %edx` placera dans le registre `%edx` l'adresse de son argument source, c'est-à-dire l'adresse contenue dans `%esp+4`.

3.2.2 Instructions arithmétiques et logiques

La deuxième famille d'instructions importante sur un processeur sont les instructions qui permettent d'effectuer les opérations arithmétiques et logiques. Voici quelques exemples d'instructions arithmétiques et logiques supportées par l'architecture [IA32].

Les instructions les plus simples sont celles qui prennent un seul argument. Il s'agit de :

- `inc` qui incrémente d'une unité la valeur stockée dans le registre/l'adresse fournie en argument et sauvegarde le résultat de l'incrémentation au même endroit. Cette instruction peut être utilisée pour implémenter des compteurs de boucles.
- `dec` est équivalente à `inc` mais décrémente son argument.
- `not` qui applique l'opération logique `NOT` à son argument et stocke le résultat à cet endroit

Il existe une variante de chacune de ces instructions pour chaque type de données à manipuler. Cette variante se reconnait grâce au dernier caractère de l'instruction (`b` pour byte, `w` pour un mot de 16 bits et `l` pour un mot de 32 bits). Nous nous limiterons aux instructions qui manipulent des mots de 32 bits.

```
movl    $0x12345678, %ecx   ; initialisation
notl    %ecx                ; calcul de NOT
movl    $0, %eax            ; %eax=0
incl    %eax                ; %eax++
```

L'architecture [IA32] supporte également des instructions arithmétiques et logiques prenant chacune deux arguments.

- `add` permet d'additionner deux nombres entiers. `add` prend comme arguments une source et une destination et place dans la destination la somme de ses deux arguments.
- `sub` permet de soustraire le premier argument du second et stocke le résultat dans le second
- `mul` permet de multiplier des nombres entiers non-signés (`imul` est le pendant de `mul` pour la multiplication de nombres signés)
- `div` permet la division de nombres entiers non-signés.
- `shl` (resp. `shr`) permet de réaliser un décalage logique vers la gauche (resp. droite)
- `xor` calcule un ou exclusif entre ses deux arguments et sauvegarde le résultat dans le second
- `and` calcule la conjonction logique entre ses deux arguments et sauvegarde le résultat dans le second

Pour illustrer le fonctionnement de ces instructions, considérons une mémoire hypothétique contenant les données suivantes. Supposons que la variable entière `a` est stockée à l'adresse `0x04`, `b` à l'adresse `0x08` et `c` à l'adresse `0x0C`.

Adresse	Variable	Valeur
0x0C	c	0x00
0x08	b	0xFF
0x04	a	0x02
0x00	__	0x01

Les trois premières instructions ci-dessous sont équivalentes à l'expression C `a=a+1;`. Pour implémenter une telle opération en C, il faut d'abord charger la valeur de la variable dans un registre. Ensuite le processeur effectue l'opération arithmétique. Enfin le résultat est sauvegardé en mémoire. Après ces trois instructions, la valeur `0x03`

est stockée à l'adresse `0x04` qui correspond à la variable `a`. Les trois dernières instructions calculent `a=b-c;`. On remarquera que le programmeur a choisi de d'abord charger la valeur de la variable `b` dans le registre `%eax`. Ensuite il utilise l'instruction `subl` en mode d'adressage immédiat pour placer dans `%eax` le résultat de la soustraction entre `%eax` et la donnée se trouvant à l'adresse `0x0C`. Enfin, le contenu de `%eax` est sauvé à l'adresse correspondant à la variable `a`.

```
movl   0x04, %eax     ; %eax=a
addl   $1, %eax       ; %eax++
movl   %eax, 0x04     ; a=%eax
movl   0x08, %eax     ; %eax=b
subl   0x0c, %eax     ; %eax=b-c
movl   %eax, 0x04     ; a=%eax
```

L'exemple ci-dessous présente la traduction directe[3] d'un fragment de programme C utilisant des variables globales en langage assembleur.

```
int j,k,g,l;
// ...
l=g^j;
j=j|k;
g=l<<6;
```

Dans le code assembleur, les noms de variables tels que `g` ou `j` correspondent à l'adresse mémoire à laquelle la variable est stockée.

```
movl    g, %eax  ; %eax=g
xorl    j, %eax  ; %eax=g^j
movl    %eax, l  ; l=%eax
movl    j, %eax  ; %eax=j
orl     k, %eax  ; %eax=j|k
movl    %eax, j  ; j=%eax
movl    l, %eax  ; %eax=l
shll    $6, %eax ; %eax=%eax << 6
movl    %eax, g  ; g=%eax
```

Les opérations arithmétiques telles que la multiplication ou la division sont plus complexes que les opérations qui ont été présentées ci-dessus. En toute généralité, la multiplication entre deux nombres de 32 bits peut donner un résultat sur 64 bits qui ne pourra donc pas être stocké entièrement dans un registre. De la même manière, une division entière retourne un quotient et un reste qui sont tous les deux sur 32 bits. L'utilisation des instructions de division et de multiplication nécessite de prendre ces problèmes en compte. Nous ne les aborderons pas dans ce cours. Des détails complémentaires sont disponibles dans [IA32] et [BryantOHallaron2011] notamment.

3.2.3 Les instructions de comparaison

Outre les opérations arithmétiques, un processeur doit être capable de réaliser des comparaisons. Ces comparaisons sont nécessaires pour implémenter des tests tels que `if (condition) { ... } else { ... }`. Sur les processeurs [IA32], les comparaisons utilisent des drapeaux qui sont mis à jour par le processeur après l'exécution de certaines instructions. Ceux-ci sont regroupés dans le registre `eflags`. Les principaux drapeaux sont :

- *ZF* (Zero Flag) : ce drapeau indique si le résultat de la dernière opération était zéro
- *SF* (Sign Flag) : indique si le résultat de la dernière instruction était négatif
- *CF* (Carry Flag) : indique si le résultat de la dernière instruction arithmétique non signée nécessitait plus de 32 bits pour être stocké
- *OF* (Overflow Flag) : indique si le résultat de la dernière instruction arithmétique signée a provoqué un dépassement de capacité

Nous utiliserons principalement les drapeaux *ZF* et *SF* dans ce chapitre. Ces drapeaux peuvent être fixés par les instructions arithmétiques standard, mais aussi par des instructions dédiées comme `cmp` et `test`. L'instruction `cmp` effectue l'équivalent d'une soustraction et met à jour les drapeaux *CF* et *SF* mais sans sauvegarder son

3. Cette traduction et la plupart des traductions utilisées dans ce chapitre ont été obtenues en utilisant l'interface web de démo du compilateur llvm qui a été configuré pour générer du code 32 bits sans optimisation. Quelques détails ont été supprimés du code assembleur pour le rendre plus compact.

résultat dans un registre. L'instruction `test` effectue elle une conjonction logique sans sauvegarder son résultat mais en mettant à jour les drapeaux.

Ces instructions de comparaison peuvent être utilisées avec les instructions `set` qui permettent de fixer la valeur d'un registre en fonction des valeurs de certains drapeaux du registre `eflags`. Chaque instruction `set` prend comme argument un registre. Pour des raisons historiques, ces instructions modifient uniquement les bits de poids faible du registre indiqué et non le registre complet. C'est un détail qui est lié à l'histoire de l'architecture [IA32].

- `sete` met le registre argument à la valeur du drapeau *ZF*. Permet d'implémenter une égalité.
- `sets` met le registre argument à la valeur du drapeau *SF*
- `setg` place dans le registre argument la valeur `~SF & ~ZF` (tout en prenant en compte les dépassements éventuels avec *OF*). Permet d'implémenter la condition >.
- `setl` place dans le registre argument la valeur de `SF` (tout en prenant en compte les dépassements éventuels avec *OF*). Permet d'implémenter notamment la condition <=.

A titre d'illustration, voici quelques expressions logiques en C et leur implémentation en assembleur lorsque les variables utilisées sont toutes des variables globales.

```
r=(h>1);
r=(j==0);
r=g<=h;
r=(j==h);
```

Le programme assembleur utilise une instruction `cmpl` pour effectuer la comparaison. Ensuite, une instruction `set` permet de fixer la valeur du byte de poids faible de `%eax` et une instruction (`movzbl`) permettant de transformer ce byte en un mot de 32 bits afin de pouvoir le stocker en mémoire. Cette traduction a été obtenue avec *llvm*, d'autres compilateurs peuvent générer du code un peu différent.

```
cmpl    $1, h         ; comparaison
setg    %al           ; %al est le byte de poids faible de %eax
movzbl  %al, %ecx     ; copie le byte dans %ecx
movl    %ecx, r       ; sauvegarde du résultat dans r

cmpl    $0, j         ; comparaison
sete    %al           ; fixe le byte de poids faible de %eax
movzbl  %al, %ecx
movl    %ecx, r       ; sauvegarde du résultat dans r

movl    g, %ecx
cmpl    h, %ecx       ; comparaison entre g et h
setl    %al           ; fixe le byte de poids faible de %eax
movzbl  %al, %ecx
movl    %ecx, r

movl    j, %ecx
cmpl    h, %ecx       ; comparaison entre j et h
sete    %al
movzbl  %al, %ecx
movl    %ecx, r
```

3.2.4 Les instructions de saut

Les instructions de saut sont des instructions de base pour tous les processeurs. Elles permettent de modifier la valeur du compteur de programme `%eip` de façon à modifier l'ordre d'exécution des instructions. Elles sont nécessaires pour implémenter les tests, les boucles et les appels de fonction. Les premiers langages de programmation et des langages tels que BASIC ou FORTRAN disposent d'une construction similaire avec l'instruction `goto`. Cependant, l'utilisation de l'instruction `goto` dans des programmes de haut niveau rend souvent le code difficile à lire et de nombreux langages de programmation n'ont plus de `goto` [Dijkstra1968]. Contrairement à Java, le C contient une instruction `goto`, mais son utilisation est fortement découragée. En C, l'instruction `goto` prend comme argument une étiquette (label en anglais). Lors de l'exécution d'un `goto`, le programme saute directement à l'exécution de l'instruction qui suit le label indiqué. Ceci est illustré dans l'exemple ci-dessous :

```
  int v=0;
  for(int i=0;i<SIZE;i++)
    for(int j=0;j<SIZE;j++) {
      if(m[i][j]>MVAL) {
        v=m[i][j];
        goto suite;
      }
    }
  printf("aucune valeur supérieure à %d\n",MVAL,v);
  goto fin;
 suite:
  printf("première valeur supérieure à %d : %d\n",MVAL,v);
 fin:
  return(EXIT_SUCCESS);
```

Si l'utilisation `goto` est en pratique prohibée dans la plupart des langages de programmation, en assembleur, les instructions de saut sont inévitables. L'instruction de saut la plus simple est `jmp`. Elle prend généralement comme argument une étiquette. Dans ce cas, l'exécution du programme après l'instruction `jmp` se poursuivra par l'exécution de l'instruction qui se trouve à l'adresse correspondant à l'étiquette fournie en argument. Il est également possible d'utiliser l'instruction `jmp` avec un registre comme argument. Ainsi, l'instruction `jmp *%eax` indique que l'exécution du programme doit se poursuivre par l'exécution de l'instruction se trouvant à l'adresse qui est contenue dans le registre `%eax`.

Il existe plusieurs variantes conditionnelles de l'instruction `jmp`. Ces variantes sont exécutées uniquement si la condition correspondante est vérifiée. Les variantes les plus fréquentes sont :

- `je` : saut si égal (teste le drapeau *ZF*) (inverse : `jne`)
- `js` : saut si négatif (teste le drapeau *SF*) (inverse : `jns`)
- `jg` : saut si strictement supérieur (teste les drapeaux *SF* et *ZF* et prend en compte un overflow éventuel) (inverse : `jl`)
- `jge` : saut si supérieur ou égal (teste le drapeaux *SF* et prend en compte un overflow éventuel) (inverse : `jle`)

Ces instructions de saut conditionnel sont utilisées pour implémenter notamment des expressions `if (condition) { ... } else { ... }` en C. Voici quelques traductions réalisées par un compilateur C en guise d'exemple.

```
if(j==0)
  r=1;

if(j>g)
  r=2;
else
  r=3;

if (j>=g)
  r=4;
```

Avant d'analyser la traduction de ce programme en assembleur, il est utile de le réécrire en utilisant l'instruction `goto` afin de s'approcher du fonctionnement de l'assembleur.

```
if(j!=0) { goto diff; }
  r=1;
diff:
  // suite

if(j<=g) { goto else; }
  r=2;
  goto fin;
else:
  r=3;
fin:
  // suite
```

```
if (j<g) { goto suivant; }
  r=4;
```

Ce code C correspond assez bien au code assembleur produit par le compilateur.

```
       cmpl    $0, j      ; j==0 ?
       jne     .LBB2_2    ; jump si j!=0
       movl    $1, r      ; r=1
.LBB2_2:

       movl    j, %eax    ; %eax=j
       cmpl    g, %eax    ; j<=g ?
       jle     .LBB2_4    ; jump si j<=g

       movl    $2, r      ; r=2
       jmp     .LBB2_5    ; jump fin expression
.LBB2_4:
       movl    $3, r      ; r=3
.LBB2_5:

       movl    j, %eax    ; %eax=j
       cmpl    g, %eax    ; j<g ?
       jl      .LBB2_7    ; jump si j<g
       movl    $4, r      ; r=4
.LBB2_7:
```

Les instructions de saut conditionnel interviennent également dans l'implémentation des boucles. Plusieurs types de boucles existent en langage C. Considérons tout d'abord une boucle `while`.

```
while(j>0)
{
  j=j-3;
}
```

Cette boucle peut se réécrire en utilisant des `goto` comme suit.

```
debut:
  if(j<=0) { goto fin; }
  j=j-3;
  goto debut;
fin:
```

On retrouve cette utilisation des instructions de saut dans la traduction en assembleur de cette boucle.

```
.LBB3_1:
      cmpl    $0, j    ; j<=0
      jle     .LBB3_3  ; jump si j<=0
      movl    j, %eax
      subl    $3, %eax
      movl    %eax, j  ; j=j-3
      jmp     .LBB3_1
.LBB3_3:
```

Les boucles `for` s'implémentent également en utilisant des instructions de saut.

```
for(j=0;j<10;j++)
  { g=g+h; }

for(j=9;j>0;j=j-1)
  { g=g-h; }
```

La première boucle démarre par l'initialisation de la variable `j` à `0`. Ensuite, la valeur de cette variable est comparée avec `10`. L'instruction `jge` fait un saut à l'adresse mémoire correspondant à l'étiquette `.LBB4_4` si la comparaison indique que `j>=10`. Sinon, les instructions suivantes calculent `g=g+h` et `j++` puis l'instruction `jmp` relance l'exécution à l'instruction de comparaison qui est stockée à l'adresse de l'étiquette `.LBB4_1`.

```
      movl    $0, j    ; j=0
.LBB4_1:
      cmpl    $10, j
      jge     .LBB4_4  ; jump si j>=10
      movl    g, %eax  ; %eax=g
      addl    h, %eax  ; %eax+=h
      movl    %eax, g  ; %eax=g
      movl    j, %eax  ; %eax=j
      addl    $1, %eax ; %eax++
      movl    %eax, j  ; j=%eax
      jmp     .LBB4_1
.LBB4_4:

      movl    $9, j    ; j=9
.LBB4_5:
      cmpl    $0, j
      jle     .LBB4_8  ; jump si j<=0
      movl    g, %eax
      subl    h, %eax
      movl    %eax, g
      movl    j, %eax  ; %eax=j
      subl    $1, %eax ; %eax--
      movl    %eax, j  ; j=%eax
      jmp     .LBB4_5
.LBB4_8:
```

La seconde boucle est organisée de façon similaire.

3.2.5 Manipulation de la pile

Les instructions `mov` permettent de déplacer de l'information à n'importe quel endroit de la mémoire. A côté de ces instructions de déplacement, il y a des instructions qui sont spécialisées dans la manipulation de la pile. La pile, qui dans un processus Unix est stockée dans les adresses hautes est essentielle au bon fonctionnement des programmes. Par convention dans l'architecture [IA32], l'adresse du sommet de la pile est toujours stockée dans le registre `%esp`. Deux instructions spéciales permettent de rajouter et de retirer une information au sommet de la pile.

- `pushl %reg` : place le contenu du registre `%reg` au sommet de la pile et décrémente dans le registre `%esp` l'adresse du sommet de la pile de 4 unités.
- `popl %reg` : retire le mot de 32 bits se trouvant au sommet de la pile, le sauvegarde dans le registre `%reg` et incrémente dans le registre `%esp` l'adresse du sommet de la pile de 4 unités.

En pratique, ces deux instructions peuvent également s'écrire en utilisant des instructions de déplacement et des instructions arithmétiques. Ainsi, `pushl %ebx` est équivalent à :

```
subl $4, %esp       ; ajoute un bloc de 32 bits au sommet de la pile
movl %ebx, (%esp)   ; sauvegarde le contenu de %ebx au sommet
```

Tandis que `popl %ecx` est équivalent à :

```
movl (%esp), %ecx ; sauve dans %ecx la donnée au sommet de la pile
addl $4, %esp     ; déplace le sommet de la pile de 4 unites vers le haut
```

Pour bien comprendre le fonctionnement de la pile, il est utile de considérer un exemple simple. Imaginons la mémoire ci-dessous et supposons qu'initialement le registre `%esp` contient la valeur `0x0C` et que les registres `eax` et `%ebx` contiennent les valeurs `0x02` et `0xFF`.

Adresse	Valeur
0x10	0x04
0x0C	0x00
0x08	0x00
0x04	0x00
0x00	0x00

```
push %eax ; %esp contient 0x08 et M[0x08]=0x02
push %ebx ; %esp contient 0x04 et M[0x04]=0xFF
pop %eax  ; %esp contient 0x08 et %eax 0xFF
pop %ebx  ; %esp contient 0x0C et %ebx 0x02
pop %eax  ; %esp contient 0x10 et %eax 0x00
```

3.2.6 Les fonctions et procédures

Les fonctions et les procédures sont essentielles dans tout langage de programmation. Une procédure est une fonction qui ne retourne pas de résultat. Nous commençons par expliquer comment les procédures peuvent être implémentées en assembleur et nous verrons ensuite comment implémenter les fonctions.

Une procédure est un ensemble d'instructions qui peuvent être appelées depuis n'importe quel endroit du programme. Généralement, une procédure est appelée depuis plusieurs endroits différents d'un programme. Pour comprendre l'implémentation des procédures, nous allons considérer des procédures de complexité croissante. Nos premières procédures ne prennent aucun argument. En C, elles peuvent s'écrire sous la forme de fonctions `void` comme suit.

```
int g=0;
int h=2;

void increase() {
 g=g+h;
}

void init_g() {
 g=1252;
}

int main(int argc, char *argv[]) {
  init_g();
  increase();
  return(EXIT_SUCCESS);
}
```

Ces deux procédures utilisent et modifient des variables globales. Nous verrons plus tard comment supporter les variables locales. Lorsque la fonction `main` appelle la procédure `init_g()` ou la procédure `increase`, il y a plusieurs opérations qui doivent être effectuées. Tout d'abord, le processeur doit transférer l'exécution du code à la première instruction de la procédure appelée. Cela se fait en associant une étiquette à chaque procédure qui correspond à l'adresse de la première instruction de cette procédure en mémoire. Une instruction de saut telle que `jmp` pourrait permettre de démarrer l'exécution de la procédure. Malheureusement, ce n'est pas suffisant car après son exécution la procédure doit pouvoir poursuivre son exécution à l'adresse de l'instruction qui suit celle d'où elle a été appelée. Pour cela, il est nécessaire que la procédure qui a été appelée puisse connaître l'adresse de l'instruction qui doit être exécutée à la fin de son exécution. Dans l'architecture [IA32], cela se fait en utilisant la pile. Vu l'importance des appels de procédure et de fonctions, l'architecture [IA32] contient deux instructions dédiés pour implémenter ces appels. L'instruction `call` est une instruction de saut qui transfère l'exécution à l'adresse de l'étiquette passée en argument et en plus elle sauvegarde au sommet de la pile l'adresse de l'instruction qui la suit. Cette adresse est l'adresse à laquelle la procédure doit revenir après son exécution. Elle est équivalente à une instruction `push` suivie d'une instruction `jmp`. L'instruction `ret` est également une instruction de saut. Elle suppose que l'adresse de retour se trouve au sommet de la pile, retire cette adresse de la pile et fait un saut à cette adresse. Elle est donc équivalente à une instruction `pop` suivie d'une instruction `jmp`. Dans l'architecture [IA32], le registre `%esp` contient en permanence le sommet de la pile. Les instructions `call` et `ret` modifient donc la valeur de ce registre lorsqu'elles sont exécutées. En assembleur, le programme ci-dessus se traduit comme suit :

```
increase:                       ; étiquette de la première instruction
            movl      g, %eax
            addl      h, %eax
            movl      %eax, g
            ret                 ; retour à l'endroit qui suit l'appel
init_g:                         ; étiquette de la première instruction
            movl      $1252, g
            ret                 ; retour à l'endroit qui suit l'appel
main:
            subl      $12, %esp
            movl      20(%esp), %eax
            movl      16(%esp), %ecx
            movl      $0, 8(%esp)
            movl      %ecx, 4(%esp)
            movl      %eax, (%esp)
            calll     init_g    ; appel à la procédure init_g
A_init_g:   calll     increase  ; appel à la procédure increase
A_increase: movl      $0, %eax
            addl      $12, %esp
            ret                 ; fin de la fonction main
g:                              ; étiquette, variable globale g
      .long   0                 ; initialisée à 0
h:                              ; étiquette, variable globale g
      .long   2                 ; initialisée à 2
```

Dans ce code assembleur, on retrouve dans le bas du code la déclaration des deux variables globales, `g` et `h` et leurs valeurs initiales. Chaque procédure a son étiquette qui correspond à l'adresse de sa première instruction. La fonction `main` débute par une manipulation de la pile qui ne nous intéresse pas pour le moment. L'appel à la procédure `init_g()` se fait via l'instruction `calll init_g` qui place sur la pile l'adresse de l'étiquette `A_init_g`. La procédure `init_g()` est très simple puisqu'elle comporte une instruction `movl` qui permet d'initialiser la variable `g` suivie d'une instruction `ret`. Celle-ci retire de la pile l'adresse `A_init_g` qui y avait été placée par l'instruction `call` et poursuit l'exécution du programme à cette adresse. L'appel à la procédure `increase` se déroule de façon similaire.

Considérons une petite variante de notre programme C dans lequel une procédure `p` appelle une procédure `q`.

```
int g=0;
int h=2;

void q() {
 g=1252;
}

void p() {
  q();
  g=g+h;
}



int main(int argc, char *argv[]) {
  p();
  return(EXIT_SUCCESS);
}
```

La compilation de ce programme produit le code assembleur suivant pour les procédures `p` et `q`.

```
q:
        movl      $1252, g
        ret                   ; retour à l'appelant
p:
        subl      $12, %esp   ; réservation d'espace sur pile
        calll     q           ; appel à la procédure q
```

```
        movl    g, %eax
        addl    h, %eax
        movl    %eax, g
        addl    $12, %esp    ; libération espace réservé sur pile
        ret                  ; retour à l'appelant
```

La seule différence par rapport au programme précédent est que la procédure `p` descend le sommet de la pile de 12 unités au début de son exécution et l'augmente de 12 unités à la fin. Ces manipulations sont nécessaires pour respecter une convention de l'architecture [IA32] qui veut que les adresses de retour des procédures soient alignées sur des blocs de 16 bytes.

Considérons maintenant une procédure qui prend un argument. Pour qu'une telle procédure puisse utiliser un argument, il faut que la procédure appelante puisse placer sa valeur à un endroit où la procédure appelée peut facilement y accéder. Dans l'architecture [IA32], c'est la pile qui joue ce rôle et permet le passage des arguments. En C, les arguments sont passés par valeur et ce sera donc les valeurs des arguments qui seront placées sur la pile. A titre d'exemple, considérons une procédure simple qui prend deux arguments entiers.

```
int g=0;
int h=2;
void init(int i, int j) {
 g=i;
 h=j;
}

int main(int argc, char *argv[]) {
  init(1252,1);
  return(EXIT_SUCCESS);
}
```

Le passage des arguments de la fonction `init` depuis la fonction `main` se fait en les plaçant sur la pile avec les instructions `movl $1252, (%esp)` et `movl $1, 4(%esp)` qui précèdent l'instruction `call init`. Le premier argument est placé au sommet de la pile et le second juste au-dessus. La fonction `main` sauvegarde d'autres registres sur la pile avant l'appel à `init`. Ces sauvegardes sont nécessaires car la fonction `main` ne sait pas quels registres seront modifiés par la fonction qu'elle appelle. En pratique l'architecture [IA32] définit des conventions d'utilisation des registres. Les registres `%eax`, `%edx` et `%ecx` sont des registres qui sont sous la responsabilité de la procédure appellante (dans ce cas `main`). Une procédure appelée (dans ce cas-ci `init`) peut modifier sans restrictions les valeurs de ces registres. Si la fonction appellante souhaite pouvoir utiliser les valeurs stockées dans ces registres après l'appel à la procédure, elle doit les sauvegarder elle-même sur la pile. C'est ce que fait la fonction `main` pour `%eax`, `%edx` et `%ecx`. Inversement, les registres `%ebx`, `%edi` et `%esi` sont des registres qui doivent être sauvés par la procédure appelée si celle-ci les modifie. La procédure `init` n'utilisant pas ces registres, elle ne les sauvegarde pas. Par contre, la fonction `main` débute en sauvegardant le registre `%esi` sur la pile.

```
init:
      subl    $8, %esp          ; réservation d'espace sur la pile
      movl    16(%esp), %eax    ; récupération du second argument
      movl    12(%esp), %ecx    ; récupération du premier argument
      movl    %ecx, 4(%esp)     ; sauvegarde sur la pile
      movl    %eax, (%esp)      ; sauvegarde sur la pile
      movl    4(%esp), %eax     ; chargement de i
      movl    %eax, g           ; g=i
      movl    (%esp), %eax      ; chargement de j
      movl    %eax, h           ; h=j
      addl    $8, %esp          ; libération de l'espace réservé
      ret
main:
      pushl   %esi
      subl    $40, %esp
      movl    52(%esp), %eax
      movl    48(%esp), %ecx
      movl    $1252, %edx
      movl    $1, %esi
```

```
    movl    $0, 36(%esp)
    movl    %ecx, 32(%esp)
    movl    %eax, 24(%esp)
    movl    $1252, (%esp)     ; premier argument sur la pile
    movl    $1, 4(%esp)       ; deuxième argument sur la pile
    movl    %esi, 20(%esp)
    movl    %edx, 16(%esp)
    calll   init              ; appel à init
    movl    $0, %eax
    addl    $40, %esp
    popl    %esi
    ret
```

La différence entre une procédure et une fonction est qu'une fonction retourne un résultat. Considérons le programme suivant et les fonctions triviales `int init()` et `int sum(int, int)`. Pour que de telles fonctions puissent s'exécuter et retourner un résultat, il faut que la procédure appelante puisse savoir où aller chercher le résultat après exécution de l'instruction `ret`.

```
int g=0;
int h=2;

int init() {
  return 1252;
}

int sum(int a, int b) {
  return a+b;
}

int main(int argc, char *argv[]) {
  g=init();
  h=sum(1,2);
  return(EXIT_SUCCESS);
}
```

La compilation du programme C ci-dessus en assembleur produit le code suivant. Dans l'architecture [IA32], la valeur de retour d'une fonction est stockée par convention dans le registre `%eax`. Cette convention est particulièrement visible lorsque l'on regarde les instructions générées pour la fonction `int init()`. La fonction `sum` retourne également son résultat dans le registre `%eax`.

```
init:
    movl    $1252, %eax
    ret
sum:
    subl    $8, %esp          ; réservation d'espace sur la pile
    movl    16(%esp), %eax    ; récupération du second argument
    movl    12(%esp), %ecx    ; récupération du premier argument
    movl    %ecx, 4(%esp)
    movl    %eax, (%esp)
    movl    4(%esp), %eax     ; %eax=a
    addl    (%esp), %eax      ; %eax=a+b
    addl    $8, %esp          ; libération de l'espace réservé
    ret
main:
    subl    $28, %esp
    movl    36(%esp), %eax
    movl    32(%esp), %ecx
    movl    $0, 24(%esp)
    movl    %ecx, 20(%esp)    ; sauvegarde sur la pile
    movl    %eax, 16(%esp)    ; sauvegarde sur la pile
    calll   init
    movl    $1, %ecx
    movl    $2, %edx
```

```
    movl    %eax, g
    movl    $1, (%esp)          ; premier argument
    movl    $2, 4(%esp)         ; second argument
    movl    %ecx, 12(%esp)      ; sauvegarde sur la pile
    movl    %edx, 8(%esp)       ; sauvegarde sur la pile
    calll   sum
    movl    $0, %ecx
    movl    %eax, h
    movl    %ecx, %eax
    addl    $28, %esp
    ret
```

Pour terminer notre exploration de la compilation de fonctions C en assembleur, considérons une fonction récursive. Par simplicité, nous utilisons la fonction `sumn` qui calcule de façon récursive la somme des n premiers entiers.

```
int sumn(int n) {
  if(n<=1)
    return n;
  else
    return n+sumn(n-1);
}
```

Lorsque cette fonction récursive est compilée, on obtient le code ci-dessous. Celui-ci démarre par réserver une zone de 28 bytes sur la pile et récupère ensuite l'argument qui est placé dans le registre `%eax`. Cet argument est utilisé comme variable locale, il est donc sauvegardé sur la pile de la fonction `sumn` dans la zone qui vient d'être réservée. Ensuite, on compare la valeur de l'argument avec `1`. Si l'argument est inférieur ou égal à `1`, on récupère la variable locale sur la pile et on la sauve à un autre endroit en préparation à la fin du code (étiquette `.LBB1_3`) ou elle sera placée dans le registre `%eax` avant l'exécution de l'instruction `ret`. Sinon, l'appel récursif est effectué. Pour cela, il faut d'abord calculer `n-1`. Cette valeur est stockée dans le registre `%ecx` puis placée sur la pile avant l'appel récursif. Comme un appel de fonction ne préserve pas `%eax` et que cette valeur est nécessaire après l'appel récursif, elle est sauvegardée sur la pile. La première instruction qui suit l'exécution de l'appel récursif récupère la valeur de la variable `n` sur la pile et la place dans le registre `%ecx`. Le résultat de l'appel récursif étant placé dans `%eax`, l'instruction `addl %ecx, %eax` calcule bien `n+sum(n-1)`. Ce résultat est placé sur la pile puis récupéré et placé dans `%eax` avant l'exécution de `ret`. Il faut noter que les 28 bytes qui avaient étés ajoutés à la pile au début de la fonction sont retirées par l'instruction `addl $28, %esp`. C'est nécessaire pour que la pile soit bien préservée lors de l'appel à une fonction.

```
sumn:
     subl    $28, %esp          ; réservation d'espace sur la pile
     movl    32(%esp), %eax     ; récupération argument
     movl    %eax, 20(%esp)     ; sauvegarde sur pile
     cmpl    $1, 20(%esp)
     jg      .LBB1_2            ; jump si n>1
     movl    20(%esp), %eax     ; récupération n
     movl    %eax, 24(%esp)
     jmp     .LBB1_3
.LBB1_2:
     movl    20(%esp), %eax
     movl    20(%esp), %ecx
     subl    $1, %ecx           ; %ecx=n-1
     movl    %ecx, (%esp)       ; argument sur pile
     movl    %eax, 16(%esp)
recursion:
     calll   sumn
     movl    16(%esp), %ecx     ; récupération de n
     addl    %ecx, %eax         ; %eax=%eax+n
     movl    %eax, 24(%esp)
.LBB1_3:
     movl    24(%esp), %eax
     addl    $28, %esp              ; libération de l'espace réservé sur la pile
     ret
```

Ce code illustre la complexité de supporter des appels récursifs en C et le coût au niveau de la gestion de la pile notamment. Ces appels récursifs doivent être réservés à des fonctions où l'appel récursif apporte une plus value claire.

CHAPITRE 4

Systèmes Multiprocesseurs

4.1 Utilisation de plusieurs threads

Les performances des microprocesseurs se sont continuellement améliorées depuis les années 1960s. Cette amélioration a été possible grâce aux progrès constants de la microélectronique qui a permis d'assembler des microprocesseurs contenant de plus en plus de transistors sur une surface de plus en plus réduite. La figure [1] ci-dessous illustre bien cette évolution puisqu'elle représente le nombre transistors par microprocesseur en fonction du temps.

Cette évolution avait été prédite par Gordon Moore dans les années 1960s [Stokes2008]. Il a formulé en 1965 une hypothèse qui prédisait que le nombre de composants par chip continuerait à doubler tous les douze mois pour la prochaine décennie. Cette prédiction s'est avérée tout à fait réaliste. Elle est maintenant connue sous le nom de *Loi de Moore* et est fréquemment utilisée pour expliquer les améliorations de performance des ordinateurs.

Le fonctionnement d'un microprocesseur est régulé par une horloge. Celle-ci rythme la plupart des opérations du processeur et notamment le chargement des instructions depuis la mémoire. Pendant de nombreuses années, les performances des microprocesseurs ont fortement dépendu de leur vitesse d'horloge. Les premiers microprocesseurs avaient des fréquences d'horloge de quelques centaines de *kHz*. A titre d'exemple, le processeur intel 4004 avait une horloge à 740 kHz en 1971. Aujourd'hui, les processeurs rapides dépassent la fréquence de 3 *GHz*. La figure ci-dessous présente l'évolution de la fréquence d'horloge des microprocesseurs depuis les années 1970s [2]. On remarque une évolution rapide jusqu'aux environs du milieu de la dernière décennie. La barrière des 10 MHz a été franchie à la fin des années 1970s. Les 100 *MHz* ont étés atteints en 1994 et le Ghz aux environs de l'an 2000.

Pendant près de quarante ans, l'évolution technologique a permis une amélioration continue des performances des microprocesseurs. Cette amélioration a directement profité aux applications informatiques car elles ont pu s'exécuter plus rapidement au fur et à mesure que la vitesse d'horloge des microprocesseurs augmentait.

Malheureusement, vers 2005 cette croissance continue s'est arrêtée. La barrière des 3 GHz s'est avérée être une barrière très couteuse à franchir d'un point de vue technologique. Aujourd'hui, les fabricants de microprocesseurs n'envisagent plus de chercher à continuer à augmenter les fréquences d'horloge des microprocesseurs.

Si pendant longtemps la fréquence d'horloge d'un microprocesseur a été une bonne heuristique pour prédire les performances du microprocesseur, ce n'est pas un indicateur parfait de performance. Certains processeurs exécutent une instruction durant chaque cycle d'horloge. D'autres processeurs prennent quelques cycles d'horloge pour exécuter chaque instruction et enfin certains processeurs sont capables d'exécuter plusieurs instructions durant chaque cycle d'horloge.

Une autre façon de mesurer les performances d'un microprocesseur est de comptabiliser le nombre d'instructions qu'il exécute par seconde. On parle en général de Millions d'Instructions par Seconde (ou *MIPS*). Si les premiers microprocesseurs effectuaient moins de 100.000 instructions par seconde, la barrière du MIPS a été franchie en 1979. Mesurées en MIPS, les performances des microprocesseurs ont continué à augmenter durant les dernières années malgré la barrière des 3 GHz comme le montre la figure ci-dessous.

1. Source : http://en.wikipedia.org/wiki/File:Transistor_Count_and_Moore%27s_Law_-_2011.svg
2. Plusieurs sites web recensent cette information, notamment http://www.intel.com/pressroom/kits/quickreffam.htm, http://en.wikipedia.org/wiki/List_of_Intel_microprocessors et http://en.wikipedia.org/wiki/Instructions_per_second

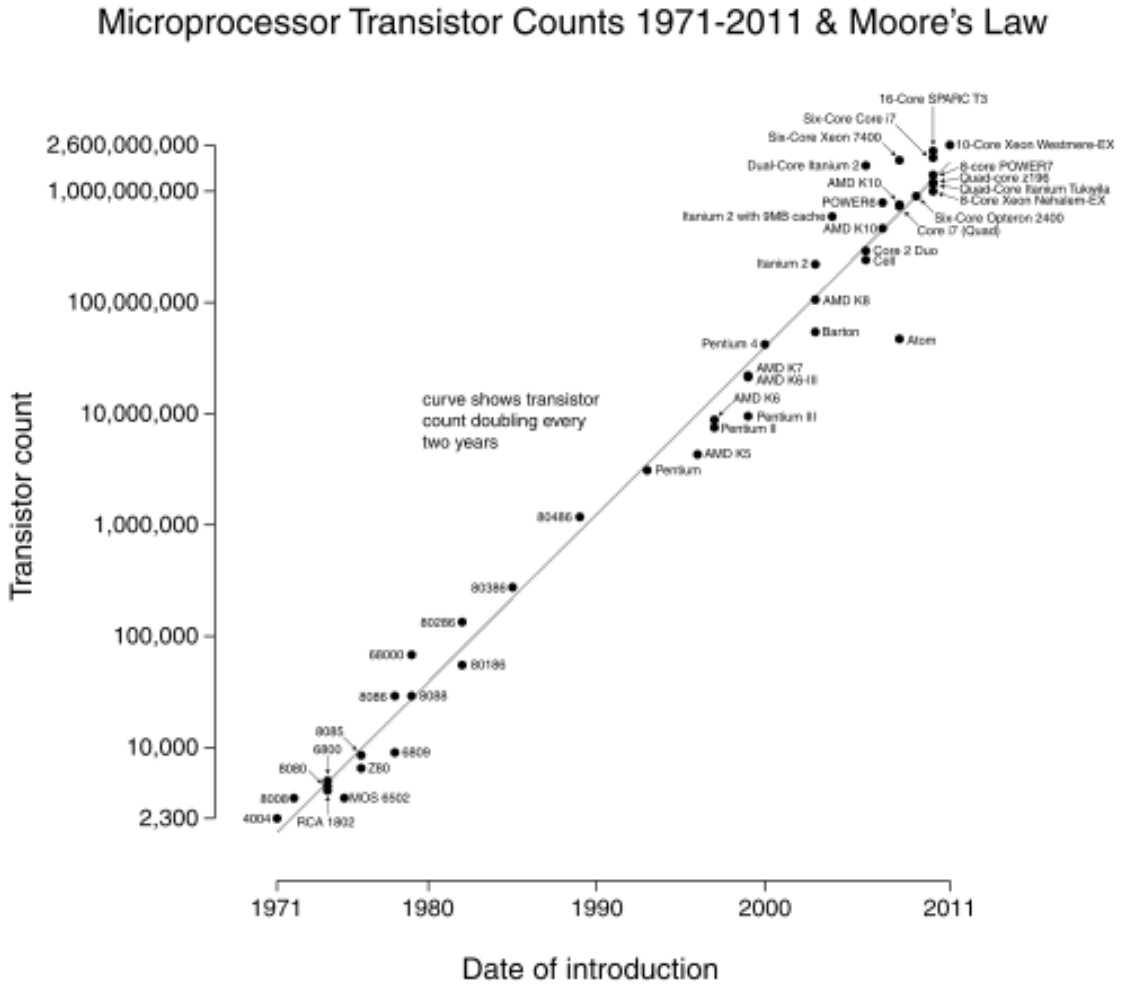


FIGURE 4.1 – Evolution du nombre de transistors par microprocesseur

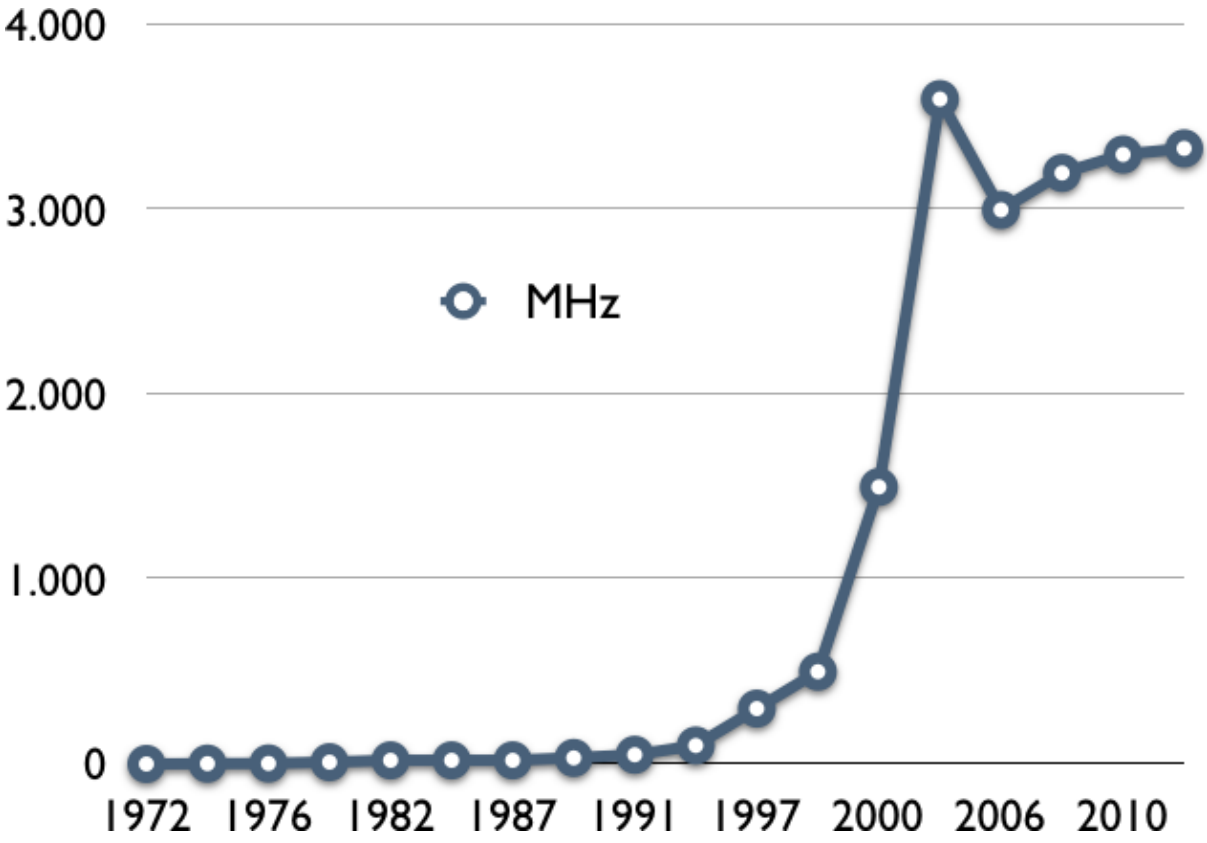


FIGURE 4.2 – Evolution de la vitesse d'horloge des microprocesseurs

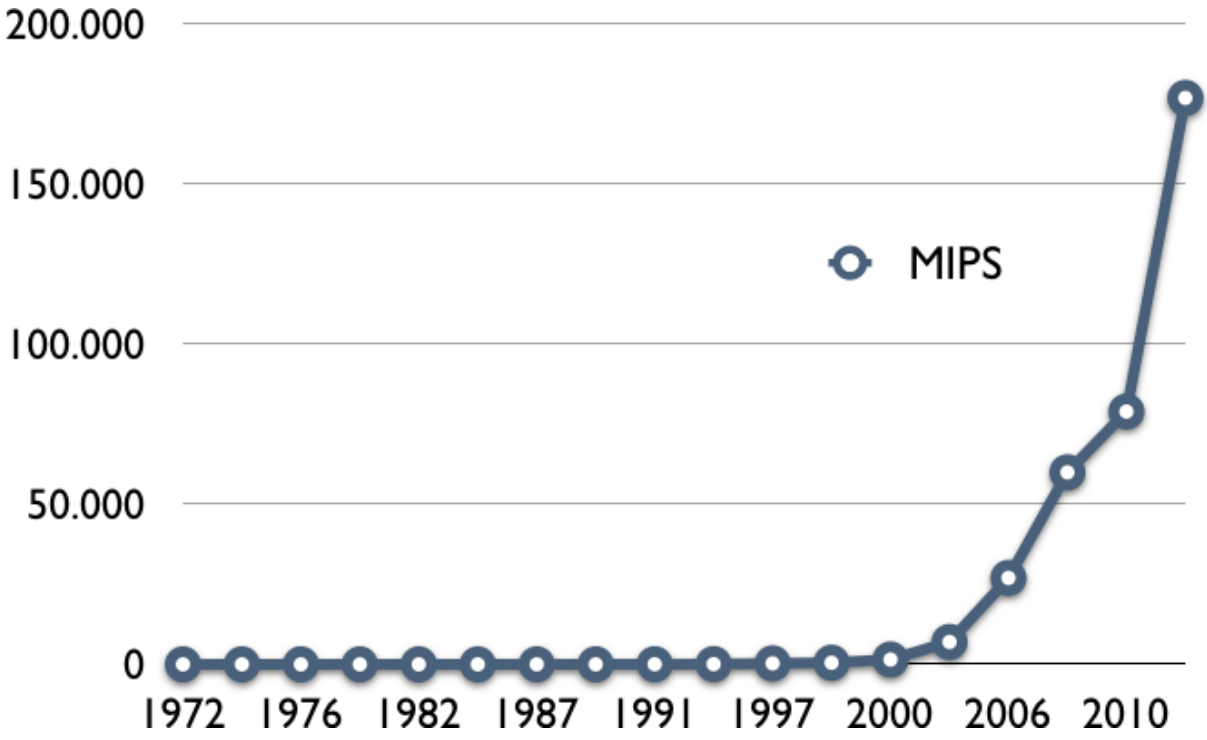


FIGURE 4.3 – Evolution des performances des microprocesseurs en MIPS

Note : Evaluation des performances de systèmes informatiques

La fréquence d'horloge d'un processeur et le nombre d'instructions qu'il est capable d'exécuter chaque seconde ne sont que quelques uns des paramètres qui influencent les performances d'un système informatique qui intègre ce processeur. Les performances globales d'un système informatique dépendent de nombreux autres facteurs comme la capacité de mémoire et ses performances, la vitesse des bus entre les différents composants, les performances des dispositifs de stockage ou des cartes réseaux. Les performances d'un système dépendront aussi fortement du type d'application utilisé. Un serveur web, un serveur de calcul scientifique et un serveur de bases de données n'auront pas les mêmes contraintes en termes de performance. L'évaluation complète des performances d'un système informatique se fait généralement en utilisant des benchmarks. Un *benchmark* est un ensemble de logiciels qui reproduisent le comportement de certaines classes d'applications de façon à pouvoir tester les performances de systèmes informatiques de façon reproductibles. Différents organismes publient de tels benchmarks. Le plus connu est probablement Standard Performance Evaluation Corporation qui publie des benchmarks et des résultats de benchmarks pour différents types de systèmes informatiques et d'applications.

Cette progression continue des performances en MIPS a été possible grâce à l'introduction de processeurs qui sont capables d'exécuter plusieurs threads d'exécution simultanément. On parle alors de processeur *multi-coeurs* ou *multi-threadé*.

La notion de thread d'exécution est très importante dans un système informatique. Elle permet non seulement de comprendre comme un ordinateur équipé d'un seul microprocesseur peut exécuter plusieurs programmes simultanément, mais aussi comment des programmes peuvent profiter des nouveaux processeurs capables d'exécuter plusieurs threads simultanément. Pour comprendre cette notion, il est intéressant de revenir à nouveau sur l'exécution d'une fonction en langage assembleur. Considérons la fonction `f` :

```
int f(int a, int b ) {
  int m=0;
  int c=0;
  while(c<b) {
     m+=a;
     c=c+1;
  }
  return m;
}
```

En assembleur, cette fonction se traduit en :

```
f:
     subl    $16, %esp
     movl    24(%esp), %eax
     movl    20(%esp), %ecx
     movl    %ecx, 12(%esp)
     movl    %eax, 8(%esp)
     movl    $0, 4(%esp)
     movl    $0, (%esp)
.LBB0_1:
     movl    (%esp), %eax
     cmpl    8(%esp), %eax
     jge     .LBB0_3

     movl    12(%esp), %eax
     movl    4(%esp), %ecx
     addl    %eax, %ecx
     movl    %ecx, 4(%esp)
     movl    (%esp), %eax
     addl    $1, %eax
     movl    %eax, (%esp)
     jmp     .LBB0_1
.LBB0_3:
     movl    4(%esp), %eax
     addl    $16, %esp
     ret
```

Pour qu'un processeur puisse exécuter cette séquence d'instructions, il faut non seulement qu'il implémente chacune de ces instructions, mais également qu'il puisse accéder :

- à la mémoire contenant les instructions à exécuter
- à la mémoire contenant les données manipulées par cette séquence d'instruction. Pour rappel, cette mémoire est divisée en plusieurs parties :
  - la zone contenant les variables globales
  - le tas
  - la pile
- aux registres et plus particulièrement, il doit accéder :
  - aux registres de données pour stocker les résultats de chacune des instructions
  - au registre `%esp` directement ou indirectement via les instructions `push` et `pop` qui permettent de manipuler la pile
  - au registre `%eip` qui contient l'adresse de l'instruction en cours d'exécution
  - au registre `eflags` qui contient l'ensemble des drapeaux

Un processeur multithreadé a la capacité d'exécuter plusieurs programmes simultanément. En pratique, ce processeur disposera de plusieurs copies des registres. Chacun de ces blocs de registres pourra être utilisé pour exécuter ces programmes simultanément à raison d'un thread d'exécution par bloc de registres. Chaque thread d'exécution va correspondre à une séquence différente d'instructions qui va modifier son propre bloc de registres. C'est grâce à cette capacité d'exécuter plusieurs threads d'exécution simultanément que les performances en *MIPS* des microprocesseurs ont pu continuer à croitre alors que leur fréquence d'horloge stagnait.

Cette capacité d'exécuter plusieurs threads d'exécution simultanément n'est pas limitée à un thread d'exécution par programme. Sachant qu'un thread d'exécution n'est finalement qu'une séquence d'instructions qui utilisent un bloc de registres, il est tout à fait possible à plusieurs séquences d'exécution appartenant à un même programme de s'exécuter simultanément. Si on revient à la fonction assembleur ci-dessus, il est tout à fait possible que deux invocations de cette fonction s'exécutent simultanément sur un microprocesseur. Pour démarrer une telle instance, il suffit de pouvoir initialiser le bloc de registres nécessaire à la nouvelle instance et ensuite de démarrer l'exécution à la première instruction de la fonction. En pratique, cela nécessite la coopération du système d'exploitation. Différents mécanismes ont été proposés pour permettre à un programme de lancer différents threads d'exécution. Aujourd'hui, le plus courant est connu sous le nom de threads POSIX. C'est celui que nous allons étudier en détail, mais il en existe d'autres.

Note : D'autres types de threads

A côté des threads POSIX, il existe d'autres types de threads. [Gove2011] présente l'implémentation des threads sur différents systèmes d'exploitation. Sous Linux, NTPL [DrepperMolnar2005] et LinuxThreads [Leroy] sont deux anciennes implémentations des threads POSIX. GNU PTH [GNUPTH] est une librairie qui implémente les threads sans interaction directe avec le système d'exploitation. Cela permet à la librairie d'être portable sur de nombreux systèmes d'exploitation. Malheureusement, tous les threads GNU PTH d'un programme doivent s'exécuter sur le même processeur.

4.1.1 Les threads POSIX

Les threads POSIX sont supportés par la plupart des variantes de Unix. Ils sont souvent implémentés à l'intérieur d'une librairie. Sous Linux, il s'agit de la librairie pthreads(7) qui doit être explicitement compilée avec le paramètre `-lpthread` lorsque l'on utilise gcc(1).

La librairie threads POSIX contient de nombreuses fonctions qui permettent de décomposer un programme en plusieurs threads d'exécution et de les gérer. Toutes ces fonctions nécessitent l'inclusion du fichier pthread.h. La première fonction importante est pthread_create(3) qui permet de créer un nouveau thread d'exécution. Cette fonction prend quatre arguments et retourne une valeur entière.

```
#include <pthread.h>

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

Le premier argument est un pointeur vers une structure de type `pthread_t`. Cette structure est définie dans pthread.h et contient toutes les informations nécessaires à l'exécution d'un thread. Chaque thread doit disposer de sa structure de données de type `pthread_t` qui lui est propre.

Le second argument permet de spécifier des attributs spécifiques au thread qui est créé. Ces attributs permettent de configurer différents paramètres associés à un thread. Nous y reviendrons ultérieurement. Si cet argument est mis à `NULL`, la librairie `pthreads` utilisera les attributs par défaut qui sont en général largement suffisants.

Le troisième argument contient l'adresse de la fonction par laquelle le nouveau thread va démarrer son exécution. Cette adresse est le point de départ de l'exécution du thread et peut être comparée à la fonction `main` qui est lancée par le système d'exploitation lorsqu'un programme est exécuté. Un thread doit toujours débuter son exécution par une fonction dont la signature est `void * function(void *)`, c'est-à-dire une fonction qui prend comme argument un pointeur générique (de type `void *`) et retourne un résultat du même type.

Le quatrième argument est l'argument qui est passé à la fonction qui débute le thread qui vient d'être créé. Cet argument est un pointeur générique de type `void *`, mais la fonction peut bien entendu le caster dans un autre type.

La fonction pthread_create(3) retourne un résultat entier. Une valeur de retour non-nulle indique une erreur et `errno` est mise à jour.

Un thread s'exécute en général pendant une certaine période de temps puis il peut retourner un résultat au thread d'exécution principal. Un thread peut retourner son résultat (de type `void *`) de deux façons au thread qui l'a lancé. Tout d'abord, un thread qui a démarré par la fonction `f` se termine lorsque cette fonction exécute `return(...)`. L'autre façon de terminer un thread d'exécution est d'appeler explicitement la fonction pthread_exit(3). Celle-ci prend un argument de type `void *` et le retourne au thread qui l'avait lancé.

Pour récupérer le résultat d'un thread d'exécution, le thread principal doit utiliser la fonction pthread_join(3). Celle-ci prend deux arguments et retourne un entier.

```
#include <pthread.h>

int
pthread_join(pthread_t thread, void **value_ptr);
```

Le premier argument de pthread_join(3) est la structure `pthread_t` correspondant au thread dont le résultat est attendu. Le second argument est un pointeur vers un pointeur générique (`void **`) qui après la terminaison du thread passé comme premier argument pointera vers la valeur de retour de ce thread.

L'exemple ci-dessous illustre une utilisation simple des fonctions pthread_create(3), pthread_join(3) et pthread_exit(3).

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int global=0;

void error(int err, char *msg) {
  fprintf(stderr,"%s a retourné %d, message d'erreur : %s\n",msg,err,strerror(errno));
  exit(EXIT_FAILURE);
}

void *thread_first(void * param) {
  global++;
  return(NULL);
}

void *thread_second(void * param) {
  global++;
  pthread_exit(NULL);
}
```

```
int main (int argc, char *argv[])  {
  pthread_t first;
  pthread_t second;
  int err;

  err=pthread_create(&first,NULL,&thread_first,NULL);
  if(err!=0)
    error(err,"pthread_create");

  err=pthread_create(&second,NULL,&thread_second,NULL);
  if(err!=0)
    error(err,"pthread_create");

  for(int i=0; i<1000000000;i++) { /*...*/ }

  err=pthread_join(second,NULL);
    if(err!=0)
      error(err,"pthread_join");

  err=pthread_join(first,NULL);
    if(err!=0)
      error(err,"pthread_join");

  printf("global: %d\n",global);

  return(EXIT_SUCCESS);
}
```

Dans ce programme, la fonction `main` lance deux threads. Le premier exécute la fonction `thread_first` et le second la fonction `thread_second`. Ces deux fonctions incrémentent une variable globale et n'utilisent pas leur argument. `thread_first` se termine par `return` tandis que `thread_second` se termine par un appel à pthread_exit(3). Après avoir créé ses deux threads, la fonction `main` démarre une longue boucle puis appelle `pthread_join` pour attendre la fin des deux threads qu'elle avait lancé.

Afin d'illustrer la possibilité de passer des arguments à un thread et d'en récupérer la valeur de retour, considérons l'exemple ci-dessous.

```
#define NTHREADS 4
void *neg (void * param) {
  int *l;
  l=(int *) param;
  int *r=(int *)malloc(sizeof(int));
  *r=-*l;
  return ((void *) r);
}

int main (int argc, char *argv[])  {
  pthread_t threads[NTHREADS];
  int arg[NTHREADS];
  int err;

  for(long i=0;i<NTHREADS;i++) {
    arg[i]=i;
    err=pthread_create(&(threads[i]),NULL,&neg,(void *) &(arg[i]));
    if(err!=0)
      error(err,"pthread_create");
  }

  for(int i=0;i<NTHREADS;i++) {
    int *r;
    err=pthread_join(threads[i],(void **)&r);
    printf("Resultat[%d]=%d\n",i,*r);
```

```
    free(r);
    if(err!=0)
      error(err,"pthread_join");
  }
  return(EXIT_SUCCESS);
}
```

Ce programme lance 4 threads d'exécution en plus du thread principal. Chaque thread d'exécution exécute la fonction `neg` qui récupère un entier comme argument et retourne l'opposé de cet entier comme résultat.

Lors d'un appel à pthread_create(3), il est important de se rappeler que cette fonction crée le thread d'exécution, mais que ce thread ne s'exécute pas nécessairement immédiatement. En effet, il est très possible que le système d'exploitation ne puisse pas activer directement le nouveau thread d'exécution, par exemple parce que l'ensemble des processeurs de la machine sont actuellement utilisés. Dans ce cas, le thread d'exécution est mis en veille par le système d'exploitation et il sera démarré plus tard. Sachant que le thread peut devoir démarrer plus tard, il est important de s'assurer que la fonction lancée par pthread_create(3) aura bien accès à son argument au moment où finalement elle démarrera. Dans l'exemple ci-dessous, cela se fait en passant comme quatrième argument l'adresse d'un entier casté en `void *`. Cette valeur est copiée sur la pile de la fonction `neg`. Celle-ci pourra accéder à cet entier via ce pointeur sans problème lorsqu'elle démarrera.

Note : Un thread doit pouvoir accéder à son argument

Lorsque l'on démarre un thread via la fonction pthread_create(3), il faut s'assurer que la fonction lancée pourra bien accéder à ses arguments. Ce n'est pas toujours le cas comme le montre l'exemple ci-dessous. Dans cet exemple, c'est l'adresse de la variable locale `i` qui est passée comme quatrième argument à la fonction pthread_create(3). Cette adresse sera copiée sur la pile de la fonction `neg` pour chacun des threads créés. Malheureusement, lorsque la fonction `neg` sera exécutée, elle trouvera sur sa pile l'adresse d'une variable qui risque fort d'avoir été modifiée après l'appel à pthread_create(3) ou pire risque d'avoir disparu car la boucle `for` s'est terminée. Il est très important de bien veiller à ce que le quatrième argument passé à pthread_create(3) existe toujours au moment de l'exécution effective de la fonction qui démarre le thread lancé.

```
/// erroné !
for(long i=0;i<NTHREADS;i++) {
  err=pthread_create(&(threads[i]),NULL,&neg,(void *)&i);
  if(err!=0)
    error(err,"pthread_create");
}
```

Concernant pthread_join(3), le code ci-dessus illustre la récupération du résultat via un pointeur vers un entier. Comme la fonction `neg` retourne un résultat de type `void *` elle doit nécessairement retourner un pointeur qui peut être casté vers un pointeur de type `void *`. C'est ce que la fonction `neg` dans l'exemple réalise. Elle alloue une zone mémoire permettant de stocker un entier et place dans cette zone mémoire la valeur de retour de la fonction. Ce pointeur est ensuite casté en un pointeur de type `void *` avant d'appeler `return`. Il faut noter que l'appel à pthread_join(3) ne se termine que lorsque le thread spécifié comme premier argument se termine. Si ce thread ne se termine pas pour n'importe quelle raison, l'appel à pthread_join(3) ne se terminera pas non plus.

4.2 Communication entre threads

Lorsque un programme a été décomposé en plusieurs threads, ceux-ci ne sont en général pas complètement indépendants et ils doivent communiquer entre eux. Cette communication entre threads est un problème complexe comme nous allons le voir. Avant d'aborder ce problème, il est utile de revenir à l'organisation d'un processus et de ses threads en mémoire. La figure ci-dessous illustre schématiquement l'organisation de la mémoire après la création d'un thread POSIX.

Le programme principal et le thread qu'il a créé partagent trois zones de la mémoire : le *segment text* qui comprend l'ensemble des instructions qui composent le programme, le *segment de données* qui comprend toutes les données statiques, initialisées ou non et enfin le *heap*. Autant le programme principal que son thread peuvent accéder à n'importe quelle information se trouvant en mémoire dans ces zones. Par contre, le programme principal et le thread qu'il vient de créer ont chacun leur propre contexte et leur propre pile.

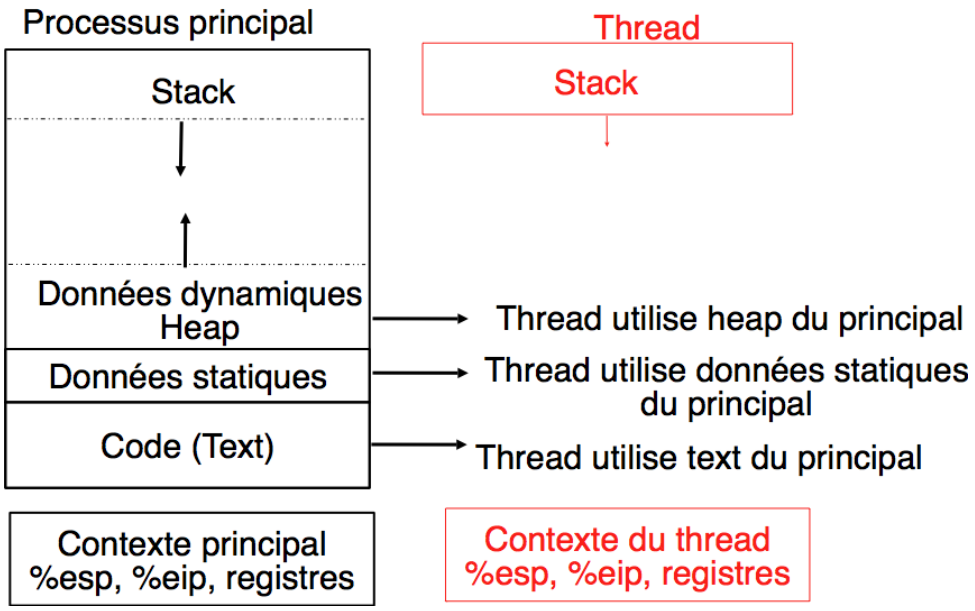


FIGURE 4.4 – Organisation de la mémoire après la création d'un thread POSIX

La première façon pour un processus de communiquer avec un thread qu'il a lancé est d'utiliser les arguments de la fonction de démarrage du thread et la valeur retournée par le thread que le processus principal peut récupérer via l'appel à pthread_join(3posix). C'est un canal de communication très limité qui ne permet pas d'échange d'information pendant l'exécution du thread.

Il est cependant assez facile pour un processus de partager de l'information avec ses threads ou même de partager de l'information entre plusieurs threads. En effet, tous les threads d'un processus ont accès aux mêmes variables globales et au même *heap*. Il est donc tout à fait possible pour n'importe quel thread de modifier la valeur d'une variable globale. Deux threads qui réalisent un calcul peuvent donc stocker des résultats intermédiaires dans une variable globale ou un tableau global. Il en va de même pour l'utilisation d'une zone de mémoire allouée par malloc(3). Chaque thread qui dispose d'un pointeur vers cette zone mémoire peut en lire le contenu ou en modifier la valeur.

Malheureusement, permettre à tous les threads de lire et d'écrire simultanément en mémoire peut être une source de problèmes. C'est une des difficultés majeures de l'utilisation de threads. Pour s'en convaincre, considérons l'exemple ci-dessous [3].

```
long global=0;
int increment(int i) {
  return i+1;
}
void *func(void * param) {
  for(int j=0;j<1000000;j++) {
    global=increment(global);
  }
  return(NULL);
}
```

Dans cet exemple, la variable `global` est incrémentée `1000000` de fois par la fonction `func`. Après l'exécution de cette fonction, la variable `global` contient la valeur `1000000`. Sur une machine multiprocesseurs, un programmeur pourrait vouloir en accélérer les performances en lançant plusieurs threads qui exécutent chacun la fonction `func`. Cela pourrait se faire en utilisant par exemple la fonction `main` ci-dessous.

```
int main (int argc, char *argv[])  {
  pthread_t thread[NTHREADS];
  int err;
  for(int i=0;i<NTHREADS;i++) {
```

3. Le programme complet est accessible via `/Threads/S5-src/pthread-test.c`

```
    err=pthread_create(&(thread[i]),NULL,&func,NULL);
    if(err!=0)
      error(err,"pthread_create");
  }
  /*...*/
  for(int i=NTHREADS-1;i>=0;i--) {
    err=pthread_join(thread[i],NULL);
    if(err!=0)
      error(err,"pthread_join");
  }
  printf("global: %ld\n",global);
  return(EXIT_SUCCESS);
}
```

Ce programme lance alors 4 threads d'exécution qui incrémentent chacun un million de fois la variable `global`. Celle-ci étant initialisée à `0`, la valeur affichée par printf(3) à la fin de l'exécution doit donc être `4000000`. L'exécution du programme ne confirme malheureusement pas cette attente.

```
$ for i in {1..5}; do ./pthread-test;  done
global: 3408577
global: 3175353
global: 1994419
global: 3051040
global: 2118713
```

Non seulement la valeur attendue (`4000000`) n'est pas atteinte, mais en plus la valeur change d'une exécution du programme à la suivante. C'est une illustration du problème majeur de la découpe d'un programme en threads. Pour bien comprendre le problème, il est utile d'analyser en détails les opérations effectuées par deux threads qui exécutent la ligne `global=increment(global);`.

La variable `global` est stockée dans une zone mémoire qui est accessibles aux deux threads. Appelons-les *T1* et *T2*. L'exécution de cette ligne par un thread nécessite l'exécution de plusieurs instructions en assembleur. Tout d'abord, il faut charger la valeur de la variable `global` depuis la mémoire vers un registre. Ensuite, il faut placer cette valeur sur la pile du thread puis appeler la fonction `increment`. Cette fonction récupère son argument sur la pile du thread, la place dans un registre, incrémente le contenu du registre et sauvegarde le résultat dans le registre `%eax`. Le résultat est retourné dans la fonction `func` et la variable globale peut enfin être mise à jour.

Malheureusement les difficultés surviennent lorsque deux threads exécutent en même temps la ligne `global=increment(global);`. Supposons qu'à cet instant, la valeur de la variable `global` est `1252`. Le premier thread charge une copie de cette variable sur sa pile. Le second fait de même. Les deux threads ont donc chacun passé la valeur `1252` comme argument à la fonction `increment`. Celle-ci s'exécute et retourne la valeur `1253` que chaque thread va récupérer dans `%eax`. Chaque thread va ensuite transférer cette valeur dans la zone mémoire correspondant à la variable `global`. Si les deux threads exécutent l'instruction assembleur correspondante exactement au même moment, les deux écritures en mémoire seront sérialisées par les processeurs sans que l'on ne puisse a priori déterminer quelle écriture se fera en premier [McKenney2005]. Alors que les deux threads ont chacun exécuté un appel à la fonction `increment`, la valeur de la variable n'a finalement été incrémentée qu'une seule fois même si cette valeur a été transférée deux fois en mémoire. Ce problème se reproduit fréquemment. C'est pour cette raison que la valeur de la variable `global` n'est pas modifiée comme attendu.

Note : Contrôler la pile d'un thread POSIX

La taille de la pile d'un thread POSIX est l'un des attributs qui peuvent être modifiés lors de l'appel à pthread_create(3) pour créer un nouveau thread. Cet attribut peut être fixé en utilisant la fonction pthread_attr_setstackaddr(3posix) comme illustré dans l'exemple ci-dessous [4] (où `thread_first` est la fonction qui sera appelée à la création du thread). En général, la valeur par défaut choisie par le système suffit, sauf lorsque le programmeur sait qu'un thread devra par exemple allouer un grand tableau auquel il sera le seul à avoir accès. Ce tableau sera alors alloué sur la pile qui devra être suffisamment grande pour le contenir.

```
pthread_t first;
pthread_attr_t attr_first;
```

4. Le programme complet est accessible via `/Threads/S6-src/pthread.c`

```
  size_t stacksize;

  int err;

  err= pthread_attr_init(&attr_first);
  if(err!=0)
    error(err,"pthread_attr_init");

  err= pthread_attr_getstacksize(&attr_first,&stacksize);
  if(err!=0)
    error(err,"pthread_attr_getstacksize");

  printf("Taille par défaut du stack : %ld\n",stacksize);

  stacksize=65536;

  err= pthread_attr_setstacksize(&attr_first,stacksize);
  if(err!=0)
    error(err,"pthread_attr_setstacksize");

  err=pthread_create(&first,&attr_first,&thread_first,NULL);
  if(err!=0)
    error(err,"pthread_create");
```

Ce problème d'accès concurrent à une zone de mémoire par plusieurs threads est un problème majeur dans le développement de programmes découpés en threads, que ceux-ci s'exécutent sur des ordinateurs monoprocesseurs ou multiprocesseurs. Dans la littérature, il est connu sous le nom de problème de la *section critique* ou *exclusion mutuelle*. La *section critique* peut être définie comme étant une séquence d'instructions qui ne peuvent *jamais* être exécutées par plusieurs threads simultanément. Dans l'exemple ci-dessus, il s'agit de la ligne `global=increment(global);`. Dans d'autres types de programmes, la section critique peut être beaucoup plus grande et comprendre par exemple la mise à jour d'une base de données. En pratique, on retrouvera une section critique chaque fois que deux threads peuvent modifier ou lire la valeur d'une même zone de la mémoire.

Le fragment de code ci-dessus présente une autre illustration d'une section critique. Dans cet exemple, la fonction `main` (non présentée), créée deux threads. Le premier exécute la fonction `inc` qui incrémente la variable `global`. Le second exécute la fonction `is_even` qui teste la valeur de cette variable et compte le nombre de fois qu'elle est paire. Après la terminaison des deux threads, le programme affiche le contenu des variables `global` et `even`.

```
long global=0;
int even=0;
int odd=0;

void test_even(int i) {
  if((i%2)==0)
    even++;
}

int increment(int i) {
  return i+1;
}

void *inc(void * param) {
  for(int j=0;j<1000000;j++) {
    global=increment(global);
  }
  pthread_exit(NULL);
}

void *is_even(void * param) {
  for(int j=0;j<1000000;j++) {
```

```
    test_even(global);
  }
  pthread_exit(NULL);
}
```

L'exécution de ces deux threads donne, sans surprise des résultats qui varient d'une exécution à l'autre.

```
$ for i in {1..5}; do ./pthread-test-if;  done
global: 1000000, even:905140
global: 1000000, even:919756
global: 1000000, even:893058
global: 1000000, even:891266
global: 1000000, even:895043
```

4.3 Coordination entre threads

L'utilisation de plusieurs threads dans un programme fonctionnant sur un seul ou plusieurs processeurs nécessite l'utilisation de mécanismes de coordination entre ces threads. Ces mécanismes ont comme objectif d'éviter que deux threads ne puissent modifier ou tester de façon non coordonnée la même zone de la mémoire.

4.3.1 Exclusion mutuelle

Le premier problème important à résoudre lorsque l'on veut coordonner plusieurs threads d'exécution d'un même processus est celui de l'*exclusion mutuelle*. Ce problème a été initialement proposé par Dijkstra en 1965 [Dijkstra1965]. Il peut être reformulé de la façon suivante pour un programme décomposé en threads.

Considérons un programme décomposé en N threads d'exécution. Supposons également que chaque thread d'exécution est cyclique, c'est-à-dire qu'il exécute régulièrement le même ensemble d'instructions, sans que la durée de ce cycle ne soit fixée ni identique pour les N threads. Chacun de ces threads peut être décomposé en deux parties distinctes. La première est la partie non-critique. C'est un ensemble d'instructions qui peuvent être exécutées par le thread sans nécessiter la moindre coordination avec un autre thread. Plus précisément, tous les threads peuvent exécuter simultanément leur partie non-critique. La seconde partie du thread est appelée sa *section critique*. Il s'agit d'un ensemble d'instructions qui ne peuvent être exécutées que par un seul et unique thread. Le problème de l'*exclusion mutuelle* est de trouver un algorithme qui permet de garantir qu'il n'y aura jamais deux threads qui simultanément exécuteront les instructions de leur section critique.

Cela revient à dire qu'il n'y aura pas de violation de la section critique. Une telle violation pourrait avoir des conséquences catastrophiques sur l'exécution du programme. Cette propriété est une propriété de *sureté* (*safety* en anglais). Dans un programme découpé en threads, une propriété de *sureté* est une propriété qui doit être vérifiée à tout instant de l'exécution du programme.

En outre, une solution au problème de l'*exclusion mutuelle* doit satisfaire les contraintes suivantes [Dijkstra1965] :

1. La solution doit considérer tous les threads de la même façon et ne peut faire aucune hypothèse sur la priorité relative des différents threads.
2. La solution ne peut faire aucune hypothèse sur la vitesse relative ou absolue d'exécution des différents threads. Elle doit rester valide quelle que soit la vitesse d'exécution non nulle de chaque thread.
3. La solution doit permettre à un thread de s'arrêter en dehors de sa section critique sans que cela n'invalide la contrainte d'exclusion mutuelle
4. Si un ou plusieurs threads souhaitent entamer leur section critique, aucun de ces threads ne doit pouvoir être empêché indéfiniment d'accéder à sa section critique.

La troisième contrainte implique que la terminaison ou le crash d'un des threads ne doit pas avoir d'impact sur le fonctionnement du programme complet et le respect de la contrainte d'exclusion mutuelle pour la section critique.

La quatrième contrainte est un peu plus subtile mais tout aussi importante. Toute solution au problème de l'exclusion mutuelle contient nécessairement un mécanisme qui permet de bloquer l'exécution d'un thread pendant qu'un autre exécute sa section critique. Il est important qu'un thread puisse accéder à sa section critique si il le souhaite. C'est un exemple de propriété de *vivacité* (*liveness* en anglais). Une propriété de *vivacité* est une propriété qui ne

peut pas être éternellement invalidée. Dans notre exemple, un thread ne pourra jamais être empêché d'accéder à sa section critique.

Exclusion mutuelle sur monoprocesseurs

Même si les threads sont très utiles dans des ordinateurs multiprocesseurs, ils ont été inventés et utilisés d'abord sur des processeurs capables d'exécuter un seul thread d'exécution à la fois. Sur un tel processeur, les threads d'exécution sont entrelacés plutôt que d'être exécutés réellement simultanément. Cet entrelacement est réalisé par le système d'exploitation.

Les systèmes d'exploitation de la famille Unix permettent d'exécuter plusieurs programmes *en même temps* sur un ordinateur, même si il est équipé d'un processeur qui n'est capable que d'exécuter un thread à la fois. Cette fonctionnalité est souvent appelée le *multitâche* (ou *multitasking* en anglais). Cette exécution simultanée de plusieurs programmes n'est en pratique qu'une illusion puisque le processeur ne peut qu'exécuter qu'une séquence d'instructions à la fois.

Pour donner cette illusion, un système d'exploitation multitâche tel que Unix exécute régulièrement des changements de contexte entre threads. Le *contexte* d'un thread est composé de l'ensemble des contenus des registres qui sont nécessaires à son exécution (y compris le contenu des registres spéciaux tels que `%esp`, `%eip` ou `%eflags`). Ces registres définissent l'état du thread du point de vue du processeur. Pour passer de l'exécution du thread *T1* à l'exécution du thread *T2*, le système d'exploitation doit initier un *changement de contexte*. Pour réaliser ce changement de contexte, le système d'exploitation initie le transfert du contenu des registres utilisés par le thread *T1* vers une zone mémoire lui appartenant. Il transfère ensuite depuis une autre zone mémoire lui appartenant le contexte du thread *T2*. Si ce changement de contexte est effectué fréquemment, il peut donner l'illusion à l'utilisateur que plusieurs threads ou programmes s'exécutent simultanément.

Sur un système Unix, il y a deux types d'événements qui peuvent provoquer un changement de contexte.

1. Le hardware génère une *interruption*
2. Un thread exécute un *appel système bloquant*

Ces deux types d'événements sont fréquents et il est important de bien comprendre comment ils sont traités par le système d'exploitation.

Une *interruption* est un signal électronique qui est généré par un dispositif connecté au microprocesseur. De nombreux dispositifs d'entrées-sorties comme les cartes réseau ou les contrôleurs de disque peuvent générer une interruption lorsqu'une information a été lue ou reçue et doit être traitée par le processeur. En outre, chaque ordinateur dispose d'une horloge temps réel qui génère des interruptions à une fréquence déterminée par le système d'exploitation mais qui est généralement comprise entre quelques dizaines et quelques milliers de *Hz*. Ces interruptions nécessitent un traitement rapide de la part du système d'exploitation. Pour cela, le processeur vérifie, à la fin de l'exécution de *chaque* instruction si un signal d'interruption [5] est présent. Si c'est le cas, le processeur sauvegarde en mémoire le contexte du thread en cours d'exécution et lance une routine de traitement d'interruption faisant partie du système d'exploitation. Cette routine analyse l'interruption présente et lance les fonctions du système d'exploitation nécessaires à son traitement. Dans le cas d'une lecture sur disque, par exemple, la routine de traitement d'interruption permettra d'aller chercher la donnée lue sur le contrôleur de disques.

Le deuxième type d'événement est l'exécution d'un appel système bloquant. Un thread exécute un *appel système* chaque fois qu'il doit interagir avec le système d'exploitation. Ces appels peuvent être exécutés directement ou via une fonction de la librairie [6]. Il existe deux types d'appels systèmes : les appels bloquants et les appels non-bloquants. Un appel système non-bloquant est un appel système que le système d'exploitation peut exécuter immédiatement. Cet appel retourne généralement une valeur qui fait partie du système d'exploitation lui-même. L'appel gettimeofday(2) qui permet de récupérer l'heure actuelle est un exemple d'appel non-bloquant. Un appel système bloquant est un appel système dont le résultat ne peut pas toujours être fourni immédiatement. Les lectures d'information en provenance de l'entrée standard (et donc généralement du clavier) sont un bon exemple simple d'appel système bloquant. Considérons un thread qui exécute la fonction de la librairie getchar(3) qui retourne un caractère lu sur *stdin*. Cette fonction utilise l'appel système read(2) pour lire un caractère sur *stdin*. Bien entendu, le système d'exploitation est obligé d'attendre que l'utilisateur presse une touche sur le clavier pour pouvoir fournir le résultat de cet appel système à l'utilisateur. Pendant tout le temps qui s'écoule entre l'exécution de

5. De nombreux processeurs supportent plusieurs signaux d'interruption différents. Dans le cadre de ce cours, nous nous limiterons à l'utilisation d'un seul signal de ce type.
6. Les appels systèmes sont décrits dans la section 2 des pages de manuel tandis que la section 3 décrit les fonctions de la librairie.

getchar(3) et la pression d'une touche sur le clavier, le thread est bloqué par le système d'exploitation. Plus aucune instruction du thread n'est exécutée tant que la fonction getchar(3) ne s'est pas terminée et le contexte du thread est mis en attente dans une zone mémoire gérée par le système d'exploitation. Il sera redémarré automatiquement par le système d'exploitation lorsque la donnée attendue sera disponible.

Ces interactions entre les threads et le système d'exploitation sont importantes. Pour bien les comprendre, il est utile de noter qu'un thread peut se trouver dans trois états différents du point de vue de son interaction avec le système d'exploitation. Ces trois états sont illustrés dans la figure ci-dessous.

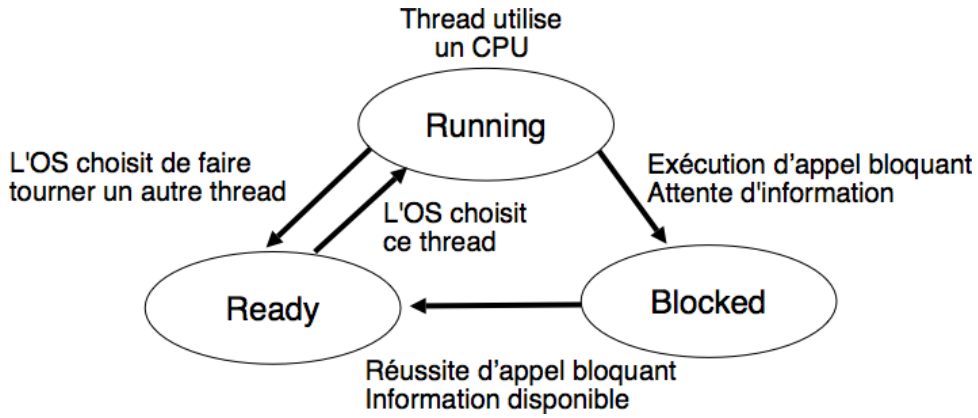


FIGURE 4.5 – Etats d'un thread d'exécution

Lorsqu'un thread est créé avec la fonction pthread_create(3), il est placé dans l'état *Ready*. Dans cet état, les instructions du thread ne s'exécutent sur aucun processeur mais il est prêt à être exécuté dès qu'un processeur se libèrera. Le deuxième état pour un thread est l'état *Running*. Dans cet état, le thread est exécuté sur un des processeurs du système. Le dernier état est l'état *Blocked*. Un thread est dans l'état *Blocked* lorsqu'il a exécuté un appel système bloquant et que le système d'exploitation attend l'information permettant de retourner le résultat de l'appel système. Pendant ce temps, les instructions du thread ne s'exécutent sur aucun processeur.

Les transitions entre les différents états d'un thread sont gérées par le système d'exploitation. Lorsque plusieurs threads d'exécution sont simultanément actifs, le système d'exploitation doit arbitrer les demandes d'utilisation du CPU de chaque thread. Cet arbitrage est réalisé par l'ordonnanceur (ou *scheduler* en anglais). Le *scheduler* est un ensemble d'algorithmes qui sont utilisés par le système d'exploitation pour sélectionner le ou les threads qui peuvent utiliser un processeur à un moment donné. Il y a souvent plus de threads qui sont dans l'état *Ready* que de processeurs disponibles et le scheduler doit déterminer quels sont les threads à exécuter.

Une description détaillée du fonctionnement d'un scheduler relève plutôt d'un cours sur les systèmes d'exploitation que d'un premier cours sur les systèmes informatiques, mais il est important de connaître les principes de base de fonctionnement de quelques schedulers.

Un premier scheduler simple est le *round-robin*. Ce scheduler maintient en permanence une liste circulaire de l'ensemble des threads qui se trouvent dans l'état *Ready* et un pointeur vers l'élément courant de cette liste. Lorsqu'un processeur devient disponible, le scheduler sélectionne le thread référencé par ce pointeur. Ce thread passe dans l'état *Running*, est retiré de la liste et le pointeur est déplacé vers l'élément suivant dans la liste. Pour éviter qu'un thread ne puisse monopoliser éternellement un processeur, un scheduler *round-robin* limite généralement le temps qu'un thread peut passer dans l'état *Running*. Lorsqu'un thread a utilisé un processeur pendant ce temps, le scheduler vérifie si il y a un thread en attente dans l'état *Ready*. Si c'est le cas, le scheduler force un changement de contexte, place le thread courant dans l'état *Ready* et le remet dans la liste circulaire tout en permettant à un nouveau thread de passer dans l'état *Running* pour s'exécuter. Lorsqu'un thread revient dans l'état *Ready*, soit parce qu'il vient d'être créé ou parce qu'il vient de quitter l'état *Blocked*, il est placé dans la liste afin de pouvoir être sélectionné par le scheduler. Un scheduler *round-robin* est équitable. Avec un tel scheduler, si N threads sont actifs en permanence, chacun recevra $\frac{1}{N}$ de temps CPU disponible.

Un second type de scheduler simple est le scheduler à priorités. Une priorité est associée à chaque thread. Lorsque le scheduler doit sélectionner un thread à exécuter, il commence d'abord par parcourir les threads ayant une haute priorité. En pratique, un scheduler à priorité maintiendra une liste circulaire pour chaque niveau de priorité. Lorsque le scheduler est appelé, il sélectionnera toujours le thread ayant la plus haute priorité et se trouvant dans l'état *Ready*. Si plusieurs threads ont le même niveau de priorité, un scheduler de type *round-robin* peut être

utilisé dans chaque niveau de priorité. Sous Unix, le scheduler utilise un scheduler à priorité avec un round-robin à chaque niveau de priorité, mais la priorité varie dynamiquement en fonction du temps de façon à favoriser les threads interactifs.

Connaissant ces bases du fonctionnement des schedulers, il est utile d'analyser en détails quels sont les événements qui peuvent provoquer des transitions entre les états d'un thread. Certains de ces événements sont provoqués par le thread lui-même. C'est le cas de la transition entre l'état *Running* et l'état *Blocked*. Elle se produit lorsque le thread exécute un *appel système bloquant*. Dans ce cas, un processeur redevient disponible et le scheduler peut sélectionner un autre thread pour s'exécuter sur ce processeur. La transition entre l'état *Blocked* et l'état *Running* dépend elle du système d'exploitation, directement lorsque le thread a été bloqué par le système d'exploitation ou indirectement lorsque le système d'exploitation attend une information venant d'un dispositif d'entrées-sorties. Les transitions entre les états *Running* et *Ready* dépendent elles entièrement du système d'exploitation. Elles se produisent lors de l'exécution du scheduler. Celui-ci est exécuté lorsque certaines interruptions surviennent. Il est exécuté à chaque interruption d'horloge. Cela permet de garantir l'exécution régulière du scheduler même si les seuls threads actifs exécutent une boucle infinie telle que `while(true);`. A l'occasion de cette interruption, le scheduler mesure le temps d'exécution de chaque thread et si un thread a consommé beaucoup de temps CPU alors que d'autres threads sont dans l'état *Ready*, le scheduler forcera un changement de contexte pour permettre à un autre thread de s'exécuter. De la même façon, une interruption relative à un dispositif d'entrées-sorties peut faire transiter un thread de l'état *Blocked* à l'état *Ready*. Cette modification du nombre de threads dans l'état *Ready* peut forcer le scheduler à devoir effectuer un changement de contexte pour permettre à ce thread de poursuivre son exécution. Sous Unix, le scheduler utilise des niveaux de priorité qui varient en fonction des opérations d'entrées sorties effectuées. Cela a comme conséquence de favoriser les threads qui effectuent des opérations d'entrées sorties par rapport aux threads qui effectuent uniquement du calcul.

Note : Un thread peut demander de passer la main.

Dans la plupart de nos exemples, les threads cherchent en permanence à exécuter des instructions. Ce n'est pas nécessairement le cas de tous les threads d'un programme. Par exemple, une application de calcul scientifique pourrait être découpée en *N+1* threads. Les *N* premiers threads réalisent le calcul tandis que le dernier calcule des statistiques. Ce dernier thread ne doit pas consommer de ressources et être en compétition pour le processeur avec les autres threads. La librairie thread POSIX contient la fonction pthread_yield(3) qui peut être utilisée par un thread pour indiquer explicitement qu'il peut être remplacé par un autre thread. Si un thread ne doit s'exécuter qu'à intervalles réguliers, il est préférable d'utiliser des appels à sleep(3) ou usleep(3). Ces fonctions de la librarie permettent de demander au système d'exploitation de bloquer le thread pendant un temps au moins égal à l'argument de la fonction.

Sur une machine monoprocesseur, tous les threads s'exécutent sur le même processeur. Une violation de section critique peut se produire lorsque le scheduler décide de réaliser un changement de contexte alors qu'un thread se trouve dans sa section critique. Si la section critique d'un thread ne contient ni d'appel système bloquant ni d'appel à pthread_yield(3), ce changement de contexte ne pourra se produire que si une interruption survient. Une solution pour résoudre le problème de l'exclusion mutuelle sur un ordinateur monoprocesseur pourrait donc être la suivante :

```
disable_interrupts();
// début section critique
// ...
// fin section critique
enable_interrupts();
```

Cette solution est possible, mais elle souffre de plusieurs inconvénients majeurs. Tout d'abord, une désactivation des interruptions perturbe le fonctionnement du système puisque sans interruptions, la plupart des opérations d'entrées-sorties et l'horloge sont inutilisables. Une telle désactivation ne peut être que très courte, par exemple pour modifier une ou quelques variables en mémoire. Ensuite, la désactivation des interruptions, comme d'autres opérations relatives au fonctionnement du matériel, est une opération privilégiée sur un microprocesseur. Elle ne peut être réalisée que par le système d'exploitation. Il faudrait donc imaginer un appel système qui permettrait à un thread de demander au système d'exploitation de désactiver les interruptions. Si un tel appel système existait, le premier programme qui exécuterait `disable_interrupts();` sans le faire suivre de `enable_interrupts();` quelques instants après pourrait rendre la machine complètement inutilisable puisque sans interruption plus aucune opération d'entrée-sortie n'est possible et qu'en plus le scheduler ne peut

plus être activé par l'interruption d'horloge. Pour toutes ces raisons, la désactivation des interruptions n'est pas un mécanisme utilisable par les threads pour résoudre le problème de l'exclusion mutuelle [7].

Algorithme de Peterson

Le problème de l'exclusion mutuelle a intéressé de nombreux informaticiens depuis le début des années 1960s [Dijkstra1965] et différentes solutions à ce problème ont été proposées. Plusieurs d'entre elles sont analysées en détails dans [Alagarsamy2003]. Dans cette section, nous nous concentrerons sur une de ces solutions proposées par G. Peterson en 1981 [Peterson1981]. Cette solution permet à plusieurs threads de coordonner leur exécution de façon à éviter une violation de section critique en utilisant uniquement des variables accessibles à tous les threads. La solution proposée par Peterson permet de gérer *N* threads [Peterson1981] mais nous nous limiterons à sa version permettant de coordonner deux threads.

Une première solution permettant de coordonner deux threads en utilisant des variables partagées pourrait être de s'appuyer sur une variable qui permet de déterminer quel est le thread qui peut entrer en section critique. Dans l'implémentation ci-dessous, la variable partagée `turn` est utilisée par les deux threads et permet de coordonner leur exécution. `turn` peut prendre les valeurs `0` ou `1`. Le premier thread exécute la boucle `while (turn != 0) { }`. Prise isolément, cette boucle pourrait apparaître comme une boucle inutile (`turn==0` avant son exécution) ou une boucle infinie (`turn==1` avant son exécution). Un tel raisonnement est incorrect lorsque la variable `turn` peut être modifiée par les deux threads. En effet, si `turn` vaut `1` au début de la boucle `while (turn != 0) { }`, la valeur de cette variable peut être modifiée par un autre thread pendant l'exécution de la boucle et donc provoquer son arrêt.

```
// thread 1
while (turn!=0)
{ /* loop */ }
section_critique();
turn=1;
// ...

// thread 2
while (turn!=1)
{ /* loop */ }
section_critique();
turn=0;
```

Il est intéressant d'analyser ces deux threads en détails pour déterminer si ils permettent d'éviter une violation de section critique et respectent les 4 contraintes précisées plus haut. Dans ces deux threads, pour qu'une violation de section critique puisse se produire, il faudrait que les deux threads passent en même temps la boucle `while` qui précède la section critique. Imaginons que le premier thread est entré dans sa section critique. Puisqu'il est sorti de sa boucle `while`, cela implique que la variable `turn` a la valeur `0`. Sinon, le premier thread serait toujours en train d'exécuter sa boucle `while`. Examinons maintenant le fonctionnement du second thread. Pour entrer dans sa section critique, celui-ci va exécuter la boucle `while (turn != 1){ }`. A ce moment, `turn` a la valeur `0`. La boucle dans le second thread va donc s'exécuter en permanence. Elle ne s'arrêtera que si la valeur de `turn` change. Or, le premier thread ne pourra changer la valeur de `turn` que lorsqu'il aura quitté sa section critique. Cette solution évite donc toute violation de la section critique. Malheureusement, elle ne fonctionne que si il y a une alternance stricte entre les deux threads. Le second s'exécute après le premier qui lui même s'exécute après le second, ... Cette alternance n'est évidemment pas acceptable.

Analysons une seconde solution. Celle-ci utilise un tableau `flag` contenant deux drapeaux, un par thread. Ces deux drapeaux sont initialisés à la valeur `false`. Pour plus de facilité, nous nommons les threads en utilisant la lettre `A` pour le premier et `B` pour le second. Le drapeau `flag[x]` est modifié par le thread `x` et sa valeur est testée par l'autre thread.

```
#define A 0
#define B 1
int flag[];
```

7. Certains systèmes d'exploitation utilisent une désactivation parfois partielle des interruptions pour résoudre des problèmes d'exclusion mutuelle qui portent sur quelques instructions à l'intérieur du système d'exploitation lui-même. Il faut cependant noter qu'une désactivation des interruptions peut être particulièrement couteuse en termes de performances dans un environnement multiprocesseurs.

```
flag[A]=false;
flag[B]=false;
```

Le premier thread peut s'écrire comme suit. Il comprend une boucle `while` qui teste le drapeau `flag[B]` du second thread. Avant d'entrer en section critique, il met son drapeau `flag[A]` à `true` et le remet à `false` dès qu'il en est sorti.

```
// Thread A
while (flag[B]==true)
{ /* loop */ }
flag[A]=true;
section_critique();
flag[A]=false;
//...
```

Le second thread est organisé d'une façon similaire.

```
// Thread B
while (flag[A]==true)
{ /* loop */ }
flag[B]=true;
section_critique();
flag[B]=false;
// ...
```

Analysons le fonctionnement de cette solution et vérifions si elle permet d'éviter toute violation de section critique. Pour qu'une violation de section critique se produise, il faudrait que les deux threads exécutent simultanément leur section critique. La boucle `while` qui précède dans chaque thread l'entrée en section critique parait éviter les problèmes puisque si le thread `A` est dans sa section critique, il a mis `flag[A]` à la valeur `true` et donc le thread `B` exécutera en permanence sa boucle `while`. Malheureusement, la situation suivante est possible. Supposons que `flag[A]` et `flag[B]` ont la valeur `false` et que les deux threads souhaitent entrer dans leur section critique en même temps. Chaque thread va pouvoir traverser sa boucle `while` sans attente puis seulement mettre son drapeau à `true`. A cet instant il est trop tard et une violation de section critique se produira. Cette violation a été illustrée sur une machine multiprocesseur qui exécute deux threads simultanément. Elle est possible également sur une machine monoprocesseur. Dans ce cas, il suffit d'imaginer que le thread `A` passe sa boucle `while` et est interrompu par le scheduler avant d'exécuter `flag[A]=true;`. Le scheduler réalise un changement de contexte et permet au thread `B` de s'exécuter. Il peut passer sa boucle `while` puis entre en section critique alors que le thread `A` est également prêt à y entrer.

Une alternative pour éviter le problème de violation de l'exclusion mutuelle pourrait être d'inverser la boucle `while` et l'assignation du drapeau. Pour le thread `A`, cela donnerait le code ci-dessous :

```
// Thread A
flag[A]=true;
while (flag[B]==true)
{ /* loop */ }
section_critique();
flag[A]=false;
//...
```

Le thread `B` peut s'implémenter de façon similaire. Analysons le fonctionnement de cette solution sur un ordinateur monoprocesseur. Un scénario possible est le suivant. Le thread `A` exécute la ligne permettant d'assigner son drapeau, `flag[A]=true;`. Après cette assignation, le scheduler interrompt ce thread et démarre le thread `B`. Celui-ci exécute `flag[B]=true;` puis démarre sa boucle `while`. Vu le contenu du drapeau `flag[A]`, celle-ci va s'exécuter en permanence. Après quelque temps, le scheduler repasse la main au thread `A` qui va lui aussi entamer sa boucle `while`. Comme `flag[B]` a été mis à `true` par le thread `B`, le thread `A` entame également sa boucle `while`. A partir de cet instant, les deux threads vont exécuter leur boucle `while` qui protège l'accès à la section critique. Malheureusement, comme chaque thread exécute sa boucle `while` aucun des threads ne va modifier son drapeau de façon à permettre à l'autre thread de sortir de sa boucle. Cette situation perdurera indéfiniment. Dans la littérature, cette situation est baptisée un *livelock*. Un *livelock* est une situation dans laquelle plusieurs threads exécutent une séquence d'instructions (dans ce cas les instructions relatives aux boucles `while`)

sans qu'aucun thread ne puisse réaliser de progrès. Un *livelock* est un problème extrêmement gênant puisque lorsqu'il survient les threads concernés continuent à utiliser le processeur mais n'exécutent aucune instruction utile. Il peut être très difficile à diagnostiquer et il est important de réfléchir à la structure du programme et aux techniques de coordination entre les threads qui sont utilisées afin de garantir qu'aucun *livelock* ne pourra se produire.

L'algorithme de Peterson [Peterson1981] combine les deux idées présentées plus tôt. Il utilise une variable `turn` qui est testée et modifiée par les deux threads comme dans la première solution et un tableau `flag[]` comme la seconde. Les drapeaux du tableau sont initialisés à `false` et la variable `turn` peut prendre la valeur `A` ou `B`.

```
#define A 0
#define B 1
int flag[];
flag[A]=false;
flag[B]=false;
```

Le thread `A` peut s'écrire comme suit.

```
// thread A
flag[A]=true;
turn=B;
while((flag[B]==true)&&(turn==B))
{ /* loop */ }
section_critique();
flag[A]=false;
// ...
```

Le thread `B` s'implémente de façon similaire.

```
// Thread B
flag[B]=true;
turn=A;
while((flag[A]==true)&&(turn==A))
{ /* loop */ }
section_critique();
flag[B]=false;
// ...
```

Pour vérifier si cette solution répond bien au problème de l'exclusion mutuelle, il nous faut d'abord vérifier qu'il ne peut y avoir de violation de la section critique. Pour qu'une violation de section critique soit possible, il faudrait que les deux threads soient sortis de leur boucle `while`. Examinons le cas où le thread `B` se trouve en section critique. Dans ce cas, `flag[B]` a la valeur `true`. Si le thread `A` veut entrer en section critique, il va d'abord devoir exécuter `flag[A]=true;` et ensuite `turn=B;`. Comme le thread `B` ne modifie ni `flag[A]` ni `turn` dans sa section critique, thread `A` va devoir exécuter sa boucle `while` jusqu'à ce que le thread `B` sorte de sa section critique et exécute `flag[B]=false;`. Il ne peut donc pas y avoir de violation de la section critique.

Il nous faut également montrer que l'algorithme de Peterson ne peut pas causer de *livelock*. Pour qu'un tel *livelock* soit possible, il faudrait que les boucles `while((flag[A]==true)&&(turn==A)) {};` et `while((flag[B]==true)&&(turn==B)) {};` puissent s'exécuter en permanence en même temps. Comme la variable `turn` ne peut prendre que la valeur `A` ou la valeur `B`, il est impossible que les deux conditions de boucle soient simultanément vraies.

Enfin, considérons l'impact de l'arrêt d'un des deux threads. Si thread `A` s'arrête hors de sa section critique, `flag[A]` a la valeur `false` et le thread `B` pourra toujours accéder à sa section critique.

Utilisation d'instruction atomique

Sur les ordinateurs actuels, il devient difficile d'utiliser l'algorithme de Peterson tel qu'il a été décrit et ce pour deux raisons. Tout d'abord, les compilateurs C sont capables d'optimiser le code qu'ils génèrent. Pour cela, ils analysent le programme à compiler et peuvent supprimer des instructions qui leur semblent être inutiles. Dans le cas de l'algorithme de Peterson, le compilateur pourrait très bien considérer que la boucle `while` est inutile puisque les variables `turn` et `flag` ont été initialisées juste avant d'entrer dans la boucle.

La deuxième raison est que sur un ordinateur multiprocesseur, chaque processeur peut réordonner les accès à la mémoire automatiquement afin d'en optimiser les performances [McKenney2005]. Cela a comme conséquence que certaines lectures et écritures en mémoires peuvent se faire dans un autre ordre que celui indiqué dans le programme sur certaines architectures de processeurs. Si dans l'algorithme de Peterson le thread `A` lit la valeur de `flag[B]` alors que l'écriture en mémoire pour `flag[A]` n'a pas encore été effectuée, une violation de la section critique est possible. En effet, dans ce cas les deux threads peuvent tous les deux passer leur boucle `while` avant que la mise à jour de leur drapeau n'aie été faite effectivement en mémoire.

Pour résoudre ce problème, les architectes de microprocesseurs ont proposé l'utilisation d'opérations atomiques. Une *opération atomique* est une opération qui lorsqu'elle est exécutée sur un processeur ne peut pas être interrompue par l'arrivée d'une interruption. Ces opérations permettent généralement de manipuler en même temps un registre et une adresse en mémoire. En plus de leur caractère interruptible, l'exécution de ces instructions atomiques par un ou plusieurs processeur implique une coordination des processeurs pour l'accès à la zone mémoire référencée dans l'instruction. Via un mécanisme qui sort du cadre de ces notes, tous les accès à la mémoire faits par ces instructions sont ordonnés par les processeurs de façon à ce qu'ils soient toujours réalisés séquentiellement.

Plusieurs types d'instructions atomiques sont supportés par différentes architectures de processeurs. A titre d'exemple, considérons l'instruction atomique `xchg` qui est supportée par les processeurs [IA32]. Cette instruction permet d'échanger, de façon atomique, le contenu d'un registre avec une zone de la mémoire. Elle prend deux arguments, un registre et une adresse en mémoire. Ainsi, l'instruction `xchgl %eax, (var)` est équivalente aux trois instructions suivantes, en supposant le registre `%ebx` initialement vide. La première sauvegarde dans `%ebx` le contenu de la mémoire à l'adresse `var`. La deuxième copie le contenu du registre `%eax` à cette adresse mémoire et la dernière transfère le contenu de `%ebx` dans `%eax` de façon à terminer l'échange de valeurs.

```
movl (var), %ebx
movl %eax, (var)
movl %ebx, %eax
```

Avec cette instruction atomique, il est possible de résoudre le problème de l'exclusion mutuelle en utilisant une zone mémoire, baptisée `lock` dans l'exemple. Cette zone mémoire contiendra la valeur `1` ou `0`. Cette zone mémoire est initialisée à `0`. Lorsqu'un thread veut accéder à sa section critique, il exécute les instructions à partir de l'étiquette `enter:`. Pour sortir de section critique, il suffit d'exécuter les instructions à partir de l'étiquette `leave:`.

```
lock:                   ; étiquette, variable
      .long   0         ; initialisée à 0

enter:
   movl    $1, %eax     ; %eax=1
   xchgl   %eax, (lock) ; instruction atomique, échange (lock) et %eax
                        ; après exécution, %eax contient la donnée qui était
                        ; dans lock et lock la valeur 1
   testl   %eax, %eax   ; met le flag ZF à vrai si %eax contient 0
   jnz     enter        ; retour à enter: si ZF n'est pas vrai
   ret

leave:
   mov     $0, %eax     ; %eax=0
   xchgl   %eax, (lock) ; instruction atomique
   ret
```

Pour bien comprendre le fonctionnement de cette solution, il faut analyser les instructions qui composent chaque routine en assembleur. La routine `leave` est la plus simple. Elle place la valeur `0` à l'adresse `lock`. Elle utilise une instruction atomique de façon à garantir que cet accès en mémoire se fait séquentiellement. Lorsque `lock` vaut `0`, cela indique qu'aucun thread ne se trouve en section critique. Si `lock` contient la valeur `1`, cela indique qu'un thread est actuellement dans sa section critique et qu'aucun autre thread ne peut y entrer. Pour entrer en section critique, un thread doit d'abord exécuter la routine `enter`. Cette routine initialise d'abord le registre `%eax` à la valeur `1`. Ensuite, l'instruction `xchgl` est utilisée pour échanger le contenu de `%eax` avec la zone mémoire `lock`. Après l'exécution de cette instruction atomique, l'adresse `lock` contiendra nécessairement la valeur `1`. Par contre, le registre `%eax` contiendra la valeur qui se trouvait à l'adresse `lock` avant l'exécution de `xchgl`. C'est en testant cette valeur que le thread pourra déterminer si il peut entrer en section critique ou non. Deux cas sont possibles :

1. `%eax==0` après exécution de l'instruction `xchgl %eax, (lock)`. Dans ce cas, le thread peut accéder à sa section critique. En effet, cela indique qu'avant l'exécution de cette instruction l'adresse `lock` contenait la valeur `0`. Cette valeur indique que la section critique était libre avant l'exécution de l'instruction `xchgl %eax, (lock)`. En outre, cette instruction a placé la valeur `1` à l'adresse `lock`, ce qui indique qu'un thread exécute actuellement sa section critique. Si un autre thread exécute l'instruction `xchgl %eax, (lock)` à cet instant, il récupèrera la valeur `1` dans `%eax` et ne pourra donc pas entre en section critique. Si deux threads exécutent simultanément et sur des processeurs différents l'instruction `xchgl %eax, (lock)`, la coordination des accès mémoires entre les processeurs garantit que ces accès mémoires seront séquentiels. Le thread qui bénéficiera du premier accès à la mémoire sera celui qui récupèrera la valeur `0` dans `%eax` et pourra entrer dans sa section critique. Le ou les autres threads récupéreront la valeur `1` dans `%eax` et boucleront.
2. `%eax==1` après exécution de l'instruction `xchgl %eax, (lock)`. Dans ce cas, le thread ne peut entrer en section critique et il entame une boucle active durant laquelle il va continuellement exécuter la boucle `enter: movl ... jnz enter`.

En pratique, rares sont les programmes qui coordonnent leurs threads en utilisant des instructions atomiques ou l'algorithme de Peterson. Ces programmes profitent généralement des fonctions de coordination qui sont implémentées dans des librairies du système d'exploitation.

Coordination par Mutex

L'algorithme de Peterson et l'utilisation d'instructions atomiques sont des mécanismes de base permettant de résoudre le problème de l'exclusion mutuelle. Ils sont utilisés par des fonctions de la libraire POSIX threads. Il est préférable pour des raisons de portabilité et de prise en compte de spécificités matérielles de certains processeurs d'utiliser les fonctions de la librairie POSIX threads plutôt que de redévelopper soi-même ces primitives de coordination entre threads.

Le premier mécanisme de coordination entre threads dans la librairie POSIX sont les *mutex*. Un *mutex* (abréviation de *mutual exclusion*) est une structure de données qui permet de contrôler l'accès à une ressource. Un *mutex* qui contrôle une ressource peut se trouver dans deux états :

— *libre* (ou *unlocked* en anglais). Cet état indique que la ressource est libre et peut être utilisée sans risquer de provoquer une violation d'exclusion mutuelle.
— *réservée* (ou *locked* en anglais). Cet état indique que la ressource associée est actuellement utilisée et qu'elle ne peut pas être utilisée par un autre thread.

Un *mutex* est toujours associé à une ressource. Cette ressource peut être une variable globale comme dans les exemples précédents, mais cela peut aussi être une structure de données plus complexe, une base de données, un fichier, ... Un mutex s'utilise par l'intermédiaire de deux fonctions de base. La fonction *lock* permet à un thread d'acquérir l'usage exclusif d'une ressource. Si la ressource est libre, elle est marquée comme réservée et le thread y accède directement. Si la ressource est occupée, le thread est bloqué par le système d'exploitation jusqu'à ce qu'elle ne devienne libre. A ce moment, le thread pourra poursuivre son exécution et utilisera la ressource avec la certitude qu'aucun autre thread ne pourra faire de même. Lorsque le thread a terminé d'utiliser la ressource associée au mutex, il appelle la fonction *unlock*. Cette fonction vérifie d'abord si un ou plusieurs autres threads sont en attente pour cette ressource (c'est-à-dire qu'ils ont appelé la fonction *lock* mais celle-ci n'a pas encore réussi). Si c'est le cas, un (et un seul) thread est choisi parmi les threads en attente et celui-ci accède à la ressource. Il est important de noter qu'un programme ne peut faire aucune hypothèse sur l'ordre dans lequel les threads qui sont en attente sur un *mutex* pourront accéder à la ressource partagée. Le programme doit être conçu en faisant l'hypothèse que si plusieurs threads sont bloqués sur un appel à *lock* pour un mutex, le thread qui sera libéré est choisi aléatoirement.

Sans entrer dans des détails qui relèvent du fonctionnement internes des systèmes d'exploitation, on peut schématiquement représenter un *mutex* comme étant une structure de données qui contient deux informations :

— la valeur actuelle du *mutex* (*locked* ou *unlocked*)
— une queue contenant l'ensemble des threads qui sont bloqués en attente du mutex

Schématiquement, l'implémentation des fonctions `lock` et `unlock` peut être représentée par le code ci-dessous.

```
lock(mutex m) {
 if(m.val==unlocked)
 {
   m.val=locked;
```

```
 }
 else
 {
   // Place this thread in m.queue;
   // This thread is blocked;
 }
}
```

Le fonction `lock` vérifie si le *mutex* est libre. Dans ce cas, le *mutex* est marqué comme réservé et la fonction `lock` réussi. Sinon, le thread qui a appelé la fonction `lock` est placé dans la queue associée au *mutex* et passe dans l'état *Blocked* jusqu'à ce qu'un autre thread ne libère le mutex.

```
unlock(mutex m) {
 if(m.queue is empty)
 {
   m.val=unlocked;
 }
 else
 {
  // Remove one thread(T) from m.queue;
  // Mark Thread(T) as ready to run;
 }
}
```

La fonction `unlock` vérifie d'abord l'état de la queue associée au *mutex*. Si la queue est vide, cela indique qu'aucun thread n'est en attente. Dans ce cas, la valeur du *mutex* est mise à *unlocked* et la fonction se termine. Sinon, un des threads en attente dans la queue associée au *mutex* est choisi et marqué comme prêt à s'exécuter. Cela indique implicitement que l'appel à `lock` fait par ce thread réussi et qu'il peut accéder à la ressource.

Le code présenté ci-dessous n'est qu'une illustration du fonctionnement des opérations `lock` et `unlock`. Pour que ces opérations fonctionnent correctement, il faut bien entendu que les modifications aux valeurs du *mutex* et à la queue qui y est associée se fassent en garantissant qu'un seul thread exécute l'une de ces opérations sur un *mutex* à un instant donné. En pratique, les implémentations de `lock` et `unlock` utilisent des instructions atomiques telles que celles qui ont été présentées dans la section précédente pour garantir cette propriété.

Les *mutex* sont fréquemment utilisés pour protéger l'accès à une zone de mémoire partagée. Ainsi, si la variable globale *g* est utilisée en écriture et en lecture par deux threads, celle-ci devra être protégée par un *mutex*. Toute modification de cette variable devra être entourée par des appels à *lock* et *unlock*.

En C, cela se fait en utilisant les fonctions pthread_mutex_lock(3posix) et pthread_mutex_unlock(3posix). Un *mutex* POSIX est représenté par une structure de données de type *pthread_mutex_t* qui est définie dans le fichier pthread.h. Avant d'être utilisé, un *mutex* doit être initialisé via la fonction pthread_mutex_init(3posix) et lorsqu'il n'est plus nécessaire, les ressources qui lui sont associées doivent être libérées avec la fonction pthread_mutex_destroy(3posix).

L'exemple ci-dessous reprend le programme dans lequel une variable globale est incrémentée par plusieurs threads.

```
#include <pthread.h>
#define NTHREADS 4

long global=0;
pthread_mutex_t mutex_global;

int increment(int i) {
  return i+1;
}

void *func(void * param) {
  int err;
  for(int j=0;j<1000000;j++) {
    err=pthread_mutex_lock(&mutex_global);
    if(err!=0)
```

```
      error(err,"pthread_mutex_lock");
    global=increment(global);
    err=pthread_mutex_unlock(&mutex_global);
    if(err!=0)
      error(err,"pthread_mutex_unlock");
  }
  return(NULL);
}

int main (int argc, char *argv[])  {
  pthread_t thread[NTHREADS];
  int err;

  err=pthread_mutex_init( &mutex_global, NULL);
  if(err!=0)
      error(err,"pthread_mutex_init");

  for(int i=0;i<NTHREADS;i++) {
    err=pthread_create(&(thread[i]),NULL,&func,NULL);
    if(err!=0)
      error(err,"pthread_create");
  }
  for(int i=0; i<1000000000;i++) { /*...*/ }

  for(int i=NTHREADS-1;i>=0;i--) {
    err=pthread_join(thread[i],NULL);
    if(err!=0)
      error(err,"pthread_join");
  }

  err=pthread_mutex_destroy(&mutex_global);
  if(err!=0)
    error(err,"pthread_mutex_destroy");

  printf("global: %ld\n",global);

  return(EXIT_SUCCESS);
}
```

Il est utile de regarder un peu plus en détails les différentes fonctions utilisées par ce programme. Tout d'abord, la ressource partagée est ici la variable `global`. Dans l'ensemble du programme, l'accès à cette variable est protégé par le *mutex* `mutex_global`. Celui-ci est représenté par une structure de données de type `pthread_mutex_t`.

Avant de pouvoir utiliser un *mutex*, il est nécessaire de l'initialiser. Cette initialisation est effectuée par la fonction pthread_mutex_init(3posix) qui prend deux arguments [8]. Le premier est un pointeur vers une structure `pthread_mutex_t` et le second un pointeur vers une structure `pthread_mutexattr_t` contenant les attributs de ce *mutex*. Tout comme lors de la création d'un thread, ces attributs permettent de spécifier des paramètres à la création du *mutex*. Ces attributs peuvent être manipulés en utilisant les fonctions pthread_mutexattr_gettype(3posix) et pthread_mutexattr_settype(3posix). Dans le cadre de ces notes, nous utiliserons exclusivement les attributs par défaut et créerons toujours un *mutex* en passant `NULL` comme second argument à la fonction pthread_mutex_init(3posix).

Lorsqu'un *mutex* POSIX est initialisé, la ressource qui lui est associée est considérée comme libre. L'accès à la ressource doit se faire en précédant tout accès à la ressource par un appel à la fonction pthread_mutex_lock(3posix). En fonction des attributs spécifiés à la création du *mutex*, il peut y avoir de très rares cas où la fonction retourne une valeur non nulle. Dans ce cas, le type d'erreur est indiqué via *errno*. Lorsque le thread n'a plus besoin de la ressource protégée par le mutex, il doit appeler la fonction pthread_mutex_unlock(3posix) pour libérer la ressource protégée.

8. Linux supporte également la macro `PTHREAD_MUTEX_INITIALIZER` qui permet d'initialiser directement un `pthread_mutex_t` déclaré comme variable globale. Dans cet exemple, la déclaration aurait été : `pthread_mutex_t global_mutex=PTHREAD_MUTEX_INITIALIZER;` et l'appel à pthread_mutex_init(3posix) aurait été inutile. Comme il s'agit d'une extension spécifique à Linux, il est préférable de ne pas l'utiliser pour garantir la portabilité du code.

pthread_mutex_lock(3posix) et pthread_mutex_unlock(3posix) sont toujours utilisés en couple. pthread_mutex_lock(3posix) doit toujours précéder l'accès à la ressource partagée et pthread_mutex_unlock(3posix) doit être appelé dès que l'accès exclusif à la ressource partagée n'est plus nécessaire.

L'utilisation des mutex permet de résoudre correctement le problème de l'exclusion mutuelle. Pour s'en convaincre, considérons le programme ci-dessus et les threads qui exécutent la fonction `func`. Celle-ci peut être résumée par les trois lignes suivantes :

```
pthread_mutex_lock(&mutex_global);
global=increment(global);
pthread_mutex_unlock(&mutex_global);
```

Pour montrer que cette solution répond bien au problème de l'exclusion mutuelle, il faut montrer qu'elle respecte la propriété de sureté et la propriété de vivacité. Pour la propriété de sureté, c'est par construction des *mutex* et parce que chaque thread exécute pthread_mutex_lock(3posix) avant d'entrer en section critique et pthread_mutex_unlock(3posix) dès qu'il en sort. Considérons le cas de deux threads qui sont en concurrence pour accéder à cette section critique. Le premier exécute pthread_mutex_lock(3posix). Il accède à sa section critique. A partir de cet instant, le second thread sera bloqué dès qu'il exécute l'appel à pthread_mutex_lock(3posix). Il restera bloqué dans l'exécution de cette fonction jusqu'à ce que le premier thread sorte de sa section critique et exécute pthread_mutex_unlock(3posix). A ce moment, le premier thread n'est plus dans sa section critique et le système peut laisser le second y entrer en terminant l'exécution de l'appel à pthread_mutex_lock(3posix). Si un troisième thread essaye à ce moment d'entrer dans la section critique, il sera bloqué sur son appel à pthread_mutex_lock(3posix).

Pour montrer que la propriété de vivacité est bien respectée, il faut montrer qu'un thread ne sera pas empêché éternellement d'entrer dans sa section critique. Un thread peut être empêché d'entrer dans sa section critique en étant bloqué sur l'appel à pthread_mutex_lock(3posix). Comme chaque thread exécute pthread_mutex_unlock(3posix) dès qu'il sort de sa section critique, le thread en attente finira par être exécuté. Pour qu'un thread utilisant le code ci-dessus ne puisse jamais entrer en section critique, il faudrait qu'il y aie en permanence plusieurs threads en attente sur pthread_mutex_unlock(3posix) et que notre thread ne soit jamais sélectionné par le système lorsque le thread précédent termine sa section critique.

4.3.2 Le problème des philosophes

Le problème de l'exclusion mutuelle considère le cas de plusieurs threads qui se partagent une ressource commune qui doit être manipulée de façon exclusive. C'est un problème fréquent qui se pose lorsque l'on développe des programmes décomposés en différents threads d'exécution, mais ce n'est pas le seul. En pratique, il est souvent nécessaire de coordonner l'accès de plusieurs threads à plusieurs ressources, chacune de ces ressources devant être utilisée de façon exclusive. Cette utilisation de plusieurs ressources simultanément peut poser des difficultés. Un des problèmes classiques qui permet d'illustrer la difficulté de coordonner l'accès à plusieurs ressources est le *problème des philosophes*. Ce problème a été proposé par Dijkstra et illustre en termes simples la difficulté de coordonner l'accès à plusieurs ressources.

Dans le *problème des philosophes*, un ensemble de philosophes doivent se partager des baguettes pour manger. Tous les philosophes se trouvent dans une même pièce qui contient une table circulaire. Chaque philosophe dispose d'une place qui lui est réservée sur cette table. La table comprend autant de baguettes que de chaises et une baguette est placée entre chaque paire de chaises. Chaque philosophe est modélisé sous la forme d'un thread qui effectue deux types d'actions : *penser* et *manger*. Pour pouvoir manger, un philosophe doit obtenir la baguette qui se trouve à sa gauche et la baguette qui se trouve à sa droite. Lorsqu'il a fini de manger, il peut retourner à son activité philosophale. La figure ci-dessous illustre une table avec les assiettes de trois philosophes et les trois baguettes qui sont à leur disposition.

Ce problème de la vie courante peut se modéliser en utilisant un programme C avec les threads POSIX. Chaque baguette est une ressource partagée qui ne peut être utilisée que par un philosophe à la fois. Elle peut donc être modélisée par un *mutex*. Chaque philosophe est modélisé par un thread qui pense puis ensuite appelle pthread_mutex_lock(3posix) pour obtenir chacune de ses baguettes. Le philosophe peut ensuite manger puis il libère ses baguettes en appelant pthread_mutex_unlock(3posix). L'extrait [9] ci-dessous comprend les fonctions utilisées par chacun des threads.

9. Le programme complet est `/Threads/S6-src/pthread-philo.c`

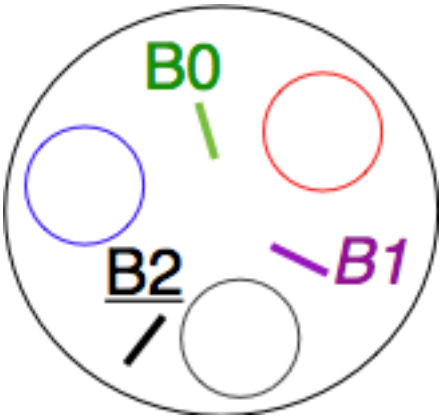


FIGURE 4.6 – Problème des philosophes

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

#define PHILOSOPHES 3

pthread_t phil[PHILOSOPHES];
pthread_mutex_t baguette[PHILOSOPHES];

void mange(int id) {
  printf("Philosophe [%d] mange\n",id);
  for(int i=0;i< rand(); i++) {
    // philosophe mange
  }
}

void* philosophe ( void* arg )
{
  int *id=(int *) arg;
  int left = *id;
  int right = (left + 1) % PHILOSOPHES;
  while(true) {
    printf("Philosophe [%d] pense\n",*id);
    pthread_mutex_lock(&baguette[left]);
    printf("Philosophe [%d] possède baguette gauche [%d]\n",*id,left);
    pthread_mutex_lock(&baguette[right]);
    printf("Philosophe [%d] possède baguette droite [%d]\n",*id,right);
    mange(*id);
    pthread_mutex_unlock(&baguette[left]);
    printf("Philosophe [%d] a libéré baguette gauche [%d]\n",*id,left);
    pthread_mutex_unlock(&baguette[right]);
    printf("Philosophe [%d] a libéré baguette droite [%d]\n",*id,right);
  }
  return (NULL);
}
```

Malheureusement, cette solution ne permet pas de résoudre le problème des philosophes. En effet, la première exécution du programme (`/Threads/S6-src/pthread-philo.c`) indique à l'écran que les philosophes se partagent les baguettes puis après une fraction de seconde le programme s'arrête en affichant une sortie qui se termine :

```
Philosophe [0] a libéré baguette gauche [0]
Philosophe [0] a libéré baguette droite [1]
Philosophe [0] pense
Philosophe [0] possède baguette gauche [0]
Philosophe [2] possède baguette gauche [2]
Philosophe [1] possède baguette gauche [1]
```

Ce blocage de programme est un autre problème majeur auquel il faut faire attention lorsque l'on découpe un programme en plusieurs threads d'exécution. Il porte le nom de *deadlock* que l'on peut traduire en français pas étreinte fatale. Un programme est en situation de deadlock lorsque tous ses threads d'exécution sont bloqués et qu'aucun d'entre eux ne peux être débloqué sans exécuter d'instructions d'un des threads bloqués. Dans ce cas particulier, le programme est bloqué parce que le `philosophe[0]` a pu réserver la `baguette[0]`. Malheureusement, en même temps le `philosophe[2]` a obtenu `baguette[2]` et `philosophe[0]` est donc bloqué sur la ligne `pthread_mutex_lock(&baguette[right]);`. Entre temps, le `philosophe[1]` a pu exécuter la ligne `pthread_mutex_lock(&baguette[left]);` et a obtenu `baguette[1]`. Dans cet état, tous les threads sont bloqués sur la ligne `pthread_mutex_lock(&baguette[right]);` et plus aucun progrès n'est possible.

Ce problème de deadlock est un des problèmes les plus graves qui peuvent survenir dans un programme découpé en plusieurs threads. Si les threads entrent dans une situation de deadlock, le programme est complètement bloqué et la seule façon de sortir du deadlock est d'arrêter le programme et de le redémarrer. Ce n'est évidemment pas acceptable. Dans l'exemple des philosophes ci-dessus, le deadlock apparait assez rapidement car les threads passent la majorité de leur temps à exécuter les fonctions de la librairie threads POSIX. En pratique, un thread exécute de nombreuses lignes de code standard et utilise rarement la fonction pthread_mutex_lock(3posix). Dans de tels programmes, les tests ne permettent pas nécessairement de détecter toutes les situations qui peuvent causer un deadlock. Il est important que le programme soit conçu de façon à toujours éviter les deadlocks. Si c'est n'est pas le cas, le programme finira toujours bien à se retrouver en situation de deadlock après un temps plus ou moins long.

Le problème des philosophes est une illustration d'un problème courant dans lequel plusieurs threads doivent utiliser deux ou plusieurs ressources partagées contrôlées par un mutex. Dans la situation de deadlock, chaque thread est bloqué à l'exécution de la ligne `pthread_mutex_lock(&baguette[right]);`. On pourrait envisager de chercher à résoudre le problème en inversant les deux appels à pthread_mutex_lock(3posix) comme ci-dessous :

```
pthread_mutex_lock(&baguette[right]);
pthread_mutex_lock(&baguette[left]);
```

Malheureusement, cette solution ne résout pas le problème du deadlock. La seule différence par rapport au programme précédent est que les mutex qui sont alloués à chaque thread en deadlock ne sont pas les mêmes. Avec cette nouvelle version du programme, lorsque le deadlock survient, les mutex suivants sont alloués :

```
Philosophe [2] possède baguette droite [1]
Philosophe [0] possède baguette droite [2]
Philosophe [1] possède baguette droite [0]
```

Pour comprendre l'origine du deadlock, il faut analyser plus en détails le fonctionnement du programme et l'ordre dans lequel les appels à pthread_mutex_lock(3posix) sont effectués. Trois mutex sont utilisés dans le programme des philosophes : `baguette[0]`, `baguette[1]` et `baguette[2]`. De façon imagée, chaque philosophe s'approprie d'abord la baguette se trouvant à sa gauche et ensuite la baguette se trouvant à sa droite.

Philosophe	premier mutex	second mutex
`philosophe[0]`	`baguette[2]`	`baguette[0]`
`philosophe[1]`	`baguette[0]`	`baguette[1]`
`philosophe[2]`	`baguette[1]`	`baguette[2]`

L'origine du deadlock dans cette solution au problème des philosophes est l'ordre dans lequel les différents philosophes s'approprient les mutex. Le `philosophe[0]` s'approprie d'abord le mutex `baguette[2]` et ensuite essaye de s'approprier le mutex `baguette[0]`. Le `philosophe[1]` par contre peut lui directement s'approprier le mutex `baguette[0]`. Au vu de l'ordre dans lequel les mutex sont alloués, il est possible que chaque thread se soit approprié son premier mutex mais soit bloqué sur la réservation du second. Dans ce cas, un deadlock

se produit puisque chaque thread est en attente de la libération d'un mutex sans qu'il ne puisse libérer lui-même le mutex qu'il s'est déjà approprié.

Il est cependant possible de résoudre le problème en forçant les threads à s'approprier les mutex qu'ils utilisent dans le même ordre. Considérons le tableau ci-dessous dans lequel `philosophe[0]` s'approprie d'abord le mutex `baguette[0]` et ensuite le mutex `baguette[2]`.

Philosophe	premier mutex	second mutex
`philosophe[0]`	`baguette[0]`	`baguette[2]`
`philosophe[1]`	`baguette[0]`	`baguette[1]`
`philosophe[2]`	`baguette[1]`	`baguette[2]`

Avec cet ordre d'allocation des mutex, un deadlock n'est plus possible. Il y aura toujours au moins un philosophe qui pourra s'approprier les deux baguettes dont il a besoin pour manger. Pour s'en convaincre, analysons les différentes exécutions possibles. Un deadlock ne pourrait survenir que si tous les philosophes cherchent à manger simultanément. Si un des philosophes ne cherche pas à manger, ses deux baguettes sont nécessairement libres et au moins un de ses voisins philosophes pourra manger. Lorsque tous les philosophes cherchent à manger simultanément, le mutex `baguette[0]` ne pourra être attribué qu'à `philosophe[0]` ou `philosophe[1]`. Considérons les deux cas possibles.

1. `philosophe[0]` s'approprie `baguette[0]`. Dans ce cas, `philosophe[1]` est bloqué sur son premier appel à pthread_mutex_lock(3posix). Comme `philosophe[1]` n'a pas pu s'approprier le mutex `baguette[0]`, il ne peut non plus s'approprier `baguette[1]`. `philosophe[2]` pourra donc s'approprier le mutex `baguette[1]`. `philosophe[0]` et `philosophe[2]` sont maintenant en compétition pour le mutex `baguette[2]`. Deux cas sont à nouveau possibles.
   (a) `philosophe[0]` s'approprie `baguette[2]`. Comme c'est le second mutex nécessaire à ce philosophe, il peut manger. Pendant ce temps, `philosophe[2]` est bloqué en attente de `baguette[2]`. Lorsque `philosophe[0]` aura terminé de manger, il libèrera les mutex `baguette[2]` et `baguette[0]`, ce qui permettra à `philosophe[2]` ou `philosophe[1]` de manger.
   (b) `philosophe[2]` s'approprie `baguette[2]`. Comme c'est le second mutex nécessaire à ce philosophe, il peut manger. Pendant ce temps, `philosophe[0]` est bloqué en attente de `baguette[2]`. Lorsque `philosophe[2]` aura terminé de manger, il libèrera les mutex `baguette[2]` et `baguette[1]`, ce qui permettra à `philosophe[0]` ou `philosophe[1]` de manger.
2. `philosophe[1]` s'approprie `baguette[0]`. Le même raisonnement que ci-dessus peut être suivi pour montrer qu'il n'y a pas de deadlock non plus dans ce cas.

La solution présentée empêche tout deadlock puisqu'à tout moment il n'y a au moins un philosophe qui peut manger. Malheureusement, il est possible avec cette solution qu'un philosophe mange alors que chacun des autres philosophes a pu s'approprier une baguette. Lorsque le nombre de philosophes est élevé (imaginez un congrès avec une centaine de philosophes), cela peut être une source importante d'inefficacité au niveau des performances.

Une implémentation possible de l'ordre présenté dans la table ci-dessus est reprise dans le programme `/Threads/S6-src/pthread-philo2.c` qui comprend la fonction `philosophe` suivante.

```
void* philosophe ( void* arg )
{
  int *id=(int *) arg;
  int left = *id;
  int right = (left + 1) % PHILOSOPHES;
  while(true) {
    // philosophe pense
    if(left<right) {
      pthread_mutex_lock(&baguette[left]);
      pthread_mutex_lock(&baguette[right]);
    }
    else {
      pthread_mutex_lock(&baguette[right]);
      pthread_mutex_lock(&baguette[left]);
    }
    mange(*id);
    pthread_mutex_unlock(&baguette[left]);
    pthread_mutex_unlock(&baguette[right]);
```

```
    }
  return (NULL);
}
```

Ce problème des philosophes est à l'origine d'une règle de bonne pratique essentielle pour tout programme découpé en threads dans lequel certains threads doivent acquérir plusieurs mutex. Pour éviter les deadlocks, il est nécessaire d'ordonnancer tous les mutex utilisés par le programme dans un ordre strict. Lorsqu'un thread doit réserver plusieurs mutex en même temps, il doit *toujours* effectuer ses appels à pthread_mutex_lock(3posix) dans l'ordre choisi pour les mutex. Si cet ordre n'est pas respecté par un des threads, un deadlock peut se produire.

4.4 Les sémaphores

Le problème de la coordination entre threads est un problème majeur. Outre les *mutex* que nous avons présenté, d'autres solutions à ce problème ont été développées. Historiquement, une des premières propositions de coordination sont les sémaphores [Dijkstra1965b]. Un *sémaphore* est une structure de données qui est maintenue par le système d'exploitation et contient :

— un entier qui stocke la valeur, positive ou nulle, du sémaphore.
— une queue qui contient les pointeurs vers les threads qui sont bloqués en attente sur ce sémaphore.

Tout comme pour les *mutex*, la queue associée à un sémaphore permet de bloquer les threads qui sont en attente d'une modification de la valeur du sémaphore.

Une implémentation des sémaphores se compose en général de quatre fonctions :

— une fonction d'initialisation qui permet de créer le sémaphore et de lui attribuer une valeur initiale nulle ou positive.
— une fonction permettant de détruire un sémaphore et de libérer les ressources qui lui sont associées.
— une fonction `post` qui est utilisée par les threads pour modifier la valeur du sémaphore. S'il n'y a pas de thread en attente dans la queue associée au sémaphore, sa valeur est incrémentée d'une unité. Sinon, un des threads en attente est libéré et passe à l'état *Ready*.
— une fonction `wait` qui est utilisée par les threads pour tester la valeur d'un sémaphore. Si la valeur du sémaphore est positive, elle est décrémentée d'une unité et la fonction réussit. Si le sémaphore a une valeur nulle, le thread est bloqué jusqu'à ce qu'un autre thread le débloque en appelant la fonction `post`.

Les sémaphores sont utilisés pour résoudre de nombreux problèmes de coordination [Downey2008]. Comme ils permettent de stocker une valeur entière, ils sont plus flexibles que les *mutex* qui sont utiles surtout pour les problèmes d'exclusion mutuelle.

4.4.1 Sémaphores POSIX

La librairie POSIX comprend une implémentation des sémaphores [10] qui expose plusieurs fonctions aux utilisateurs. La page de manuel sem_overview(7) présente de façon sommaire les fonctions de la librairie relatives aux sémaphores. Les quatre principales sont les suivantes :

```
#include <semaphore.h>

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

Le fichier semaphore.h contient les différentes définitions de structures qui sont nécessaires au bon fonctionnement des sémaphores ainsi que les signatures des fonctions de l'API. Un sémaphore est représenté par une structure de données de type `sem_t`. Toutes les fonctions de manipulation des sémaphores prennent comme argument un pointeur vers le sémaphore concerné.

10. Les systèmes Unix supportent également des sémaphores dits *System V* du nom de la version de Unix dans laquelle ils ont été introduits. Dans ces notes, nous nous focalisons sur les sémaphores POSIX qui ont une API un peu plus simple que l'API des sémaphores *System V*. Les principales fonctions pour les sémaphores *System V* sont semget(3posix), semctl(3posix) et semop(3posix).

Pour pouvoir utiliser un sémaphore, il faut d'abord l'initialiser. Cela se fait en utilisant la fonction sem_init(3) qui prend comme argument un pointeur vers le sémaphore à initialiser. Nous n'utiliserons pas le second argument dans ce chapitre. Le troisième argument est la valeur initiale, positive ou nulle, du sémaphore.

La fonction sem_destroy(3) permet de libérer un sémaphore qui a été initialisé avec sem_init(3). Les sémaphores consomment des ressources qui peuvent être limitées dans certains environnements. Il est important de détruire proprement les sémaphores dès qu'ils ne sont plus nécessaires.

Les deux principales fonctions de manipulation des sémaphores sont sem_wait(3) et sem_post(3). Certains auteurs utilisent `down` ou `P` à la place de sem_wait(3) et `up` ou `V` à la place de sem_post(3) [Downey2008]. Schématiquement, l'opération `sem_wait` peut s'implémenter en utilisant le pseudo-code suivant :

```
int sem_wait(semaphore *s)
{
 s->val=s->val-1;
 if(s->val<0)
 {
  // Place this thread in s.queue;
  // This thread is blocked;
 }
}
```

La fonction sem_post(3) quant à elle peut schématiquement s'implémenter comme suit :

```
int sem_post(semaphore *s)
{
  s->val=s->val+1;
  if(s.val<=0)
  {
    // Remove one thread(T) from s.queue;
    // Mark Thread(T) as ready to run;
  }
}
```

Ces deux opérations sont bien entendu des opérations qui ne peuvent s'exécuter simultanément. Leur implémentation réelle comprend des sections critiques qui doivent être construites avec soin. Le pseudo-code ci-dessus ignore ces sections critiques. Des détails complémentaires sur l'implémentation des sémaphores peuvent être obtenus dans un textbook sur les systèmes d'exploitation [Stallings2011] [Tanenbaum+2009] .

La meilleure façon de comprendre leur utilisation est d'analyser des problèmes classiques de coordination qui peuvent être résolus en utilisant des sémaphores.

4.4.2 Exclusion mutuelle

Les sémaphores permettent de résoudre de nombreux problèmes classiques. Le premier est celui de l'exclusion mutuelle. Lorsqu'il est initialisé à `1`, un sémaphore peut être utilisé de la même façon qu'un *mutex*. En utilisant des sémaphores, une exclusion mutuelle peut être protégée comme suit :

```
#include <semaphore.h>

//...

sem_t semaphore;

sem_init(&semaphore, 0, 1);

sem_wait(&semaphore);
// section critique
sem_post(&semaphore);

sem_destroy(&semaphore);
```

Les sémaphores peuvent être utilisés pour d'autres types de synchronisation. Par exemple, considérons une application découpée en threads dans laquelle la fonction `after` ne peut jamais être exécutée avant la fin de l'exécution de la fonction `before`. Ce problème de coordination peut facilement être résolu en utilisant un sémaphore qui est initialisé à la valeur `0`. La fonction `after` doit démarrer par un appel à sem_wait(3) sur ce sémaphore tandis que la fonction `before` doit se terminer par un appel à la fonction sem_post(3) sur ce sémaphore. De cette façon, si le thread qui exécute la fonction `after` est trop rapide, il sera bloqué sur l'appel à sem_wait(3). S'il arrive à cette fonction après la fin de la fonction `before` dans l'autre thread, il pourra passer sans être bloqué. Le programme ci-dessous illustre cette utilisation des sémaphores POSIX.

```
#define NTHREADS 2
sem_t semaphore;

void *before(void * param) {
  // do something
  for(int j=0;j<1000000;j++) {
  }
  sem_post(&semaphore);
  return(NULL);
}

void *after(void * param) {
  sem_wait(&semaphore);
  // do something
  for(int j=0;j<1000000;j++) {
  }
  return(NULL);
}

int main (int argc, char *argv[])  {
  pthread_t thread[NTHREADS];
  void * (* func[])(void *)={before, after};
  int err;

  err=sem_init(&semaphore, 0,0);
  if(err!=0) {
      error(err,"sem_init");
  }
  for(int i=0;i<NTHREADS;i++) {
    err=pthread_create(&(thread[i]),NULL,func[i],NULL);
    if(err!=0) {
      error(err,"pthread_create");
    }
  }

  for(int i=0;i<NTHREADS;i++) {
    err=pthread_join(thread[i],NULL);
    if(err!=0) {
      error(err,"pthread_join");
    }
  }
  sem_destroy(&semaphore);
  if(err!=0) {
      error(err,"sem_destroy");
  }
  return(EXIT_SUCCESS);
}
```

Si conceptuellement un sémaphore initialisé à la valeur `1` est généralement utilisé comme un *mutex*, il y a une différence importante entre les implémentations des sémaphores et des *mutex*. Un sémaphore est conçu pour être manipulé par différents threads et il est fort possible qu'un thread exécute sem_wait(3) et qu'un autre exécute sem_post(3). Pour les mutex, certaines implémentations supposent que le même thread exécute pthread_mutex_lock(3posix) et pthread_mutex_unlock(3posix). Lorsque ces opérations doivent être effectuées

dans des threads différents, il est préférable d'utiliser des sémaphores à la place de mutex.

4.4.3 Problème du rendez-vous

Le problème du rendez-vous [Downey2008] est un problème assez courant dans les applications multithreadées. Considérons une application découpée en *N* threads. Chacun de ces threads travaille en deux phases. Durant la première phase, tous les threads sont indépendants et peuvent s'exécuter simultanément. Cependant, un thread ne peut démarrer sa seconde phase que si tous les *N* threads ont terminé leur première phase. L'organisation de chaque thread est donc :

```
premiere_phase();
// rendez-vous
seconde_phase();
```

Chaque thread doit pouvoir être bloqué à la fin de la première phase en attendant que tous les autres threads aient fini d'exécuter leur première phase. Cela peut s'implémenter en utilisant un mutex et un sémaphore.

```
sem_t rendezvous;
pthread_mutex_t mutex;
int count=0;

sem_init(&rendezvous,0,0);
```

La variable `count` permet de compter le nombre de threads qui ont atteint le point de rendez-vous. Le `mutex` protège les accès à la variable `count` qui est partagée entre les différents threads. Le sémaphore `rendezvous` est initialisé à la valeur `0`. Le rendez-vous se fera en bloquant les threads sur le sémaphore `rendezvous` tant que les `N` threads ne sont pas arrivés à cet endroit.

```
premiere_phase();

 // section critique
 pthread_mutex_lock(&mutex);
 count++;
  if(count==N) {
    // tous les threads sont arrivés
    sem_post(&rendezvous);
    }
 pthread_mutex_unlock(&mutex);
// attente à la barrière
sem_wait(&rendezvous);
// libération d'un autre thread en attente
sem_post(&rendezvous);

seconde_phase();
```

Le pseudo-code ci-dessus présente une solution permettant de résoudre ce problème du rendez-vous. Le sémaphore étant initialisé à `0`, le premier thread qui aura terminé la première phase sera bloqué sur `sem_wait(&rendezvous);`. Les `N-1` premiers threads qui auront terminé leur première phase seront tous bloqués à cet endroit. Lorsque le dernier thread finira sa première phase, il incrémentera `count` puis exécutera `sem_post(&rendezvous);` ce qui libèrera un premier thread. Le dernier thread sera ensuite bloqué sur `sem_wait(&rendezvous);` mais il ne restera pas bloqué longtemps car chaque fois qu'un thread parvient à passer `sem_wait(&rendezvous);`, il exécute immédiatement `sem_post(&rendezvous);` ce qui permet de libérer un autre thread en cascade.

Cette solution permet de résoudre le problème du rendez-vous avec un nombre fixe de threads. Certaines implémentations de la librairie des threads POSIX comprennent une barrière qui peut s'utiliser de la même façon que la solution ci-dessus. Une barrière est une structure de données de type `pthread_barrier_t`. Elle s'initialise en utilisant la fonction pthread_barrier_init(3posix) qui prend trois arguments : un pointeur vers une barrière, des attributs optionnels et le nombre de threads qui doivent avoir atteint la barrière pour que celle-ci s'ouvre. La fonction pthread_barrier_destroy(3posix) permet de détruire une barrière. Enfin, la fonction pthread_barrier_wait(3posix)

qui prend comme argument un pointeur vers une barrière bloque le thread correspondant à celle-ci tant que le nombre de threads requis pour passer la barrière n'a pas été atteint.

4.4.4 Problème des producteurs-consommateurs

Le problème des producteurs-consommateurs est un problème extrêmement fréquent et important dans les applications découpées en plusieurs threads. Il est courant de structurer une telle application, notamment si elle réalise de longs calculs, en deux types de threads :

— les *producteurs* : Ce sont des threads qui produisent des données et placent le résultat de leurs calculs dans une zone mémoire accessible aux consommateurs.

— les *consommateurs* : Ce sont des threads qui utilisent les valeurs calculées par les producteurs.

Ces deux types de threads communiquent en utilisant un buffer qui a une capacité limitée à *N* slots comme illustré dans la figure ci-dessous.

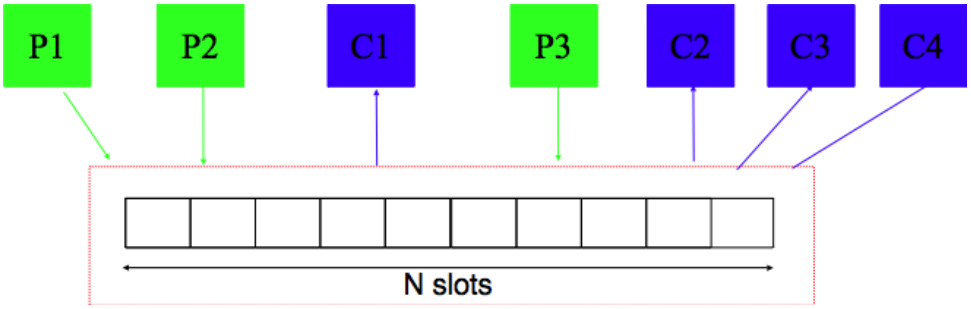


FIGURE 4.7 – Problème des producteurs-consommateurs

La difficulté du problème est de trouver une solution qui permet aux producteurs et aux consommateurs d'avancer à leur rythme sans que les producteurs ne bloquent inutilement les consommateurs et inversement. Le nombre de producteurs et de consommateurs ne doit pas nécessairement être connu à l'avance et ne doit pas être fixe. Un producteur peut arrêter de produire à n'importe quel moment.

Le buffer étant partagé entre les producteurs et les consommateurs, il doit nécessairement être protégé par un *mutex*. Les producteurs doivent pouvoir ajouter de l'information dans le buffer partagé tant qu'il y a au moins un slot de libre dans le buffer. Un producteur ne doit être bloqué que si tout le buffer est rempli. Inversement, les consommateurs doivent être bloqués uniquement si le buffer est entièrement vide. Dès qu'une donnée est ajoutée dans le buffer, un consommateur doit être réveillé pour traiter cette donnée.

Ce problème peut être résolu en utilisant deux sémaphores et un mutex. L'accès au buffer, que ce soit par les consommateurs ou les producteurs est une section critique. Cet accès doit donc être protégé par l'utilisation d'un mutex. Quant aux sémaphores, le premier, baptisé `empty` dans l'exemple ci-dessous, sert à compter le nombre de slots qui sont vides dans le buffer partagé. Ce sémaphore doit être initialisé à la taille du buffer puisqu'initialement celui-ci est vide. Le second sémaphore est baptisé `full` dans le pseudo-code ci-dessous. Sa valeur représente le nombre de slots du buffer qui sont occupés. Il doit être initialisé à la valeur `0`.

```
// Initialisation
#define N 10 // slots du buffer
pthread_mutex_t mutex;
sem_t empty;
sem_t full;

pthread_mutex_init(&mutex, NULL);
sem_init(&empty, 0 , N);  // buffer vide
sem_init(&full, 0 , 0);   // buffer vide
```

Le fonctionnement général d'un producteur est le suivant. Tout d'abord, le producteur est mis en attente sur le sémaphore `empty`. Il ne pourra passer que si il y a au moins un slot du buffer qui est vide. Lorsque la ligne `sem_wait(&empty);` réussit, le producteur s'approprie le `mutex` et modifie le buffer de façon à insérer l'élément produit (dans ce cas un entier). Il libère ensuite le `mutex` pour sortir de sa section critique.

```
// Producteur
void producer(void)
{
  int item;
  while(true)
  {
    item=produce(item);
    sem_wait(&empty); // attente d'un slot libre
    pthread_mutex_lock(&mutex);
     // section critique
     insert_item();
    pthread_mutex_unlock(&mutex);
    sem_post(&full); // il y a un slot rempli en plus
  }
}
```

Le consommateur quant à lui essaie d'abord de prendre le sémaphore `full`. Si celui-ci est positif, cela indique la présence d'au moins un élément dans le buffer partagé. Ensuite, il entre dans la section critique protégée par le `mutex` et récupère la donnée se trouvant dans le buffer. Puis, il incrémente la valeur du sémaphore `empty` de façon à indiquer à un producteur qu'un nouveau slot est disponible dans le buffer.

```
// Consommateur
void consumer(void)
{
 int item;
 while(true)
 {
   sem_wait(&full); // attente d'un slot rempli
   pthread_mutex_lock(&mutex);
    // section critique
    item=remove(item);
   pthread_mutex_unlock(&mutex);
   sem_post(&empty); // il y a un slot libre en plus
 }
}
```

De nombreux programmes découpés en threads fonctionnent avec un ensemble de producteurs et un ensemble de consommateurs.

4.4.5 Problème des readers-writers

Le *problème des readers-writers* est un peu différent du précédent. Il permet de modéliser un problème qui survient lorsque des threads doivent accéder à une base de données [Courtois+1971]. Les threads sont généralement de deux types.

- les lecteurs (*readers*) sont des threads qui lisent une structure de données (ou une base de données) mais ne la modifient pas. Comme ces threads se contentent de lire de l'information en mémoire, rien ne s'oppose à ce que plusieurs *readers* s'exécutent simultanément.
- les écrivains (*writers*). Ce sont des threads qui modifient une structure de données (ou une base de données). Pendant qu'un *writer* manipule la structure de données, il ne peut y avoir aucun autre *writer* ni de *reader* qui accède à cette structure de données. Sinon, la concurrence des opérations de lecture et d'écriture donnerait un résultat incorrect.

Une première solution à ce problème est d'utiliser un mutex et un sémaphore [Courtois+1971].

```
pthread_mutex_t mutex;
sem_t db;  // accès à la db
int readcount=0; // nombre de readers

sem_init(&db, NULL, 1).
```

La solution utilise une variable partagée : `readcount`. L'accès à cette variable est protégé par `mutex`. Le sémaphore `db` sert à réguler l'accès des *writers* à la base de données. Le mutex est initiliasé comme d'habitude

par la fonction pthread_mutex_init(3posix). Le sémaphore `db` est initialisé à la valeur `1`. Le *writer* est assez simple :

```
void writer(void)
{
 while(true)
 {
   prepare_data();
   sem_wait(&db);
     // section critique, un seul writer à la fois
     write_database();
   sem_post(&db);
   }
 }
}
```

Le sémaphore `db` sert à assurer l'exclusion mutuelle entre les *writers* pour l'accès à la base de données. Le fonctionnement des *readers* est plus intéressant. Pour éviter un conflit entre les *writers* et les *readers*, il est nécessaire d'empêcher aux *readers* d'accéder à la base de données pendant qu'un *writer* la modifie. Cela peut se faire en utilisant l'entier *readcount* qui permet de compter le nombre de *readers* qui manipulent la base de données. Cette variable est testée et modifiée par tous les *readers*, elle doit donc être protégée par un *mutex*. Intuitivement, lorsque le premier *reader* veut accéder à la base de données (`readcount==0`), il essaye de décrémenter le sémaphore `db`. Si ce sémaphore est libre, le *reader* accède à la base de données. Sinon, il bloque sur `sem_wait(&db);` et comme il possède `mutex`, tous les autres *readers* sont bloqués sur `pthread_mutex_lock(&mutex);`. Dès que le premier *reader* est débloqué, il autorise en cascade l'accès à tous les autres *readers* qui sont en attente en libérant `pthread_mutex_unlock(&mutex);`. Lorsqu'un *reader* arrête d'utiliser la base de données, il vérifie s'il était le dernier *reader*. Si c'est le cas, il libère le sémaphore `db` de façon à permettre à un *writer* d'y accéder. Sinon, il décrémente simplement la variable `readcount` pour tenir à jour le nombre de *readers* qui sont actuellement en train d'accéder à la base de données.

```
void reader(void)
{
 while(true)
 {
   pthread_mutex_lock(&mutex);
     // section critique
     readcount++;
     if (readcount==1)
     { // arrivée du premier reader
       sem_wait(&db);
     }
   pthread_mutex_unlock(&mutex);
   read_database();
   pthread_mutex_lock(&mutex);
     // section critique
     readcount--;
     if(readcount==0)
     { // départ du dernier reader
       sem_post(&db);
     }
   pthread_mutex_unlock(&mutex);
   process_data();
 }
}
```

Cette solution fonctionne et garantit qu'il n'y aura jamais qu'un seul *writer* qui accède à la base de données. Malheureusement, elle souffre d'un inconvénient majeur lorsqu'il y a de nombreux *readers*. Dans ce cas, il est tout à fait possible qu'il y ait en permanence des *readers* qui accèdent à la base de données et que les *writers* soient toujours empêchés d'y accéder. En effet, dès que le premier *reader* a effectué `sem_wait(&db);`, aucun autre *reader* ne devra exécuter cette opération tant qu'il restera au moins un *reader* accédant à la base de données. Les *writers* par contre resteront bloqués sur l'exécution de `sem_wait(&db);`.

En utilisant des sémaphores à la place des *mutex*, il est possible de contourner ce problème. Cependant, cela nécessite d'utiliser plusieurs sémaphores. Intuitivement, l'idée de la solution est de donner priorité aux *writers* par rapport aux *readers*. Dès qu'un *writer* est prêt à accéder à la base de données, il faut empêcher de nouveaux *readers* d'y accéder tout en permettant aux *readers* présents de terminer leur lecture.

Cette solution utilise trois mutex, deux sémaphores et deux variables partagées : `readcount` et `writecount`. Ces deux variables servent respectivement à compter le nombre de *readers* qui accèdent à la base de données et le nombre de *writers* qui veulent y accéder. Le sémaphore `wsem` est utilisé pour bloquer les *writers* tandis que le sémaphore `rsem` sert à bloquer les *readers*. Le mutex `z` a un rôle particulier qui sera plus clair lorsque le code des *readers* aura été présenté. Les deux sémaphores sont initialisés à la valeur `1`.

```
/* Initialisation */
pthread_mutex_t mutex_readcount; // protège readcount
pthread_mutex_t mutex_writecount; // protège writecount
pthread_mutex_t z; // un seul reader en attente
sem_t wsem;        // accès exclusif à la db
sem_t rsem;        // pour bloquer des readers
int readcount=0;
int writecount=0;

sem_init(&wsem, 0, 1);
sem_init(&rsem, 0, 1);
```

Un *writer* utilise la variable `writecount` pour compter le nombre de *writers* qui veulent accéder à la base de données. Cette variable est protégée par `mutex_writecount`. Le sémaphore `wsem` est utilisé pour garantir qu'il n'y a qu'un seul *writer* qui peut accéder à un moment donné à la base de données. Cette utilisation est similaire à celle du sémaphore `db` dans la solution précédente.

```
/* Writer */
while(true)
{
  think_up_data();

  pthread_mutex_lock(&mutex_writecount);
    // section critique - writecount
    writecount=writecount+1;
    if(writecount==1) {
      // premier writer arrive
      sem_wait(&rsem);
    }
  pthread_mutex_unlock(&mutex_writecount);

  sem_wait(&wsem);
    // section critique, un seul writer à la fois
    write_database();
  sem_post(&wsem);

  pthread_mutex_lock(&mutex_writecount);
     // section critique - writecount
     writecount=writecount-1;
     if(writecount==0) {
       // départ du dernier writer
       sem_post(&rsem);
     }
  pthread_mutex_unlock(&mutex_writecount);
 }
```

Pour comprendre le reste du fonctionnement des *writers*, il faut analyser en parallèle le fonctionnement des *readers* car les deux types de threads interagissent de façon importante. Un *reader* utilise la variable `readcount` protégée par le `mutex_readcount` pour compter le nombre de *readers* en attente. Un *reader* utilise deux sémaphores. Le premier est `wsem` qui joue un rôle similaire au sémaphore `db` de la solution précédente. Le premier *reader* qui veut accéder à la base de données (`readcount==1`) effectue `sem_wait(&wsem)` pour garantir qu'il n'y aura pas de *writer* qui accède à la base de données pendant qu'il s'y trouve. Lorsque le dernier *reader* n'a plus besoin

d'accéder à la base de données (`readcount==0`), il libère les *writers* qui étaient potentiellement en attente en exécutant `sem_post(&wsem)`.

Le sémaphore `rsem` répond à un autre besoin. Il permet de bloquer les *readers* en attente lorsqu'un *writer* veut accéder à la base de données. En effet, le premier *writer* qui veut accéder à la base de données exécute `sem_wait(&rsem)`. Cela a pour effet de bloquer les nouveaux *readers* qui voudraient accéder à la base de données sur `sem_wait(&rsem)`. Ils ne seront débloqués que lorsque le dernier *writer* (`writecount==0`) quittera la base de données et exécutera `sem_post(&rsem)`. Lorsqu'aucun writer n'accède à la base de données, les readers peuvent facilement exécuter `sem_wait(&rsem)` qui sera rapidement suivi de `sem_post(&rsem)`.

```
/* Reader */
while(true)
{
  pthread_mutex_lock(&z);
    // exclusion mutuelle, un seul reader en attente sur rsem
    sem_wait(&rsem);

    pthread_mutex_lock(&mutex_readcount);
       // exclusion mutuelle, readercount
       readcount=readcount+1;
       if (readcount==1) {
          // arrivée du premier reader
          sem_wait(&wsem);
       }
  pthread_mutex_unlock(&mutex_readcount);
  sem_post(&rsem);  // libération du prochain reader
  pthread_mutex_unlock(&z);

  read_database();

  pthread_mutex_lock(&mutex_readcount);
     // exclusion mutuelle, readcount
     readcount=readcount-1;
     if(readcount==0) {
          // départ du dernier reader
          sem_post(&wsem);
     }
  pthread_mutex_unlock(&mutex_readcount);
  use_data_read();
}
```

Pour comprendre l'utilité du mutex `z`, il faut imaginer une solution dans laquelle il n'est pas utilisé. Dans ce cas, imaginons que plusieurs *readers* accèdent à la base de données et que deux *readers* et deux *writers* veulent y accéder. Le premier *reader* exécute `sem_wait(&rsem);`. Le premier *writer* va exécuter `sem_wait(&rsem);` et sera bloqué en attendant que le premier *reader* exécute `sem_post(&rsem);`. Le deuxième *writer* sera bloqué sur `pthread_mutex_lock(&mutex_writecount);`. Lorsque le premier *reader* exécute `pthread_mutex_unlock(&mutex_readcount);`, il permet au second *reader* de passer le mutex et d'exécuter `sem_wait(&rsem);`. Lorsque le premier *reader* exécute finalement `sem_post(&rsem);`, le système devra libérer un des threads en attente, c'est-à-dire le second *reader* ou le premier *writer*. Cette solution ne donne pas complètement la priorité aux *writers*. Le mutex `z` permet d'éviter ce problème en n'ayant qu'un seul *reader* à la fois qui peut exécuter la séquence `sem_wait(&rsem); ... sem_post(&rsem);`. Avec le mutex `z`, le second *reader* est nécessairement en attente sur le mutex `z` lorsque le premier *reader* exécute `sem_post(&rsem);`. Si un *writer* est en attente à ce moment, il sera nécessairement débloqué.

Note : Read-Write locks

Certaines implémentations de la librairie des threads POSIX contiennent des *Read-Write locks*. Ceux-ci constituent une API de plus haut niveau qui s'appuie sur des sémaphores pour résoudre le *problème des readers-writers*. Les fonctions de création et de suppression de ces locks sont : pthread_rwlock_init(3posix), pthread_rwlock_destroy(3posix). Les fonctions pthread_rwlock_rdlock(3posix) et pthread_rwlock_unlock(3posix) sont réservées aux readers tandis que les fonctions pthread_rwlock_wrlock(3posix) et pthread_rwlock_unlock(3posix) sont utilisables par les writers.

Des exemples d'utilisation de ces *Read-Write locks* peuvent être obtenus dans [Gove2011].

4.5 Compléments sur les threads POSIX

Il existe différentes implémentations des threads POSIX. Les mécanismes de coordination utilisables varient parfois d'une implémentation à l'autre. Dans les sections précédentes, nous nous sommes focalisés sur les fonctions principales qui sont en général bien implémentées. Une discussion plus détaillée des fonctions implémentées sous Linux peut se trouver dans [Kerrisk2010]. [Gove2011] présente de façon détaillée les mécanismes de coordination utilisables sous Linux, Windows et Oracle Solaris. [StevensRago2008] comprend également une description des threads POSIX mais présente des exemples sur des versions plus anciennes de Linux, FreeBSD, Solaris et MacOS.

Il reste cependant quelques concepts qu'il est utile de connaître lorsque l'on développe des programmes multi-threadés en langage C.

4.5.1 Variables `volatile`

Normalement, dans un programme C, lorsqu'une variable est définie, ses accès sont contrôlés entièrement par le compilateur. Si la variable est utilisée dans plusieurs calculs successifs, il peut être utile d'un point de vue des performances de stocker la valeur de cette variable dans un registre pendant au moins le temps correspondant à l'exécution de quelques instructions [11]. Cette optimisation peut éventuellement poser des difficultés dans certains programmes utilisant des threads puisqu'une variable peut être potentiellement modifiée ou lue par plusieurs threads simultanément.

Les premiers compilateurs C avaient pris en compte un problème similaire. Lorsqu'un programme ou un système d'exploitation interagit avec des dispositifs d'entrée-sortie, cela se fait parfois en permettant au dispositif d'écrire directement en mémoire à une adresse connue par le système d'exploitation. La valeur présente à cette adresse peut donc être modifiée par le dispositif d'entrée-sortie sans que le programme ne soit responsable de cette modification. Face à ce problème, les inventeurs du langage C ont introduit le qualificatif `volatile`. Lorsqu'une variable est `volatile`, cela indique au compilateur qu'il doit recharger la variable de la mémoire chaque fois qu'elle est utilisée.

Pour bien comprendre l'impact de ce qualificatif, il est intéressant d'analyser le code assembleur générée par un compilateur C dans l'exemple suivant.

```
int x=1;
int v[2];

void f(void ) {
  v[0]=x;
  v[1]=x;
}
```

Dans ce cas, la fonction `f` est traduite en la séquence d'instructions suivante :

```
f:
      movl    x, %eax
      movl    %eax, v
      movl    %eax, v+4
      ret
```

Si par contre la variable `x` est déclarée comme étant `volatile`, le compilateur ajoute une instruction `movl x, %eax` qui permet de recharger la valeur de `x` dans un registre avant la seconde utilisation.

11. Les premiers compilateurs C permettaient au programmeur de donner des indications au compilateur en faisant précéder les déclarations de certaines variables avec le qualificatif `register` [KernighanRitchie1998]. Ce qualificatif indiquait que la variable était utilisée fréquemment et que le compilateur devrait en placer le contenu dans un registre. Les compilateurs actuels sont nettement plus performants et ils sont capables de détecter quelles sont les variables qu'il faut placer dans un registre. Il est inutile de chercher à influencer le compilateur en utilisant le qualificatif `register`. Les compilateurs actuels, dont gcc(1) supportent de nombreuses options permettant d'optimiser les performances des programmes compilés. Certaines ont comme objectif d'accélérer l'exécution du programme, d'autres visent à réduire sa taille. Pour les programmes qui consomment beaucoup de temps CPU, il est utile d'activer l'optimisation du compilateur.

```
f:
     movl    x, %eax
     movl    %eax, v
     movl    x, %eax
     movl    %eax, v+4
     ret
```

Le qualificatif `volatile` force le compilateur à recharger la variable depuis la mémoire avant chaque utilisation. Ce qualificatif est utile lorsque le contenu stocké à une adresse mémoire peut être modifié par une autre source que le programme lui-même. C'est le cas dans les threads, mais marquer les variables partagées par des threads comme `volatile` ne suffit pas. Si ces variables sont modifiées par certains threads, il est nécessaire d'utiliser des *mutex* ou d'autres techniques de coordination pour réguler l'accès en ces variables partagées. En pratique, la documentation du programme devra spécifier quelles variables sont partagées entre les threads et la technique de coordination éventuelle qui est utilisée pour en réguler les accès. L'utilisation du qualificatif `volatile` permet de forcer le compilateur à recharger le contenu de la variable depuis la mémoire avant toute utilisation. C'est une règle de bonne pratique qu'il est utile de suivre. Il faut cependant noter que dans l'exemple ci-dessus, l'utilisation du qualificatif `volatile` augmente le nombre d'accès à la mémoire et peut donc dans certains cas réduire les performances.

4.5.2 Variables spécifiques à un thread

Dans un programme C séquentiel, on doit souvent combiner les variables globales, les variables locales et les arguments de fonctions. Lorsque le programme est découpé en threads, les variables globales restent utilisables, mais il faut faire attention aux problèmes d'accès concurrent. En pratique, il est parfois utile de pouvoir disposer dans chaque thread de variables qui tout en étant accessibles depuis toutes les fonctions du thread ne sont pas accessibles aux autres threads. Il y a différentes solutions pour résoudre ce problème.

Une première solution serait d'utiliser une zone mémoire qui est spécifique au thread et d'y placer par exemple une structure contenant toutes les variables auxquelles on souhaite pouvoir accéder depuis toutes les fonctions du thread. Cette zone mémoire pourrait être créée avant l'appel à pthread_create(3) et un pointeur vers cette zone pourrait être passé comme argument à la fonction qui démarre le thread. Malheureusement l'argument qui est passé à cette fonction n'est pas équivalent à une variable globale et n'est pas accessible à toutes les fonctions du thread.

Une deuxième solution serait d'avoir un tableau global qui contiendrait des pointeurs vers des zones de mémoires qui ont été allouées pour chaque thread. Chaque thread pourrait alors accéder à ce tableau sur base de son identifiant. Cette solution pourrait fonctionner si le nombre de threads est fixe et que les identifiants de threads sont des entiers croissants. Malheureusement la libraire threads POSIX ne fournit pas de tels identifiants croissants. Officiellement, la fonction pthread_self(3) retourne un identifiant unique d'un thread qui a été créé. Malheureusement cet identifiant est de type `pthread_t` et ne peut pas être utilisé comme index dans un tableau. Sous Linux, l'appel système non-standard gettid(2) retourne l'identifiant du thread, mais il ne peut pas non plus être utilisé comme index dans un tableau.

Pour résoudre ce problème, deux solutions sont possibles. La première combine une extension au langage C qui est supportée par gcc(1) avec la librairie threads POSIX. il s'agit du qualificatif `__thread` qui peut être utilisé avant une déclaration de variable. Lorsqu'il est utilisé dans la déclaration d'une variable globale, il indique au compilateur et à la libraire POSIX qu'une copie de cette variable doit être créée pour chaque thread. Cette variable est initialisée au démarrage du thread et est utilisable uniquement à l'intérieur de ce thread. Le programme ci-dessous illustre cette utilisation du qualificatif `__thread`.

```
#define LOOP 1000000
#define NTHREADS 4

__thread int count=0;
int global_count=0;

void *f( void* param) {
  for(int i=0;i<LOOP;i++) {
    count++;
    global_count=global_count-1;
```

```
  }
  printf("Valeurs : count=%d, global_count=%d\n",count, global_count);
  return(NULL);
}

int main (int argc, char *argv[])  {
  pthread_t threads[NTHREADS];
  int err;

  for(int i=0;i<NTHREADS;i++) {
    count=i;   // local au thread du programme principal
    err=pthread_create(&(threads[i]),NULL,&f,NULL);
    if(err!=0)
      error(err,"pthread_create");
  }

  for(int i=0;i<NTHREADS;i++) {
    err=pthread_join(threads[i],NULL);
    if(err!=0)
      error(err,"pthread_create");
  }

  return(EXIT_SUCCESS);
}
```

Lors de son exécution, ce programme affiche la sortie suivante sur *stdout*. Cette sortie illustre bien que les variables dont la déclaration est précédée du qualificatif `__thread` sont utilisables uniquement à l'intérieur d'un thread.

```
Valeurs : count=1000000, global_count=-870754
Valeurs : count=1000000, global_count=-880737
Valeurs : count=1000000, global_count=-916383
Valeurs : count=1000000, global_count=-923423
```

La seconde solution proposée par la librairie POSIX est plus complexe. Elle nécessite l'utilisation des fonctions pthread_key_create(3posix), pthread_setspecific(3posix), pthread_getspecific(3posix) et pthread_key_delete(3posix). Cette API est malheureusement plus difficile à utiliser que le qualificatif `__thread`, mais elle illustre ce qu'il se passe en pratique lorsque ce qualificatif est utilisé.

Pour avoir une variable accessible depuis toutes les fonctions d'un thread, il faut tout d'abord créer une clé qui identifie cette variable. Cette clé est de type `pthread_key_t` et c'est l'adresse de cette structure en mémoire qui sert d'identifiant pour la variable spécifique à chaque thread. Cette clé ne doit être créée qu'une seule fois. Cela peut se faire dans le programme qui lance les threads ou alors dans le premier thread lancé en utilisant la fonction pthread_once(3posix). Une clé est créée grâce à la fonction pthread_key_create(3posix). Cette fonction prend deux arguments. Le premier est un pointeur vers une structure de type `pthread_key_t`. Le second est la fonction optionnelle à appeler lorsque le thread utilisant la clé se termine.

Il faut noter que la fonction pthread_key_create(3posix) associe en pratique le pointeur `NULL` à la clé qui a été créée dans chaque thread. Le thread qui veut utiliser la variable correspondant à cette clé doit réserver la zone mémoire correspondante. Cela se fait en général en utilisant malloc(3) puis en appelant la fonction pthread_setspecific(3posix). Celle-ci prend deux arguments. Le premier est une clé de type `pthread_key_t` qui a été préalablement créée. Le second est un pointeur (de type `void *`) vers la zone mémoire correspondant à la variable spécifique. Une fois que le lien entre la clé et le pointeur a été fait, la fonction pthread_getspecific(3posix) peut être utilisée pour récupérer le pointeur depuis n'importe quelle fonction du thread. L'implémentation des fonctions pthread_setspecific(3posix) et pthread_getspecific(3posix) garantit que chaque thread aura sa variable qui lui est propre.

L'exemple ci-dessous illustre l'utilisation de cette API. Elle est nettement plus lourde à utiliser que le qualificatif `__thread`. Dans ce code, chaque thread démarre par la fonction `f`. Celle-ci crée une variable spécifique de type `int` qui joue le même rôle que la variable `__thread int count;` dans l'exemple précédent. La fonction `g` qui est appelée sans argument peut accéder à la zone mémoire créée en appelant `pthread_getspecific(count)`. Elle peut ensuite exécuter ses calculs en utilisant le pointeur

`count_ptr`. Avant de se terminer, la fonction `f` libère la zone mémoire qui avait été allouée par malloc(3). Une alternative à l'appel explicite à free(3) aurait été de passer `free` comme second argument à pthread_key_create(3posix) lors de la création de la clé `count`. En effet, ce second argument est la fonction à appeler à la fin du thread pour libérer la mémoire correspondant à cette clé.

```
#define LOOP 1000000
#define NTHREADS 4

pthread_key_t count;
int global_count=0;

void g(void ) {
  void * data=pthread_getspecific(count);
  if(data==NULL)
    error(-1,"pthread_getspecific");
  int *count_ptr=(int *)data;
  for(int i=0;i<LOOP;i++) {
    *count_ptr=*(count_ptr)+1;
    global_count=global_count-1;
  }
}

void *f( void* param) {
  int err;
  int *int_ptr=malloc(sizeof(int));
  *int_ptr=0;
  err=pthread_setspecific(count, (void *)int_ptr);
  if(err!=0)
    error(err,"pthread_setspecific");
  g();
  printf("Valeurs : count=%d, global_count=%d\n",*int_ptr, global_count);
  free(int_ptr);
  return(NULL);
}

int main (int argc, char *argv[])  {
  pthread_t threads[NTHREADS];
  int err;

  err=pthread_key_create(&(count),NULL);
  if(err!=0)
    error(err,"pthread_key_create");

  for(int i=0;i<NTHREADS;i++) {
    err=pthread_create(&(threads[i]),NULL,&f,NULL);
    if(err!=0)
      error(err,"pthread_create");
  }

  for(int i=0;i<NTHREADS;i++) {
    err=pthread_join(threads[i],NULL);
    if(err!=0)
      error(err,"pthread_create");
  }
  err=pthread_key_delete(count);
  if(err!=0)
    error(err,"pthread_key_delete");

  return(EXIT_SUCCESS);
}
```

En pratique, on préfèrera évidemment d'utiliser le qualificatif `__thread` à la place de l'API explicite lorsque c'est possible. Cependant, il ne faut pas oublier que lorsque ce qualificatif est utilisé, le compilateur doit introduire

dans le programme du code permettant de faire le même genre d'opérations que les fonctions explicites de la librairie.

4.5.3 Fonctions thread-safe

Dans un programme séquentiel, il n'y a qu'un thread d'exécution et de nombreux programmeurs, y compris ceux qui ont développé la librairie standard, utilisent cette hypothèse lors de l'écriture de fonctions. Lorsqu'un programme est découpé en threads, chaque fonction peut être appelée par plusieurs threads simultanément. Cette exécution simultanée d'une fonction peut poser des difficultés notamment lorsque la fonction utilise des variables globales ou des variables statiques.

Pour comprendre le problème, il est intéressant de comparer plusieurs implémentations d'une fonction simple. Considérons le problème de déterminer l'élément maximum d'une structure de données contenant des entiers. Si la structure de données est un tableau, une solution simple est de le parcourir entièrement pour déterminer l'élément maximum. C'est ce que fait la fonction `max_vector` dans le programme ci-dessous. Dans un programme purement séquentiel dans lequel le tableau peut être modifié de temps en temps, parcourir tout le tableau pour déterminer son maximum n'est pas nécessairement la solution la plus efficace. Une alternative est de mettre à jour la valeur du maximum chaque fois qu'un élément du tableau est modifié. Les fonctions `max_global` et `max_static` sont deux solutions possibles. Chacune de ces fonctions doit être appelée chaque fois qu'un élément du tableau est modifié. `max_global` stocke dans une variable globale la valeur actuelle du maximum du tableau et met à jour cette valeur à chaque appel. La fonction `max_static` fait de même en utilisant une variable statique. Ces deux solutions sont équivalentes et elles pourraient très bien être intégrées à une librairie utilisée par de nombreux programmes.

```
#include <stdint.h>
#define SIZE 10000

int g_max=INT32_MIN;
int v[SIZE];

int max_vector(int n, int *v) {
  int max=INT32_MIN;
  for(int i=0;i<n;i++) {
    if(v[i]>max)
      max=v[i];
  }
  return max;
}

int max_global(int *v) {
    if (*v>g_max) {
      g_max=*v;
    }
    return(g_max);
}

int max_static(int *v){
  static int s_max=INT32_MIN;
  if (*v>s_max) {
    s_max=*v;
  }
  return(s_max);
}
```

Considérons maintenant un programme découpé en plusieurs threads qui chacun maintient un tableau d'entiers dont il faut connaître le maximum. Ces tableaux d'entiers sont distincts et ne sont pas partagés entre les threads. La fonction `max_vector` peut être utilisée par chaque thread pour déterminer le maximum du tableau. Par contre, les fonctions `max_global` et `max_static` ne peuvent pas être utilisées. En effet, chacune de ces fonctions maintient *un* état (dans ce cas le maximum calculé) alors qu'elle peut être appelée par différents threads qui auraient chacun besoin d'un état qui leur est propre. Pour que ces fonctions soient utilisables, il faudrait que les variables `s_max` et `g_max` soient spécifiques à chaque thread.

En pratique, ce problème de l'accès concurrent à des fonctions se pose pour de nombreuses fonctions et notamment celles de la librairie standard. Lorsque l'on développe une fonction qui peut être réutilisée, il est important de s'assurer que cette fonction peut être exécutée par plusieurs threads simultanément sans que cela ne pose de problèmes à l'exécution.

Ce problème affecte certaines fonctions de la librairie standard et plusieurs d'entre elles ont dû être modifiées pour pouvoir supporter les threads. A titre d'exemple, considérons la fonction strerror(3). Cette fonction prend comme argument le numéro de l'erreur et retourne une chaîne de caractères décrivant cette erreur. Cette fonction ne peut pas être utilisée telle quelle par des threads qui pourraient l'appeler simultanément. Pour s'en convaincre, regardons une version simplifiée d'une implémentation de cette fonction [12]. Cette fonction utilise le tableau sys_errlist(3) qui contient les messages d'erreur associés aux principaux codes numériques d'erreur. Lorsque l'erreur est une erreur standard, tout se passe bien et la fonction retourne simplement un pointeur vers l'entrée du tableau `sys_errlist` correspondante. Par contre, si le code d'erreur n'est pas connu, un message est généré dans le tableau `buf[32]` qui est déclaré de façon statique. Si plusieurs threads exécutent `strerror`, ce sera le même tableau qui sera utilisé dans les différents threads. On pourrait remplacer le tableau statique par une allocation de zone mémoire faite via malloc(3), mais alors la zone mémoire créée risque de ne jamais être libérée par free(3) car l'utilisateur de strerror(3) ne doit pas libérer le pointeur qu'il a reçu, ce qui pose d'autres problèmes en pratique.

```
char * strerror (int errnoval)
{
 char * msg;
 static char buf[32];
 if ((errnoval < 0) || (errnoval >= sys_nerr))
   { // Out of range, just return NULL
     msg = NULL;
   }
 else if ((sys_errlist == NULL) || (sys_errlist[errnoval] == NULL))
   { // In range, but no sys_errlist or no entry at this index.
     sprintf (buf, "Error %d", errnoval);
     msg = buf;
   }
 else
   { // In range, and a valid message.  Just return the message.
     msg = (char *) sys_errlist[errnoval];
   }
 return (msg);
}
```

La fonction strerror_r(3) évite ce problème de tableau statique en utilisant trois arguments : le code d'erreur, un pointeur `char *` vers la zone devant stocker le message d'erreur et la taille de cette zone. Cela permet à strerror_r(3) d'utiliser une zone mémoire qui lui est passée par le thread qu'il appelle et garantit que chaque thread disposera de son message d'erreur. Voici une implémentation possible de strerror_r(3) [13].

```
strerror_r(int num, char *buf, size_t buflen)
{
  #define UPREFIX "Unknown error: %u"
  unsigned int errnum = num;
  int retval = 0;
  size_t slen;
  if (errnum < (unsigned int) sys_nerr) {
     slen = strlcpy(buf, sys_errlist[errnum], buflen);
  } else {
     slen = snprintf(buf, buflen, UPREFIX, errnum);
     retval = EINVAL;
  }
  if (slen >= buflen)
       retval = ERANGE;
```

12. Cette implémentation est adaptée de http://opensource.apple.com/source/gcc/gcc-926/libiberty/strerror.c et est dans le domaine public.
13. Cette implémentation est adaptée de https://www-asim.lip6.fr/trac/netbsdtsar/browser/vendor/netbsd/5/src/lib/libc/string/strerror_r.c?rev=2 et est *Copyright (c) 1988 Regents of the University of California.*

```
  return retval;
}
```

Lorsque l'on intègre des fonctions provenant de la librairie standard ou d'une autre librairie dans un programme découpé en threads, il est important de vérifier que les fonctions utilisées sont bien *thread-safe*. La page de manuel pthreads(7) liste les fonctions qui ne sont pas *thread-safe* dans la librairie standard.

4.6 Loi de Amdahl

En découpant un programme en threads, il est théoriquement possible d'améliorer les performances du programme en lui permettant d'exécuter plusieurs threads d'exécution simultanément. Dans certains cas, la découpe d'un programme en différents threads est naturelle et relativement facile à réaliser. Dans d'autres cas, elle est nettement plus compliquée. Pour s'en convaincre, il suffit de considérer un grand tableau contenant plusieurs centaines de millions de nombres. Considérons un programme simple qui doit trouver dans ce tableau quel est l'élément du tableau pour lequel l'application d'une fonction complexe `f` donne le résultat minimal. Une implémentation purement séquentielle se contenterait de parcourir l'entièreté du tableau et d'appliquer la fonction `f` à chacun des éléments. A la fin de son exécution, le programme retournera l'élément qui donne la valeur minimale. Un tel problème est très facile à découper en threads. Il suffit de découper le tableau en `N` sous-tableaux, de lancer un thread de calcul sur chaque sous-tableau et ensuite de fusionner les résultats de chaque thread.

Un autre problème est de trier le contenu d'un tel tableau dans l'ordre croissant. De nombreux algorithmes séquentiels de tri existent pour ordonner un tableau. La découpe de ce problème en thread est nettement moins évidente que dans le problème précédent et les algorithmes de tri adaptés à une utilisation dans plusieurs threads ne sont pas une simple extension des algorithmes séquentiels.

Dans les années 1960s, à l'époque des premières réflexions sur l'utilisation de plusieurs processeurs pour résoudre un problème, Gene Amdahl [Amdahl1967] a analysé quelles étaient les gains que l'on pouvait attendre de l'utilisation de plusieurs processeurs. Dans sa réflexion, il considère un programme `P` qui peut être découpé en deux parties :

- une partie purement séquentielle. Il s'agit par exemple de l'initialisation de l'algorithme utilisé, de la collecte des résultats, ...
- une partie qui est parallélisable. Il s'agit en général du coeur de l'algorithme.

Plus les opérations réalisées à l'intérieur d'un programme sont indépendantes entre elles, plus le programme est parallélisable et inversement. Pour Amdahl, si le temps d'exécution d'un programme séquentiel est T et qu'une fraction f de ce programme est parallélisable, alors le gain qui peut être obtenu de la parallélisation est $\frac{T}{T\times((1-f)+\frac{f}{N})} = \frac{1}{(1-f)+\frac{f}{N}}$ lorsque le programme est découpé en N threads. Cette formule, connue sous le nom de la *loi de Amdahl* fixe une limite théorique sur le gain que l'on peut obtenir en parallélisant un programme. La figure ci-dessous [14] illustre le gain théorique que l'on peut obtenir en parallélisant un programme en fonction du nombre de processeur et pour différentes fractions parallélisables.

La loi de Amdahl doit être considérée comme un maximum théorique qui est difficile d'atteindre. Elle suppose que la parallélisation est parfaite, c'est-à-dire que la création et la terminaison de threads n'ont pas de coût en terme de performance. En pratique, c'est loin d'être le cas et il peut être difficile d'estimer a priori le gain qu'une parallélisation permettra d'obtenir. En pratique, avant de découper un programme séquentiel en threads, il est important de bien identifier la partie séquentielle et la partie parallélisable du programme. Si la partie séquentielle est trop importante, le gain dû à la parallélisation risque d'être faible. Si par contre la partie purement séquentielle est faible, il est possible d'obtenir théoriquement des gains élevés. Le tout sera de trouver des solutions efficaces qui permettront aux threads de fonctionner le plus indépendamment possible.

En pratique, avant de s'attaquer à la découpe d'un programme séquentiel en threads, il est important de bien comprendre quelles sont les parties du programme qui sont les plus consommatrices de temps CPU. Ce seront souvent les boucles ou les grandes structures de données. Si le programme séquentiel existe, il est utile d'analyser son exécution avec des outils de profiling tels que gprof(1) [Graham+1982] ou oprofile. Un profiler est un logiciel qui permet d'analyser l'exécution d'un autre logiciel de façon à pouvoir déterminer notamment quelles sont les fonctions ou parties du programmes les plus exécutées. Ces parties de programme sont celles sur lesquelles l'effort de paralélisation devra porter en pratique.

14. Source : http://en.wikipedia.org/wiki/Amdahl's_law

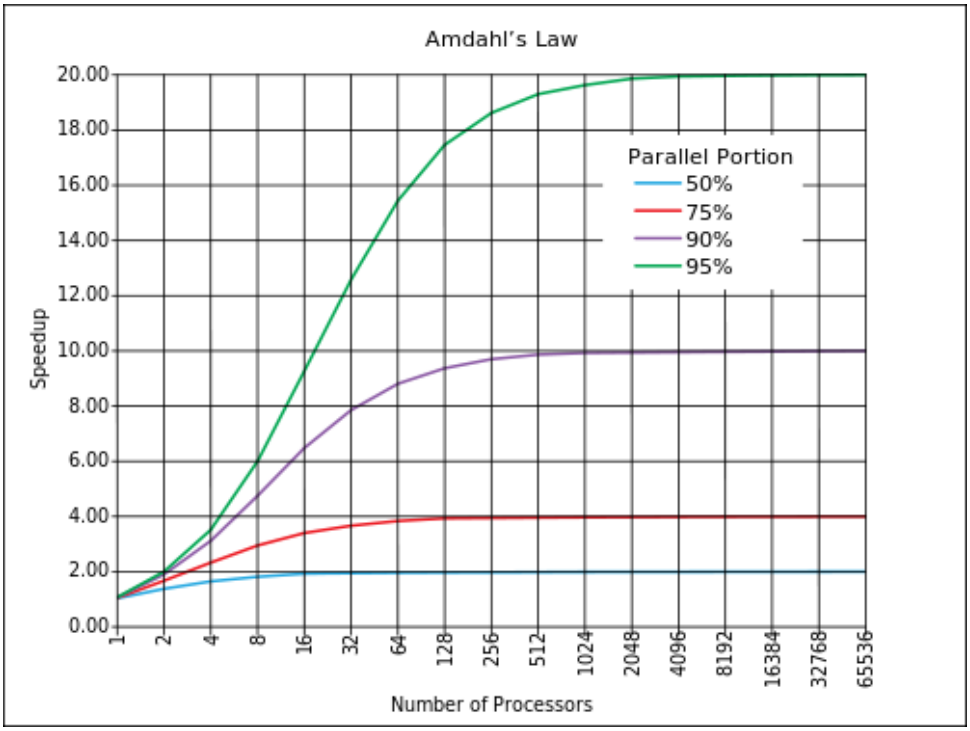


FIGURE 4.8 – Loi de Amdahl (source wikipedia)

Dans un programme découpé en threads, toute utilisation de fonctions de coordination comme des sémaphores ou des mutex, bien qu'elle soit nécessaire pour la correction du programme, risque d'avoir un impact négatif sur les performances. Pour s'en convaincre, il est intéressant de réfléchir au problème des producteurs-consommateurs. Il correspond à de nombreux programmes réels. Les performances d'une implémentation du problème des producteurs consommateurs dépendront fortement de la taille du buffer entre les producteurs et les consommateurs et de leur nombre et/ou vitesses relatives. Idéalement, il faudrait que le buffer soit en moyenne rempli à moitié. De cette façon, chaque producteur pourra déposer de l'information dans le buffer et chaque consommateur pourra en retirer. Si le buffer est souvent vide, cela indique que les consommateurs sont plus rapides que les producteurs. Ces consommateurs risquent d'être inutilement bloqués, ce qui affectera les performances. Il en va de même si le buffer était plein. Dans ce cas, les producteurs seraient souvent bloqués.

4.7 Les processus

Un système d'exploitation multitâche et multi-utilisateurs tel que Unix ou Linux permet d'exécuter de nombreux programmes simultanément. Sous Unix, les programmes sont exécutés sous la forme de *processus*. Un processus peut être défini comme étant une instance de programme qui est en train d'être exécutée sur un ou plusieurs processeurs sous le contrôle d'un système d'exploitation. Un processus comprend donc un ensemble d'instructions pour le processeur, mais aussi des données qui sont stockées en mémoire et un contexte (si le processus utilise un seul thread d'exécution, plusieurs contextes sinon). En outre, le système d'exploitation maintient un certain nombre de structures de données qui sont nécessaires au bon fonctionnement du processus. Ces structures de données sont créées au démarrage du processus, mises à jour durant la vie du processus et supprimées lorsque le processus se termine.

4.7.1 Les librairies

Lorsqu'un programme s'exécute à l'intérieur d'un processus, il exécute des instructions qui ont différentes *origines*. Il y a bien entendu les instructions qui proviennent du code source du programme qui a été converti en assembleur par le compilateur. Ces instructions correspondent au code source développé par le programmeur. Il s'agit notamment de toutes les opérations mathématiques et logiques, les boucles et les appels de fonctions internes

au programme. Comme nous l'avons vu précédemment, ces instructions peuvent provenir d'un seul module ou de plusieurs modules. Dans ce dernier cas, le linker intervient pour combiner différents modules en un exécutable complet.

A côté des instructions qui correspondent aux lignes de code écrites par le développeur du programme, un processus va également exécuter de nombreuses fonctions qui font partie d'une des librairies standard du système. Tout environnement de développement comprend des librairies qui permettent de faciliter le travail des programmeurs en leur fournissant des fonctions permettant de résoudre de nombreux problèmes classiques. Un système d'exploitation tel que Unix ou Linux contient de nombreuses librairies de ce type. Nous avons déjà eu l'occasion de discuter des fonctions provenant de la librairie standard comme printf(3) ou malloc(3) et celles de la librairie pthreads(7). Ce ne sont que deux librairies parmi d'autres. Un système Linux contient plusieurs centaines de librairies utilisables par le programmeur.

A titre d'exemple, considérons la librairie math.h(7posix). Cette librairie contient de nombreuses fonctions mathématiques. Pour les utiliser dans un programme, il faut non seulement y inclure le fichier header math.h qui contient les prototypes et constantes utilisées par la librairie, mais aussi indiquer au linker que l'exécutable doit être lié avec la librairie math.h(7posix). Cela se fait en utilisant le flag `-l` de gcc(1).

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


int main (int argc, char *argv[])  {
  double n1=1.0;
  double n2=-3.14;
  printf("Maximum : %f\n",fmax(n1,n2));

  return(EXIT_SUCCESS);
}
```

Le programme `/Threads/S8-src/math.c` ci-dessus doit être compilé en utilisant la commande `gcc -Wall -Werror math.c -o math -lm`. Le paramètre `-lm` indique au compilateur qu'il doit charger la librairie `m`. Cette librairie, est une des librairies standard du système, elle réside généralement dans le répertoire `/usr/lib`[15]. En pratique, gcc(1) charge automatiquement la librairie `C` standard lors de la compilation de tout programme. Cela revient à utiliser implicitement le paramètre `-lc`.

Lors de l'utilisation de telles librairies, on s'attendrait à ce que toutes les instructions correspondant aux fonctions de la librairie utilisée soient présentes à l'intérieur de l'exécutable. En pratique, ce n'est pas exactement le cas. Même si notre programme d'exemple utilise fmax(3) de la librairie math.h(7posix) et printf(3) de la librairie standard, son exécutable ne contient que quelques milliers d'octets.

```
$ ls -l math*
-rwxr-xr-x 1 obo stafinfo 6764 Mar 15  2012 math
-rw-r--r-- 1 obo stafinfo  373 Mar 15  2012 math.c
```

Une analyse plus détaillée de l'exécutable avec objdump(1) révèle que si l'exécutable contient bien des appels à ces fonctions, leur code n'y est pas entièrement inclus.

```
$gcc -g -lm math.c -o math
$objdump -S -d math
math:     file format elf64-x86-64
...
0000000000400468 <fmax@plt>:
400468:      ff 25 fa 04 20 00       jmpq   *0x2004fa(%rip)        # 600968 <
_GLOBAL_OFFSET_TABLE_+0x28>
40046e:      68 02 00 00 00          pushq  $0x2
400473:      e9 c0 ff ff ff          jmpq   400438 <_init+0x18>
...
```

15. Par défaut, gcc(1) cherche après les librairies spécifiées dans les répertoires de librairies standards, mais aussi dans les répertoires listés dans la variable d'environnement `LD_LIBRARY_PATH`. Il est également possible de spécifier des répertoires supplémentaires contenant les librairies avec l'argument `-L` de gcc(1).

```
 #include <stdio.h>
 #include <stdlib.h>
 #include <math.h>
 int main (int argc, char *argv[])  {
400564:      55                    push   %rbp
400565:      48 89 e5              mov    %rsp,%rbp
400568:      48 83 ec 20           sub    $0x20,%rsp
40056c:      89 7d ec              mov    %edi,-0x14(%rbp)
40056f:      48 89 75 e0           mov    %rsi,-0x20(%rbp)
 double n1=1.0;
400573:      48 b8 00 00 00 00 00  mov    $0x3ff0000000000000,%rax
40057a:      00 f0 3f
40057d:      48 89 45 f0           mov    %rax,-0x10(%rbp)
 double n2=-3.14;
400581:      48 b8 1f 85 eb 51 b8  mov    $0xc0091eb851eb851f,%rax
400588:      1e 09 c0
40058b:      48 89 45 f8           mov    %rax,-0x8(%rbp)
 printf("Maximum : %f\n",fmax(n1,n2));
40058f:      f2 0f 10 4d f8        movsd  -0x8(%rbp),%xmm1
400594:      f2 0f 10 45 f0        movsd  -0x10(%rbp),%xmm0
400599:      e8 ca fe ff ff        callq  400468 <fmax@plt>
40059e:      b8 b8 06 40 00        mov    $0x4006b8,%eax
4005a3:      48 89 c7              mov    %rax,%rdi
4005a6:      b8 01 00 00 00        mov    $0x1,%eax
4005ab:      e8 98 fe ff ff        callq  400448 <printf@plt>
 return(EXIT_SUCCESS);
4005b0:      b8 00 00 00 00        mov    $0x0,%eax
 }
```

La taille réduite des exécutables sous Linux et de nombreuses variantes de Unix s'explique par l'utilisation de librairies partagées. Un programme peut utiliser deux types de librairies : des librairies statiques et des librairies partagées. Une *librairie statique* (ou *static library* en anglais) est une librairie de fonctions qui est intégrée directement avec le programme. Elle fait entièrement partie de l'exécutable. C'est la première solution pour intégrer des librairies dans un programme. Son avantage principal est que l'exécutable est complet et comprend toutes les instructions qui sont nécessaires au fonctionnement du programme. Malheureusement, tous les programmes qui utilisent des fonctions d'une librairie courante, comme par exemple la librairie standard, doivent inclure le code relatif à toutes les fonctions qu'ils utilisent. Sachant que chaque programme ou presque utilise des fonctions comme printf(3), cela conduit à sauvegarder de très nombreuses copies du même code. Ce problème peut être résolu en utilisant des librairies partagées [16] . Une *librairie partagée* (ou *shared library* en anglais) est un ensemble de fonctions qui peuvent être appelées par un programme mais sont stockées dans un seul fichier sur disque. Ce fichier unique est utilisé automatiquement par tous les programmes qui utilisent des fonctions de la librairie.

Il est parfois intéressant de pouvoir créer une librairie qui peut être liée de façon statique avec des programmes, par exemple lorsque ceux-ci doivent être exécutés sur d'autres ordinateurs que ceux sur lesquels ils ont été compilés. A titre d'illustration, considérons une librairie minuscule contenant une seule fonction `imax` qui calcule le maximum entre deux entiers. L'implémentation de cette fonction est très simple.

```
int imax(int i, int j) {
  return ((i>j) ? i : j);

}
```

Cette fonction est déclarée dans le fichier header `imax.h` et peut être utilisée dans un programme comme ci-dessous.

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

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

16. Dans certains cas, on parle également de librairies dynamiques car ces librairies sont chargées dynamiquement à l'exécution du programme.

```
  int n1=1;
  int n2=-3;
  printf("Maximum : %d\n",imax(n1,n2));

  return(EXIT_SUCCESS);
}
```

En pratique, la construction d'une librairie se fait en deux étapes principales. Tout d'abord, il faut compiler les fichiers objet correspondant aux différents modules de la librairie. Cela peut se faire avec gcc(1) comme pour un programme C classique. Ensuite, il faut regrouper les différents modules dans une archive qui constituera la librairie qui peut être utilisée par des programmes. Par convention, toutes les librairies ont un nom qui commence par `lib` et se termine par l'extension `.a`. Sous Linux, cette opération est réalisée par l'utilitaire ar(1). La page de manuel de ar(1) décrit plus en détails son utilisation. En pratique, les opérations les plus fréquentes avec ar(1) sont :

— ajout d'un module objet à une librairie : `ar r libname.a module.o`
— suppression d'un module objet d'une librairie : `ar d libname.a module.o`

Il est aussi possible de lister le contenu de la librairie `libname.a` avec la commande `ar tv libname.a`.

L'archive contenant la librairie peut être liée en utilisant le linker à n'importe quel programme qui en utilise une ou plusieurs fonctions. Le linker de gcc(1) peut effectuer cette opération comme illustré par le `Makefile` ci-dessous. Il faut noter que l'argument `--static` permet de forcer le compilateur à inclure le code de la librairie dans l'exécutable.

```
#
# Makefile for library imax and imath
#

GCC        = gcc
AR         = ar
ARFLAGS = -cvq
CFLAGS = -Wall -std=c99 -g -c
LDFLAGS = --static -g

all: imath

imax.o: imax.c
        @echo compiling imax
        $(GCC) $(CFLAGS) imax.c

libimax.a: imax.o
        @echo building libimax
        $(AR) $(ARFLAGS) libimax.a imax.o

imath.o: imath.c imax.h
        @echo compiling imath.o
        $(GCC) $(CFLAGS) imath.c

imath: imath.o libimax.a
        @echo building imath
        $(GCC) $(LDFLAGS)  -o imath libimax.a imath.o

clean:
        rm imath libimax.a imax.o imath.o
```

Ce `Makefile` est un petit peu plus long que ceux que nous avons utilisés jusque maintenant. Il illustre une structure courante pour de nombreux fichiers `Makefile`. La première partie définit des constantes qui sont utilisées dans le reste du `Makefile`. Il s'agit tout d'abord du compilateur et du programme de construction de librairies qui sont utilisés. Définir ces programmes comme des constantes dans le `Makefile` permet de facilement en changer lorsque c'est nécessaire. Ensuite, trois constantes sont définies avec les arguments de base du compilateur et de `ar`. A nouveau, définir ces constantes une fois pour toutes facilite leur modification. Ensuite, la première cible est la cible `all:`. Comme c'est la première, c'est la cible par défaut qui sera utilisée lorsque make(1) est appelé sans argument. Elle dépend de l'exécutable `imath` qui est une des cibles du `Makefile`. La cible `clean:` permet

d'effacer les fichiers objet et exécutables construites par le `Makefile`. Il est utile d'avoir une telle cible lorsque l'on doit diffuser un projet en C ou le rendre dans le cadre d'un cours. Enfin, les autres cibles correspondent aux fichiers objet, à la librairie et à l'exécutable qui sont construits. La commande `@echo` affiche ses arguments sur la sortie standard. Enfin, la chaîne de caractères `$(GCC)` est remplacée par le constante définie au début du fichier. Des compléments d'information sur make(1) peuvent être obtenus dans divers documents dont make(1), [Mecklenburg+2004] ou [GNUMake].

Lorsqu'un programme est compilé de façon à utiliser une librairie dynamique, c'est le système d'exploitation qui analyse le programme lors de son chargement et intègre automatiquement les fonctions des librairies qui sont nécessaires à son exécution. L'entête de l'exécutable contient de l'information générée par le compilateur qui permet de localiser les librairies dynamiques qui doivent être intégrées de cette façon. L'utilitaire ldd(1) permet de visualiser quelles sont les librairies partagées utilisées par un programme.

```
$ ldd imath
linux-vdso.so.1 =>  (0x00007fffe41ff000)
libc.so.6 => /lib64/libc.so.6 (0x0000003eb2400000)
/lib64/ld-linux-x86-64.so.2 (0x0000003eb2000000)
```

4.7.2 Appels Système

Outre l'utilisation de fonctions de librairies, les programmes doivent interagir avec le système d'exploitation. Un système d'exploitation tel que Unix comprend à la fois des utilitaires comme grep(1), ls(1), ... qui sont directement exécutables depuis le shell et un noyau ou *kernel* en anglais. Le *kernel* contient les fonctions de base du système d'exploitation qui lui permettent à la fois d'interagir avec le matériel mais aussi de gérer les processus des utilisateurs. En pratique, le kernel peut être vu comme étant un programme spécial qui est toujours présent en mémoire. Parmi l'ensemble des fonctions contenues dans le *kernel*, il y en a un petit nombre, typiquement de quelques dizaines à quelques centaines, qui sont utilisables par les processus lancés par les utilisateurs. Ce sont les appels système. Un *appel système* est une fonction du *kernel* qui peut être appelée par n'importe quel processus. Comme nous l'avons vu lorsque nous avons décrit le fonctionnement du langage d'assemblage, l'exécution d'une fonction dans un programme comprend plusieurs étapes :

1. Placer les arguments de la fonction à un endroit (la pile) où la fonction peut y accéder
2. Sauvegarder sur la pile l'adresse de retour
3. Modifier le registre `%eip` de façon à ce que la prochaine instruction à exécuter soit celle de la fonction à exécuter
4. La fonction récupère ses arguments (sur la pile) et réalise son calcul
5. La fonction sauve son résultat à un endroit (`%eax`) convenu avec la fonction appelante
6. La fonction récupère l'adresse de retour sur la pile et modifie `%eip` de façon à retourner à la fonction appelante

L'exécution d'un appel système comprend les mêmes étapes avec une différence importante c'est que le flux d'exécution des instructions doit passer du programme utilisateur au noyau du système d'exploitation. Pour comprendre le fonctionnement et l'exécution d'un appel système, il est utile d'analyser les six points mentionnés ci-dessus.

Le premier problème à résoudre pour exécuter un appel système est de pouvoir placer les arguments de l'appel système dans un endroit auquel le *kernel* pourra facilement accéder. Il existe de nombreux appels systèmes avec différents arguments. La liste complète des appels systèmes est reprise dans la page de manuel syscalls(2). La table ci-dessous illustre quelques appels systèmes et leurs arguments.

Appel système	Arguments
getpid(2)	`void`
fork(2)	`void`
read(2)	`int fildes, void *buf, size_t nbyte`
kill(2)	`pid_t pid, int sig`
brk(2)	`const void *addr`

Sous Linux, les arguments d'un appel système sont placés par convention dans des registres. Sur [IA32], le premier argument est placé dans le registre `%ebx`, le second dans `%ecx`, ... Le *kernel* peut donc facilement récupérer les arguments d'un appel système en lisant le contenu des registres.

Le second problème à résoudre est celui de l'adresse de retour. Celle-ci est automatiquement sauvegardée lors de l'exécution de l'instruction qui fait appel au kernel, tout comme l'instruction `calll` sauvegarde directement l'adresse de retour d'une fonction appelée sur la pile.

Le troisième problème à résoudre est de passer de l'exécution du processus utilisateur à l'exécution du *kernel*. Les processeurs actuels peuvent fonctionner dans au minimum deux modes : le *mode utilisateur* et le *mode protégé*. Lorsque le processeur fonctionne en mode protégé, toutes les instructions du processeur et toutes les adresses mémoire sont utilisables. Lorsqu'il fonctionne en mode utilisateur, quelques instructions spécifiques de manipulation du matériel et certaines adresses mémoire ne sont pas utilisables. Cette division en deux modes de fonctionnement permet d'avoir une séparation claire entre le système d'exploitation et les processus lancés par les utilisateurs. Le noyau du système d'exploitation s'exécute en mode protégé et peut donc utiliser entièrement le processeur et les dispositifs matériels de l'ordinateur. Les processus utilisateurs par contre s'exécutent en mode utilisateur. Ils ne peuvent donc pas directement exécuter les instructions permettant une interaction avec des dispositifs matériel. Cette interaction doit passer par le noyau du système d'exploitation qui sert de médiateur et vérifie la validité des demandes faites par un processus utilisateur.

Les transitions entre les modes protégé et utilisateur sont importantes car elles rythment le fonctionnement du système d'exploitation. Lorsque l'ordinateur démarre, le processeur est placé en mode protégé et le *kernel* se charge. Il initialise différentes structures de données et lance init(8) le premier processus du système. Dès que init(8) a été lancé, le processeur passe en mode utilisateur et exécute les instructions de ce processus. Après cette phase de démarrage, les instructions du *kernel* seront exécutées lorsque soit une interruption matérielle surviendra ou qu'un processus utilisateur exécutera un appel système. L'interruption matérielle place automatiquement le processeur en mode protégé et le *kernel* exécute la routine de traitement d'interruption correspondant à l'interruption qui est apparue. Un appel système démarre par l'exécution d'une instruction spéciale (parfois appelée interruption logicielle) qui place le processeur en mode protégé et puis démarre l'exécution d'une instruction placée à une adresse spéciale en mémoire. Sur certains processeurs de la famille [IA32], l'instruction `int 0x80` permet ce passage du mode utilisateur au mode protégé. Sur d'autres processeurs, c'est l'instruction `syscall` qui joue ce rôle. L'exécution de cette instruction est la seule possibilité pour un programme d'exécuter des instructions du *kernel*. En pratique, cette instruction fait passer le processeur en mode protégé et démarre l'exécution d'une routine du *kernel*. Cette routine commence par sauvegarder le contexte du processus qui exécute l'appel système demandé. Chaque appel système est identifié par un nombre entier et le *kernel* contient une table avec pour chaque appel système l'adresse de la fonction à exécuter pour cet appel système. En pratique, le numéro de l'appel système à exécuter est placé par le processus appelant dans le registre `%eax`.

L'appel système peut donc s'exécuter en utilisant les arguments qui se trouvent dans les différents registres. Lorsque l'appel système se termine, le résultat est placé dans le registre `%eax` et une instruction spéciale permet de retourner en mode utilisateur et d'exécuter dans le processus appelant l'instruction qui suit celle qui a provoqué l'exécution de l'appel système. Si l'appel système a échoué, le *kernel* doit aussi mettre à jour le contenu de `errno` avant de retourner au processus appelant.

Ces opérations sont importantes pour comprendre le fonctionnement d'un système informatique et la différence entre un appel système et une fonction de la librairie. En pratique, la librairie cache cette complexité au programmeur en lui permettant d'utiliser des fonctions de plus haut niveau [17] . Cependant, il faut être conscient que ces fonctions s'appuient elles-même sur des appels systèmes pour s'exécuter. Ainsi par exemple, la fonction printf(3) utilise l'appel système write(2) pour écrire sur la sortie standard. La commande strace(1) permet de tracer l'ensemble des appels systèmes faits par un processus. A titre d'exemple, voici les appels systèmes effectués par le programme `imath` présenté plus haut.

```
$ strace ./imath
execve("./imath", ["./imath"], [/* 34 vars */]) = 0
uname({sys="Linux", node="baluran.info.ucl.ac.be", ...}) = 0
brk(0)                                  = 0xa31000
brk(0xa32180)                           = 0xa32180
arch_prctl(ARCH_SET_FS, 0xa31860)       = 0
brk(0xa53180)                           = 0xa53180
brk(0xa54000)                           = 0xa54000
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)  = 0x7f1f8fd44000
write(1, "Maximum : 1\n", 12Maximum : 1
```

17. En pratique, il correspond une fonction de la librairie à chaque appel système. Cette fonction a le même nom que l'appel système et les mêmes arguments et fait appel à syscall(2) pour l'exécution de l'appel système.

```
)            = 12
exit_group(0)                           = ?
```

4.7.3 Création d'un processus

Pour comprendre le fonctionnement de Unix, il est utile d'analyser plus en détails toutes les opérations qui sont effectuées à chaque fois que l'on lance un programme depuis un shell tel que bash(1). Considérons l'exécution de la commande `/bin/true` depuis le shell.

Schématiquement, l'exécution de ce programme se déroule comme suit. Le shell va d'abord localiser [18] l'exécutable `/bin/true` qui est stocké dans le système de fichiers. Ensuite, il va créer un processus et y exécuter l'exécutable. Le shell va ensuite attendre la fin de l'exécution du programme `true` et récupérer sa valeur de retour (retournée par exit(2)) pour ensuite poursuivre son exécution.

Comme nous l'avons expliqué plus haut, le *kernel* Linux gère l'ensemble des processus qui sont utilisés à un moment. Il intervient pour toutes les opérations de création et de fin d'un processus. La création d'un processus est un événément important dans un système d'exploitation. Elle permet notamment l'exécution de programmes. Ces opérations nécessitent une interaction avec le *kernel* et se font donc en utilisant des appels systèmes. Avant d'analyser en détails comment Linux supporte précisément la création de processus, il est intéressant de réfléchir aux opérations qui doivent être effectuées lors de l'exécution d'un programme. Considérons par exemple un utilisateur qui exécute la commande `/usr/bin/expr 1 + 2` depuis un shell bash(1) interactif. Pour exécuter cette commande, il va falloir exécuter un nouveau processus contenant les instructions assembleur se trouvant dans l'exécutable `/usr/bin/expr`, lui passer les arguments `1 + 2`, l'exécuter, récupérer sa valeur de retour et la retourner au shell qui pourra l'utiliser et poursuivre son exécution.

Les designers de Unix ont choisi de construire un appel système pour chacune de ces opérations. Le premier est l'appel système fork(2). C'est l'appel système qui permet de créer un processus. Schématiquement, cet appel système crée une copie complète du processus qui l'a exécuté. Après exécution de fork(2), il y a deux copies du même processus en mémoire. Le processus qui a exécuté fork(2) est considéré comme étant le *processus père* tandis que celui qui a été créé par l'exécution de fork(2) est le *processus fils*.

```
#include <unistd.h>

pid_t fork(void);
```

L'appel système fork(2) est atypique car il est exécuté par un processus mais provoque la création d'un second processus qui est identique au premier. Après l'exécution de l'appel système fork(2), il y a donc deux séquences d'instructions qui vont s'exécuter, l'une dans le processus père et l'autre dans le processus fils. Le processus fils démarre son exécution à la récupération du résultat de l'appel système fork(2) effectué par son père. Le processus père et le processus fils récupèrent une valeur de retour différente pour cet appel système. Cette valeur de retour est d'ailleurs la seule façon de distinguer le *processus père* du *processus fils* lorsque celui-ci démarre.

- l'appel système fork(2) retourne la valeur `-1` en cas d'erreur et met à jour la variable `errno`. En cas d'erreur, aucun processus n'est créé.
- l'appel système fork(2) retourne la valeur `0` dans le processus fils.
- l'appel système fork(2) retourne une valeur positive dans le processus père. Cette valeur est l'identifiant du processus fils créé.

Pour bien comprendre le fonctionnement de fork(2), analysons l'exemple `/Threads/S8-src/fork.c` ci-dessous :

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

int g=0; // segment données

int main (int argc, char *argv[])  {
  int l=1252; // sur la pile
```

18. La variable d'environnement `PATH` contient la liste des répertoires que le shell parcoure afin de localiser un exécutable à lancer lorsque l'utilisateur ne fournit par le chemin complet de l'exécutable à lancer.

```
  int *m;      // sur le heap
  m=(int *) malloc(sizeof(int));
  *m=-1;

  pid_t pid;

  pid=fork();

  if (pid==-1) {
    // erreur à l'exécution de fork
    perror("fork");
    exit(EXIT_FAILURE);
  }
  // pas d'erreur
  if (pid==0) {
    // processus fils
    l++;
    g++;
    *m=17;
    printf("Dans le processus fils g=%d, l=%d et *m=%d\n",g,l,*m);
    free(m);
    return(EXIT_SUCCESS);
  }
  else {
    // processus père
    sleep(2);
    printf("Dans le processus père g=%d, l=%d et *m=%d\n",g,l,*m);
    free(m);
    // ...
    return(EXIT_SUCCESS);
  }
}
```

Lors de son exécution, ce programme affiche les deux lignes suivantes sur sa sortie standard :

```
Dans le processus fils g=1, l=1253 et *m=17
Dans le processus père g=0, l=1252 et *m=-1
```

Lors de l'exécution de ce programme, deux variables sont initialisées en mémoire. La variable globale `g` est initialisée à la valeur `0` tandis que le variable locale `l` est initialisée à la valeur `1252`. malloc(3) est utilisé pour réserver une zone mémoire sur le *heap* et son contenu est initialisé à `-1`. Lorsque le processus père fait appel à fork(2) le noyau du système d'exploitation crée une copie identique à celui-ci en mémoire. Cette copie contient tous les segments du processus père (code, données, heap et stack) dans l'état exact dans lequel ils étaient au moment de l'exécution de l'appel système fork(2). Le contexte du processus père est copié et devient le contexte du processus fils. A cet instant, les deux processus sont complètement identiques à l'exception de certaines données qui sont maintenues par le système d'exploitation, comme l'identifiant de processus. Chaque processus qui s'exécute sur un système Unix a un identifiant unique et est retourné par l'appel système getpid(2). Le processus père et le processus fils ont un identifiant différent.

Les deux processus vont se différencier dès la fin de l'exécution de l'appel système fork(2). Comme tout appel système, fork(2) place sa valeur de retour dans le registre `%eax`. Comme indiqué plus haut, cette valeur sera positive dans le processus père. Celui-ci exécute `sleep(2);` et reste donc bloqué pendant deux secondes avant d'afficher un message sur sa sortie standard. Le processus fils de son côté incrémente les variables `l` et `g` et modifie la zone mémoire pointée par `*m` puis affiche leur contenu sur sa sortie standard puis se termine.

L'exécution de ce programme illustre bien que le processus fils démarre avec une copie du processus père lorsque l'appel système fork(2) se termine. Le processus fils peut modifier les variables qui ont été initialisées par le processus mais ces modifications n'ont aucun impact sur les variables utilisées dans le processus père. Même si le processus père et le processus fils sont identiques au moment de la création du processus fils, ils sont complètement indépendants par après. C'est une différence importante avec les threads. Contrairement à ce qu'il se passe avec les threads, un processus père et un processus fils ne partagent ni le segment de données, ni le heap ni le stack. Ces

zones mémoires ne peuvent pas être utilisées directement pour permettre à un processus père de communiquer avec son fils.

Note : Quel est le processus qui s'exécute en premier après fork(2) ?

Après l'exécution de l'appel système fork(2) et la création du processus fils, le *kernel* se trouve face à deux processus qui sont dans l'état *Ready*. Si il y a deux processeurs libres, le *kernel* pourra les démarrer quasi simultanément. Par contre, si un seul processeur est disponible, le *kernel* devra exécuter l'un des deux processus en premier. En pratique, rien ne permet de contrôler si le *kernel* commencera d'abord l'exécution du processus père ou l'exécution du processus fils. Tout programme utilisant fork(2) doit pouvoir fonctionner correctement quel que soit l'ordre d'exécution des processus père et fils.

Le *kernel* gère les processus et attribue un identifiant à chaque processus. Le type `pid_t` est utilisé pour les identifiants de processus sous Unix. Ce type correspond à un nombre entier généralement non-signé. Le nombre maximum de processus qui peuvent être lancés sur un système Linux est un des paramètres fixés à la compilation ou au démarrage du kernel. L'appel système getpid(2) retourne l'identifiant du processus courant tandis que l'appel système getppid(2) retourne l'identifiant du processus père.

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


int main (int argc, char *argv[])  {
  int pid=getpid();
  int ppid=getppid();
  printf("Processus %d, parent:%d\n",pid,ppid);

  return(EXIT_SUCCESS);
}
```

Après l'exécution de fork(2) le processus père et le processus fils ont un identifiant de processus différent mais ils partagent certaines ressources qui sont gérées par le *kernel*. C'est le cas notamment des flux standard *stdin*, *stdout* et *stderr*. Lorsque le *kernel* crée un processus fils, il conserve la même sortie standard que le processus père. C'est ce qui nous permet de visualiser le résultat de l'exemple précédent. Cependant, le processus père et le processus fils sont en concurrence pour écrire sur la sortie standard. Si aucune précaution n'est prise, ces deux processus risquent d'écrire de façon désordonnée sur la sortie standard.

Pour mieux comprendre le problème, analysons l'exécution du programme ci-dessous. Il crée un processus fils puis le père et le fils écrivent sur *stdout*.

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


void output(char c) {
  printf("Processus : %d\n",getpid());
  srand(getpid()+time(NULL));
  for(int i=0;i<60;i++) {
    putchar(c);
    int err=usleep((unsigned int) (rand()%10000));
    if(err<0) {
      perror("usleep");
      exit(EXIT_FAILURE);
    }
  }
}
```

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

  if(argc > 1) {
    setbuf(stdout,NULL);
  }

  pid_t pid;

  pid=fork();
  if (pid==-1) {
    // erreur à l'exécution de fork
    perror("fork");
    exit(EXIT_FAILURE);
  }
  // pas d'erreur
  if (pid==0) {
    // processus fils
    output('f');
    return(EXIT_SUCCESS);
  }
  else {
    // processus père
    output('P');
    printf("\n");
    sleep(1);
    // ...
    return(EXIT_SUCCESS);
  }
}
```

Lors de son exécution, il affiche les lignes suivantes sur *stdout*.

```
$ ./fork-buf
Processus : 852
Processus : 853
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

A première vue, on pourrait penser qu'il n'y a pas de problèmes d'accès concurrents à *stdout* puisque la sortie produite par ce programme semble claire. Cependant, la fonction `output` utilisée par le père et le fils pour afficher les caractères `P` et `f` devrait permettre aux deux processus d'afficher des caractères `P` et `f` avec une certaine alternance. Il n'en est rien et la sortie standard semble indiquer que le processus père a affiché tous les caractères `P` simultanément. Ce n'est pas possible si l'on regarde le code source de la fonction `output`.

L'utilitaire strace(1) qui permet de tracer tous les appels systèmes effectués par un processus nous permet d'analyser comment le processus père et le processus fils écrivent sur *stdout*. Lorsqu'ils utilisent les fonctions printf(3) et putchar(3) de la librairie standard, ils utilisent des fonctions qui finalement utilisent l'appel système write(2) qui est le seul permettant d'écrire sur un flux tel que *stdout*. L'exécution de strace(1) montre que chaque processus effectue deux appels systèmes write(2). Le premier correspond à l'appel à printf(3) et le second à tous les appels à putchar(3) fait par le processus.

```
strace -f -e trace=write ./fork-buf
Process 1398 attached
[pid  1398] write(1, "Processus : 1398\n", 17 <unfinished ...>
[pid  1397] write(1, "Processus : 1397\n", 17Processus : 1397
) = 17
Processus : 1398
[pid  1398] <... write resumed> )       = 17
[pid  1398] write(1, "fffffffffffffffffffffffffffffff"..., 60fffffffff ffffffffffffffffffffff
Process 1398 detached
--- SIGCHLD (Child exited) @ 0 (0) ---
write(1, "PPPPPPPPPPPPPPPPPPPPPPPPPPPPPP"..., 61PPPPPPPPPPPPPPPPPPPPP PPPPPPPPPPPPPPPPPPPPPPPP
) = 61
```

L'exécution d'un appel système tel que write(2) peut être couteux au niveau des performances. Une règle de bonne pratique est d'essayer de minimiser le nombre d'appels systèmes effectués par un programme. La librairie standard prend en compte cette règle et utilise un buffer dans lequel elle stocke temporairement les données produites par les fonctions telles que printf(3) ou putchar(3) avant de les écrire via write(2). Ce buffer peut être contrôlé via les fonctions setvbuf(3) et setbuf(3). Ces fonctions permettent de fixer la taille du buffer utilisé par la librairie standard.

Lorsque le buffer de la librairie stdio(3) est désactivé en faisant `setbuf(stdout,NULL);`, le processus père et le processus fils écrivent bien de façon entrelacée sur la sortie standard.

```
$ ./fork-buf 1
Processus : 1419
PProcessus : 1420
fffPffPffPffPfPffPPfPPPfPfffPfPfPPfffPfPffPPfPPPPfPPfPPfPfPPfPfPffPfPfPfffPPfPPffPffPPfPfPfPPfPf
```

L'utilisation de *strace(1)* lors de cette exécution montre effectivement que chaque appel à la fonction putchar(3) provoque une exécution de l'appel système write(2) :

```
[pid  1420] write(1, "f", 1f)            = 1
[pid  1419] write(1, "P", 1P)            = 1
[pid  1419] write(1, "P", 1P)            = 1
[pid  1420] write(1, "f", 1f)            = 1
```

Note : Faut-il modifier le buffer de la librairie stdio(3) ?

En pratique, il est préférable de ne pas désactiver le buffer utilisé par la librairie stdio(3) car cela peut avoir des conséquences négatives sur les performances des programmes. Par contre, lorsque l'on développe des programmes qui utilisent plusieurs processus il est important de se souvenir de l'existence de ce buffer car il peut expliquer certains comportements qui pourraient apparaître comme étant bizarres lorsque l'on observe l'exécution de processus via les messages qu'ils affichent sur *stderr* ou *stdout*. Lorsque l'on soupçonne un comportement bizarre qui pourrait être expliqué par des interactions avec ce buffer, il est possible d'ajouter dans le programme des appels explicites à la fonction fflush(3) qui a pour effet de vider immédiatement le buffer de stdio(3).

Note : Génération de nombres aléatoires

Le programme `/Threads/S8-src/fork-buf.c` présenté ci-dessus est un exemple d'utilisation de nombres aléatoires. Ceux-ci sont générés avec la fonction rand(3) de la librairie standard. Cette fonction utilise un générateur de nombres aléatoires qui génère toujours la même séquence de nombres aléatoires lorsqu'elle est initialisée avec la même semence par la fonction srand(3). Souvent, les programmeurs qui utilisent des nombres aléatoires cherchent à ce que la séquence générée diffère d'une exécution du programme à l'autre. Une façon simple de procéder est d'utiliser comme semence la somme entre le temps courant retourné par time(3posix) et l'identifiant du processus obtenu via getpid(2). Une telle semence n'est cependant pas suffisante pour toutes les applications. Certaines applications cryptographiques notamment nécessitent des nombres aléatoires qui ne peuvent pas être facilement prédits. Pour ces applications, il est nécessaire d'utiliser des semences qui sont parfaitement aléatoires, comme random(4).

4.7.4 Fin d'un processus

Il y a deux événements importants dans la vie d'un processus sous Unix. Sa création avec l'appel système fork(2) et sa terminaison. Nous avons déjà vu qu'un programme C (et donc un processus) pouvait se terminer de deux façons principales [19] :

— par l'exécution de `return(...)` dans la fonction `main`

— par un appel explicite à la fonction exit(3) dans la fonction `main` ou n'importe quelle fonction du processus

Ces fonctions appellent en fait la fonction de la librairie exit(3). Cette fonction permet de faire plus que simplement terminer le processus en cours d'exécution et retourner sa valeur de retour. Il est en effet possible d'associer une ou plusieurs fonctions de terminaison à exit(3) via la fonction atexit(3). Lorsque exit(3) est appelée, elle lance d'abord

19. Si le processus a été découpé en threads, le processus peut aussi se terminer lorsque son dernier thread se termine en exécutant `return(...)` dans sa fonction de démarrage ou par un appel explicite à pthread_exit(3).

les fonctions enregistrées par atexit(3) puis termine correctement le processus. Ces fonctions de terminaison d'un processus sont utilisées lorsque par exemple un processus utilise des services particuliers du système d'exploitation comme par exemple une mémoire partagée entre plusieurs processus. Ces services consomment des ressources et il est nécessaire de les libérer correctement lorsqu'un processus se termine comme nous le verrons ultérieurement.

L'exemple ci-dessous illustre brièvement l'utilisation de atexit(3).

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

void e1() {
  printf("Exécution de la fonction e1\n");
}

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

  int err;
  err=atexit(e1);
  if(err==-1) {
    perror("atexit");
    exit(EXIT_FAILURE);
  }
  return(EXIT_SUCCESS);

}
```

Après avoir exécuté les fonctions de terminaison, la fonction exit(3) appelle fflush(3) sur tous les flux existants puis les ferme proprement. Ensuite, la fonction exit(3) exécute l'appel système exit(2). Cet appel système est particulier. C'est le seul appel système qui n'a pas de valeur de retour, et pour cause ! Il ferme tous les fichiers qui étaient encore ouverts (normalement un processus devrait fermer proprement tous ses fichiers avant de s'arrêter) et libère les ressources qui étaient associées au processus.

```
#include <unistd.h>

void _exit(int status);
```

L'appel système exit(2) permet au processus qui se termine de retourner un statut à son processus père. Pour récupérer le statut de son fils, un processus père doit utiliser l'appel système waitpid(2).

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

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

L'appel système waitpid(2) prend trois arguments. C'est un appel système bloquant. Le premier argument permet de spécifier quel est le processus fils dont la terminaison est attendue. Un premier argument négatif indique que waitpid(2) attend la terminaison de n'importe quel processus fils. Si le premier argument est positif, alors il contient un identifiant de processus fils et waitpid(2) attendra la terminaison de ce processus [20]. Le second argument est un pointeur vers un entier qui après le retour de waitpid(2) contiendra le statut retourné par le processus fils. Le troisième argument permet de spécifier des options à waitpid(2) que nous n'utiliserons pas. La fonction wait(2) est une simplification de waitpid(2) qui permet d'attendre n'importe quel processus fils. `wait(p)` est en pratique équivalent à `waitpid(-1,p,0)`.

Un processus qui lance un processus fils avec fork(2) *doit* attendre la terminaison de son processus fils en utilisant waitpid(2). Le programme ci-dessous illustre l'utilisation de waitpid(2).

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

20. Si le processus dont l'identifiant est passé comme argument s'est déjà terminé, alors waitpid(2) retourne en indiquant une erreur.

```
int main (int argc, char *argv[])  {
  int status;
  pid_t pid;

  pid=fork();

  if (pid==-1) {
    // erreur à l'exécution de fork
    perror("fork");
    exit(EXIT_FAILURE);
  }
  // pas d'erreur
  if (pid==0) {
    sleep(8);
    return(42);
  }
  else {
    // processus père
    int fils=waitpid(pid,&status,0);
    if(fils==-1) {
      perror("wait");
      exit(EXIT_FAILURE);
    }
    if(WIFEXITED(status)) {
      printf("Le fils %d s'est terminé correctement et a retourné la valeur %d\n",fils,WEXITSTATUS
      return(EXIT_SUCCESS);
    }
    else {
      if( WIFSIGNALED(status)) {
        printf("Le fils %d a été tué par le signal %d\n",fils,WTERMSIG(status));
      }
      return(EXIT_FAILURE);
    }
  }
}
```

Dans ce programme, le processus père récupère la valeur retournée par le fils qu'il a créé. Lors de l'exécution de `waitpid(pid,&status,0);`, la valeur de retour du fils est placée dans l'entier dont l'adresse est `status`. Cet entier contient non-seulement la valeur de retour du processus fils (dans les 8 bits de poids faible), mais aussi une information permettant de déterminer si le processus fils s'est terminé correctement ou a été terminé de façon abrupte via l'utilisation de kill(1). Les macros `WEXITSTATUS` et `WTERMSIG` utilisées pour extraire la valeur de retour et la raison de la terminaison abrupte sont décrites dans waitpid(2).

Même si un processus *doit* attendre la terminaison de tout processus fils qu'il a lancé, il arrive parfois qu'un processus n'attende pas ses fils. Cela peut arriver lorsqu'un processus s'arrête suite à une erreur avant de pouvoir récupérer ses fils. Ce cas est illustré par l'exemple ci-dessous dans lequel le processus père se termine sans attendre son fils.

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


int main (int argc, char *argv[])  {
  pid_t pid;

  pid=fork();

  if (pid==-1) {
```

```
    // erreur à l'exécution de fork
    perror("fork");
    exit(EXIT_FAILURE);
  }
  // pas d'erreur
  if (pid==0) {
    printf("Processus : %d, père : %d\n",getpid(),getppid());
    fflush(stdout);
    sleep(3);
    printf("Processus : %d, père : %d\n",getpid(),getppid());
    return(EXIT_SUCCESS);
  }
  else {
    // processus père
    sleep(1);
    printf("Fin du processus père [%d]\n",getpid());
    return(EXIT_FAILURE);
  }
}
```

Du point de vue du *kernel* cette situation est ennuyeuse car il maintient pour chaque processus non seulement son identifiant de processus mais également l'identifiant de son processus père qui est retourné par getpid(2). Lorsque le père se termine avant son fils, le processus fils est dit *orphelin* et le kernel modifie ses structures de données pour que le père de ce *processus orphelin* soit le processus dont l'identifiant est `1`. Ce processus est le processus init(8) qui est lancé au démarrage du système et n'est jamais arrêté.

```
Processus : 28750, père : 28749
Fin du processus père [28749]
Processus : 28750, père : 1
```

A côté des processus orphelins dont nous venons de parler, un système Unix peut également héberger des *processus zombie*. Un *processus zombie* est un processus qui s'est terminé mais dont la valeur de retour n'a pas encore été récupérée par son père. Dans ce cas, le *kernel* libère l'ensemble des ressources associées au processus fils et ne conserve de ce processus qu'une petite structure de données contenant notamment son identifiant, l'identifiant de son processus père et sa valeur de retour. En pratique, il est préférable d'éviter les processus zombie car ils consomment quand même un peu de ressources.

4.7.5 Exécution d'un programme

fork(2) et waitpid(2) permettent respectivement de créer et de terminer des processus. Pour comprendre la façon dont les programmes sont exécutés, il nous reste à expliquer le fonctionnement de l'appel système execve(2). Cet appel système permet l'exécution d'un programme. Lors de son exécution, l'image en mémoire du processus qui effectue execve(2) est remplacée par l'image de l'exécutable passé en argument à execve(2) et son exécution démarre à sa fonction `main`.

```
#include <unistd.h>

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

execve(2) prend trois arguments. Le premier est le nom complet du fichier exécutable qui doit être lancé. Le second est un pointeur vers un tableau de chaines de caractères contenant les arguments à passer à l'exécutable. Le troisième est un pointeur vers l'environnement qui sera nécessaire à l'exécution du programme. Comme execve(2) remplace l'image mémoire du programme en cours d'exécution, il ne retourne une valeur de retour que si l'appel système échoue. Cela peut être le cas si son premier argument n'est pas un fichier exécutable accessible par exemple.

execve(2) s'utilise souvent juste après l'exécution de fork(2), mais il est aussi possible de l'utiliser directement dans un programme. Dans ce cas, le programme qui exécute avec succès execve(2) disparaît et est remplacé par le programme appelé. Le programme ci-dessous illustre une utilisation simple de execve(2).

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

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

  char *arguments[]={"expr", "1", "+", "2", NULL};
  char *environnement[]={"LANG=fr",NULL};

  printf("Exécution du processus %d\n",getpid());
  printf("Exécution de /usr/bin/expr\n");
  int err=execve("/usr/bin/expr", arguments, environnement);
  if(err!=0) {
    perror("execve");
    exit(EXIT_FAILURE);
  }
  // jamais atteint
  printf("Ce message ne sera jamais affiché\n");
  return(EXIT_SUCCESS);

}
```

Lors de son exécution, ce programme affiche sur sa sortie standard les lignes suivantes :

```
3
```

Il y a quelques points importants à noter concernant l'utilisation de execve(2). Tout d'abord, execve(2) remplace l'entièreté de l'image mémoire du processus qui exécute cet appel système, y compris les arguments, les variables d'environnement. Par contre, le *kernel* conserve certaines informations qu'il maintenait pour le processus. C'est le cas notamment de l'identifiant du processus et de l'identifiant du processus père. Si le processus qui a effectué execve(2) avait lancé des threads, ceux-ci seraient immédiatement supprimés puisque l'image du processus en cours d'exécution est remplacé lors de l'exécution de execve(2). Les flux standard (*stdin*, *stdout* et *stderr*) sont utilisables par le programme exécuté via execve(2). Il faut cependant noter que lors de l'appel à execve(2), les données qui se trouveraient éventuellement dans le buffer de la librairie *stdio* ne sont pas automatiquement envoyées vers leurs flux respectifs. Cela pourrait paraître étonnant puisque lorsqu'un processus se termine avec exit(3), exit(3) vide les buffers de *stdio* avant d'appeler exit(2). execve(2) est un appel système qui est exécuté par le kernel. Celui-ci ne peut pas savoir si il y a des données en attente d'écriture dans *stdio*. Il ne peut donc pas automatiquement vider les buffers maintenus par la librairie *stdio*. Si des données ont été écrites avec printf(3) avant l'exécution de execve(2), il est préférable de forcer leur écriture via fflush(3) avant d'appeler execve(2).

L'appel système execve(2) est très souvent exécuté dans un shell tel que bash(1). Lorsqu'un shell lance un programme externe, il doit d'abord utiliser fork(2) pour créer une copie de lui-même. Ensuite, le processus père se met en attente via waitpid(2) de la valeur de retour du processus fils créé. Le processus fils quant à lui utilise execve(2) pour exécuter le programme demandé.

La programme ci-dessous est un exemple un peu plus complexe de l'utilisation de fork(2), execve(2) et waitpid(2). Ce programme prend comme argument une liste d'exécutables et il essaye de les exécuter l'un à la suite de l'autre. Pour cela, il parcourt ses arguments et essaye pour chaque argument de créer un processus fils et d'y exécuter le programme correspondant. Si le programme a pu être exécuté, sa valeur de retour est récupérée par le processus père. Si l'appel à execve(2) a échoué, le processus fils se termine avec `127` comme valeur de retour. Comme celle-ci est stockée sur 8 bits, c'est la plus grande valeur de retour positive qu'il est possible de retourner depuis un processus fils. Cette valeur indique au processus père que le fils n'a pas réussi à exécuter execve(2).

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <libgen.h>

extern char **environ;
```

```c
int main (int argc, char *argv[])  {
  int status;
  pid_t pid;

  for(int i=1;i<argc;i++) {
    // création du fils
    pid=fork();
    if (pid==-1) {
      perror("fork");
      exit(EXIT_FAILURE);
    }
    if (pid==0) {
      // fils
      printf ("Exécution de la commande %s [pid=%d]\n",argv[i],getpid());
      fflush(stdout);
      char *arguments[2];
      arguments[0]=basename(argv[i]);
      arguments[1]=NULL;
      int err=execve(argv[i], arguments, environ);
      if(err!=0)
        return(127);
    } // fils
    else {
      // processus père
      int fils=waitpid(pid,&status,0);
      if(fils==-1) {
        perror("wait");
        exit(EXIT_FAILURE);
      }
      if(WIFEXITED(status)) {
        if(WEXITSTATUS(status)==0)
          printf("La commande %s [%d] s'est terminée correctement\n",argv[i],fils);
        else
          if (WEXITSTATUS(status)==127)
            printf("La commande %s n'a pu être exécutée\n",argv[i]);
          else
            printf("La commande %s [%d] a retourné %d\n",argv[i],fils,WEXITSTATUS(status));
      }
      else {
        if( WIFSIGNALED(status))
          printf("La commande %s [%d] ne s'est pas terminée correctement\n",argv[i],fils);
      }
      fflush(stdout);
    } // père
  } // for loop
  return(EXIT_SUCCESS);
}
```

Lors de son exécution, ce programme affiche sur sa sortie standard les lignes suivantes :

```
$./fork-manyexec /bin/true /bin/false /bin/none
Exécution de la commande /bin/true [pid=14217]
La commande /bin/true [14217] s'est terminée correctement
Exécution de la commande /bin/false [pid=14218]
La commande /bin/false [14218] a retourné 1
Exécution de la commande /bin/none [pid=14219]
La commande /bin/none n'a pu être exécutée
```

En pratique, il existe plusieurs fonctions de la librairie standard qui apportent de petites variations à execve(2). Il s'agit de execl(3), execlp(3), execle(3), execv(3posix) et execv(3). Ces fonctions utilisent toutes l'appel système execve(2). Elles permettent de spécifier de différentes façons le programme à exécuter ou les variables d'environnement. Enfin, la fonction system(3) de la librairie permet d'exécuter une commande du shell directement depuis un programme.

Outre les exécutables compilés, Unix et Linux supportent également l'exécution de programmes interprétés. Contrairement aux programmes compilés que nous avons manipulé jusque maintenant, un programme interprété est un programme écrit dans un langage qui doit être utilisé via un *interpréteur*. Un *interpréteur* est un programme qui lit des commandes sous la forme de texte et exécute directement les instructions correspondant à ces commandes. Unix supporte de nombreux interpréteurs et comme nous allons le voir il est très facile de rajouter de nouveaux interpréteurs de commande. L'interpréteur le plus connu est bash(1) et ses nombreuses variantes. En voici quelques autres :

— awk(1) est un langage de programmation interprété qui permet de facilement manipuler des textes
— perl(1) est un langage de programmation complet qui a été initialement développé pour la manipulation de textes, mais est utilisé dans de nombreuses autres applications
— python(1) est un langage de programmation complet

Pour comprendre la façon dont Unix interagit avec les interpréteurs de commande, il est bon de voir en détails comment execve(2) reconnaît qu'un fichier contient un programme qui peut être exécuté. Tout d'abord, le système de fichiers contient pour chaque fichier des métadonnées qui fournissent de l'information sur le possesseur du fichier, sa date de création, ... Une de ces métadonnées est un bit [21] qui indique si le fichier est exécutable ou non. Ce bit peut être manipulé en utilisant la commande chmod(1). Lorsqu'un programme est compilé avec gcc(1), celui-ci utilise chmod(1) pour marquer le programme comme étant exécutable.

```
$ ls -l a.out
-rwxr-xr-x 1 obo stafinfo 8178 Mar 16 13:42 a.out
$ chmod -x a.out
$ ./a.out
-bash: ./a.out: Permission denied
$ chmod +x a.out
$ ./a.out
  exécution de a.out
$ ls -l a.out
-rwxr-xr-x 1 obo stafinfo 8178 Mar 16 13:42 a.out
```

Lorsqu'execve(2) est appelé, il vérifie d'abord ce bit de permission. Si il n'indique pas que le programme est exécutable, execve(2) retourne une erreur. Ensuite, execve(2) ouvre le fichier dont le nom a été passé comme premier argument. Par convention, le début du fichier contient une séquence d'octets ou de caractères qui indiquent le type de fichier dont il s'agit. La commande file(1) permet de tester le type d'un fichier inconnu.

```
$ file fork-execve.c
fork-execve.c: UTF-8 C program text
$ file script.sh
script.sh: Bourne-Again shell script text executable
$ file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),   dynamically linked (uses share
```

Pour les exécutables, deux cas de figure sont possibles :

1. le fichier contient un programme compilé et directement exécutable. Sur les systèmes Linux actuels, ce fichier sera au format elf(5). Il débute par une entête qui contient une chaîne de caractères utilisée comme marqueur ou chaîne magique. L'entête fournit de l'information sur le type d'exécutable et sa structure. Voici à titre d'illustration le contenu de l'entête d'un programme compilé décortiqué par l'utilitaire readelf(1) :

```
$ readelf -h a.out
ELF Header:
 Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
 Class:                             ELF64
 Data:                              2's complement, little endian
 Version:                           1 (current)
 OS/ABI:                            UNIX - Linux
 ABI Version:                       0
 Type:                              EXEC (Executable file)
 Machine:                           Advanced Micro Devices X86-64
```

21. En pratique, il y a trois bits qui jouent ce rôle en fonction du possesseur du fichier et de l'utilisateur qui souhaite l'exécuter. Nous décrirons ces bits en détails dans un prochain chapitre.

```
Version:                           0x1
Entry point address:               0x4006e0
Start of program headers:          64 (bytes into file)
Start of section headers:          3712 (bytes into file)
Flags:                             0x0
Size of this header:               64 (bytes)
Size of program headers:           56 (bytes)
Number of program headers:         8
Size of section headers:           64 (bytes)
Number of section headers:         30
Section header string table index: 27
```

2. Le fichier contient un programme en langage interprété. Dans ce cas, la première ligne débute par `#!` suivi du nom complet de l'interpréteur à utiliser et de ses paramètres éventuels. Le programme interprété commence sur la deuxième ligne. A titre d'exemple, voici un petit script bash(1) qui permet de tester si un fichier est interprétable ou non en testant la valeur des deux premiers caractères du fichier et ses métadonnées.

```
#!/bin/bash
# script.sh
if [ $# -ne 1 ]
then
    echo "Usage: `basename $0` fichier"
    exit 1
fi
if [ -x ${1} ]
then
    head -1 $1 | grep "^#\!" >>/dev/null
    if [ $? ]
    then
        echo "Script interprétable"
        exit 0
    else
        echo "Script non-interprétable"
        exit 1
    fi
else
    echo "Bit x non mis dans les métadonnées"
    exit 1
fi
```

Sous Unix et Linux, n'importe quel programmeur peut définir son propre interpréteur. Il suffit qu'il s'agisse d'un exécutable compilé et que le nom de cet interpréteur soit présent dans la première ligne du fichier à interpréter. Lors de l'exécution d'un programme utilisant cet interpréteur, celui-ci recevra le contenu du fichier et pourra l'interpréter. Ainsi, par exemple le programme interprété ci-dessous est tout à fait valide.

```
#!/usr/bin/tail -n +1
Hello, world
SINF1252
```

Lors de son exécution via execve(2), l'interpréteur tail(1) va être chargé avec comme arguments `-n +1` et il affichera sur *stdout* la ligne `SINF1252`.

Cette facilité d'ajouter de nouveaux interpréteurs de commande est une des forces des systèmes d'exploitation de la famille Unix.

4.7.6 Table des processus

Un système d'exploitation tel que Linux maintient certaines informations concernant chaque processus dans sa *table des processus*. Une description complète du contenu de cette table des processus sort du cadre de ce chapitre. Par contre, il est intéressant de noter que sous Linux il existe de nombreux utilitaires qui permettent de consulter le contenu de la table des processus et notamment :

— ps(1) qui est l'utilitaire de base pour accéder à la table de processus et lister les processus en cours d'exécution
— top(1) qui affiche de façon interactive les processus qui consomment actuellement du temps CPU, de la mémoire, ...
— pstree(1) qui affiche l'arbre des processus avec les relations père-fils

Tous ces utilitaires utilisent les informations contenues dans le répertoire `/proc`. Il s'agit d'un répertoire spécial qui contient de l'information à propos du système d'exploitation y compris la table de processus. Son contenu est détaillé dans la page de manuel qui lui est consacrée : proc(5).

A titre d'illustration, considérons le shell d'un utilisateur en cours. Les informations maintenues dans la table des processus pour ce processus sont accessibles depuis `/proc/pid` où `pid` est l'identifiant du processus en cours d'exécution. Linux stocke de très nombreuses informations sur chaque processus. Celles-ci sont structurées dans des fichiers et des répertoires :

```
$ ls -l /proc/18557
total 0
dr-xr-xr-x 2 obo stafinfo 0 Mar 18 16:37 attr
-r-------- 1 obo stafinfo 0 Mar 18 16:37 auxv
-r--r--r-- 1 obo stafinfo 0 Mar 18 16:37 cgroup
--w------- 1 obo stafinfo 0 Mar 18 16:37 clear_refs
-r--r--r-- 1 obo stafinfo 0 Mar 18 14:56 cmdline
-rw-r--r-- 1 obo stafinfo 0 Mar 18 16:37 coredump_filter
-r--r--r-- 1 obo stafinfo 0 Mar 18 16:37 cpuset
lrwxrwxrwx 1 obo stafinfo 0 Mar 18 16:37 cwd ->  /etinfo/users2/obo/sinf1252/SINF1252/
-r-------- 1 obo stafinfo 0 Mar 18 16:37 environ
lrwxrwxrwx 1 obo stafinfo 0 Mar 18 16:37 exe -> /bin/bash
dr-x------ 2 obo stafinfo 0 Mar 18 14:56 fd
dr-x------ 2 obo stafinfo 0 Mar 18 16:37 fdinfo
-r--r--r-- 1 obo stafinfo 0 Mar 18 16:37 io
-rw------- 1 obo stafinfo 0 Mar 18 16:37 limits
-rw-r--r-- 1 obo stafinfo 0 Mar 18 16:37 loginuid
-r--r--r-- 1 obo stafinfo 0 Mar 18 16:37 maps
-rw------- 1 obo stafinfo 0 Mar 18 16:37 mem
-r--r--r-- 1 obo stafinfo 0 Mar 18 16:37 mountinfo
-r--r--r-- 1 obo stafinfo 0 Mar 18 16:37 mounts
-r-------- 1 obo stafinfo 0 Mar 18 16:37 mountstats
dr-xr-xr-x 6 obo stafinfo 0 Mar 18 16:37 net
-r--r--r-- 1 obo stafinfo 0 Mar 18 16:37 numa_maps
-rw-r--r-- 1 obo stafinfo 0 Mar 18 16:37 oom_adj
-r--r--r-- 1 obo stafinfo 0 Mar 18 16:37 oom_score
-r-------- 1 obo stafinfo 0 Mar 18 16:37 pagemap
-r-------- 1 obo stafinfo 0 Mar 18 16:37 personality
lrwxrwxrwx 1 obo stafinfo 0 Mar 18 16:37 root -> /
-rw-r--r-- 1 obo stafinfo 0 Mar 18 16:37 sched
-r--r--r-- 1 obo stafinfo 0 Mar 18 16:37 schedstat
-r--r--r-- 1 obo stafinfo 0 Mar 18 16:37 sessionid
-r--r--r-- 1 obo stafinfo 0 Mar 18 16:37 smaps
-r-------- 1 obo stafinfo 0 Mar 18 16:37 stack
-r--r--r-- 1 obo stafinfo 0 Mar 18 14:56 stat
-r--r--r-- 1 obo stafinfo 0 Mar 18 16:37 statm
-r--r--r-- 1 obo stafinfo 0 Mar 18 14:56 status
-r-------- 1 obo stafinfo 0 Mar 18 16:37 syscall
dr-xr-xr-x 3 obo stafinfo 0 Mar 18 15:59 task
-r--r--r-- 1 obo stafinfo 0 Mar 18 16:37 wchan
```

Certaines des entrées dans `/proc` sont des fichiers, d'autres sont des répertoires. A titre d'exemple, voici quelques unes des entrées utiles à ce stade de notre exploration de Linux.

— `cmdline` est un fichier texte contenant la ligne de commande utilisée pour lancer le processus
— `environ` est un fichier texte contenant les variables d'environnement passées au processus

```
$ (cat /proc/18557/environ; echo) | tr '\000' '\n'
USER=obo
```

```
LOGNAME=obo
HOME=/etinfo/users2/obo
PATH=/usr/local/bin:/bin:/usr/bin
MAIL=/var/mail/obo
SHELL=/bin/bash
```

— `status` est une indication sur l'état actuel du processus. Les premières lignes indiquent dans quel état le processus se trouve ainsi que son identifiant, l'identifiant de son père, ...

```
$ cat /proc/$$/status | head -5
Name:     bash
State:    S (sleeping)
Tgid:     18557
Pid:      18557
PPid:     18556
```

— `limits` est un fichier texte contenant les limites actuelles imposées par le système sur le processus. Ces limites peuvent être modifiées en utilisant *ulimit(1)* à l'intérieur de bash(1) ou via les appels systèmes getrlimit(2)/setrlimit(2).

```
$ cat /proc/18557/limits
Limit                     Soft Limit           Hard Limit           Units
Max cpu time              unlimited            unlimited            seconds
Max file size             unlimited            unlimited            bytes
Max data size             unlimited            unlimited            bytes
Max stack size            10485760             unlimited            bytes
Max core file size        0                    unlimited            bytes
Max resident set          unlimited            unlimited            bytes
Max processes             1024                 24064                processes
Max open files            1024                 1024                 files
Max locked memory         65536                65536                bytes
Max address space         unlimited            unlimited            bytes
Max file locks            unlimited            unlimited            locks
Max pending signals       24064                24064                signals
Max msgqueue size         819200               819200               bytes
Max nice priority         0                    0
Max realtime priority     0                    0
Max realtime timeout      unlimited            unlimited            us
```

— `task` est un répertoire qui contient pour chaque thread lancé par le processus un sous-répertoire avec toutes les informations qui sont relatives à ce thread.

Nous aurons l'occasion de présenter ultérieurement d'autres éléments utiles se trouvant dans `/proc`. Une description plus détaillée est disponible dans la page de manuel proc(5) et des livres de référence tels que [Kerrisk2010].

CHAPITRE 5

Fichiers

5.1 Gestion des utilisateurs

Unix est un système d'exploitation multi-utilisateurs. Un tel système impose des contraintes de sécurité qui n'existent pas sur un système mono-utilisateur. Il est intéressant de passer en revue quelques unes de ces contraintes :

- il doit être possible d'identifier et/ou d'authentifier les utilisateurs du système
- il doit être possible d'exécuter des processus appartenant à plusieurs utilisateurs simultanément et de déterminer quel utilisateur est responsable de chaque opération
- le système d'exploitation doit fournir des mécanismes simples qui permettent de contrôler l'accès aux différentes ressources (mémoire, stockage, ...).
- il doit être possible d'allouer certaines ressources à un utilisateur particulier à un moment donné

Aujourd'hui, la plupart des systèmes informatiques demandent une authentification de l'utilisateur sous la forme d'un mot de passe, d'une manipulation particulière voire d'une identification biométrique comme une empreinte digitale. Cette authentification permet de vérifier que l'utilisateur est autorisé à manipuler le système informatique. Cela n'a pas toujours été le cas et de nombreux systèmes informatiques plus anciens étaient conçus pour être utilisés par un seul utilisateur qui était simplement celui qui interagissait physiquement avec l'ordinateur.

Les systèmes Unix supportent différents mécanismes d'authentification. Le plus simple et le plus utilisé est l'authentification par mot de passe. Chaque utilisateur est identifié par un nom d'utilisateur et il doit prouver son identité en tapant son mot de passe au démarrage de toute session sur le système. En pratique, une session peut s'établir localement sur l'ordinateur via son interface graphique par exemple ou à distance en faisant tourner un serveur tel que sshd(8) sur le système Unix et en permettant aux utilisateurs de s'y connecter via Internet en utilisant un client ssh(1). Dans les deux cas, le système d'exploitation lance un processus login(1) qui permet de vérifier le nom d'utilisateur et le mot de passe fourni par l'utilisateur. Si le mot de passe correspond à celui qui est stocké sur le système, l'utilisateur est authentifié et son shell peut démarrer. Sinon, l'accès au système est refusé.

Lorsqu'un utilisateur se connecte sur un système Unix, il fournit son nom d'utilisateur ou *username*. Ce nom d'utilisateur est une chaîne de caractères qui est facile à mémoriser par l'utilisateur. D'un point de vue implémentation, un système d'exploitation préfère manipuler des nombres plutôt que des chaînes de caractères. Unix associe à chaque utilisateur un identifiant qui est stocké sous la forme d'un nombre entier positif. La table de correspondance entre l'identifiant d'utilisateur et le nom d'utilisateur est le fichier */etc/passwd*. Ce fichier texte, comme la grande majorité des fichiers de configuration d'un système Unix, comprend pour chaque utilisateur l'information suivante :

- nom d'utilisateur (username)
- mot de passe (sur les anciennes versions de Unix)
- identifiant de l'utilisateur (*userid*)
- identifiant du groupe principal auquel l'utilisateur appartient
- nom et prénom de l'utilisateur
- répertoire de démarrage de l'utilisateur
- shell de l'utilisateur

L'extrait ci-dessous présente un exemple de fichier `/etc/passwd`. Des détails complémentaires sont disponibles dans la page de manuel passwd(5). Un utilisateur peut modifier les informations le concernant dans ce fichier avec la commande passwd(1).

```
# Exemple de /etc/passwd
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
slampion:*:1252:1252:Séraphin Lampion:/home/slampion:/bin/bash
```

Il y a en pratique trois types d'utilisateurs sur un système Unix. L'utilisateur *root* est l'administrateur du système. C'est l'utilisateur qui a le droit de réaliser toutes les opérations sur le système. Il peut créer de nouveaux utilisateurs, mais aussi reformatter les disques, arrêter le système, interrompre des processus utilisateurs ou accéder à l'ensemble des fichiers sans restriction. Par convention, cet utilisateur a l'identifiant `0`. Ensuite, il y a tous les utilisateurs *normaux* du système Unix. Ceux-ci ont le droit d'accéder à leurs fichiers, d'interagir avec leurs processus mais en général ne peuvent pas manipuler les fichiers d'autres utilisateurs ou interrompre leurs processus. L'utilisateur *slampion* dans l'exemple ci-dessus est un utilisateur *normal*. Enfin, pour faciliter l'administration du système, certains systèmes Unix utilisent des utilisateurs qui correspondent à un service particulier comme l'utilisateur *daemon* dans l'exemple ci-dessus. Une discussion de ce type d'utilisateur sort du cadre de ces notes. Le lecteur intéressé pourra consulter une référence sur l'administration des système Unix telle que [AdelsteinLubanovic2007] ou [Nemeth+2010].

Unix associe à chaque processus un identifiant d'utilisateur. Cet identifiant est stocké dans l'entrée du processus dans la table des processus. Un processus peut récupérer son identifiant d'utilisateur via l'appel système getuid(2). Outre cet appel système, il existe également l'appel système setuid(2) qui permet de modifier le *userid* du processus en cours d'exécution. Pour des raisons évidentes de sécurité, seul un processus appartenant à l'administrateur système (*root*) peut exécuter cet appel système. C'est le cas par exemple du processus login(1) qui appartient initialement à *root* puis exécute setuid(2) afin d'appartenir à l'utilisateur authentifié puis exécute execve(2) pour lancer le premier shell appartenant à l'utilisateur.

En pratique, il est parfois utile d'associer des droits d'accès à des groupes d'utilisateurs plutôt qu'à un utilisateur particulier. Par exemple, un département universitaire peut avoir un groupe correspondant à tous les étudiants et un autre aux membres du staff pour leur donner des permissions différentes. Un utilisateur peut appartenir à un groupe principal et plusieurs groupes secondaires. Le groupe principal est spécifié dans le fichier passwd(5) tandis que le fichier `/etc/group` décrit dans group(5) contient les groupes secondaires.

5.2 Systèmes de fichiers

Outre un processeur et une mémoire, la plupart des ordinateurs actuels sont en général équipés d'un ou plusieurs dispositifs de stockage. Les dispositifs les plus courants sont le disque dur, le lecteur de CD/DVD, la clé USB, la carte mémoire, ... Ces dispositifs de stockage ont des caractéristiques techniques très différentes. Certains stockent l'information sous forme magnétique, d'autres sous forme électrique ou en creusant via un laser des trous dans un support physique. D'un point de vue logique, ils offrent tous une interface très similaire au système d'exploitation qui veut les utiliser.

En pratique, on peut modéliser la plupart des dispositifs de stockage comme étant une zone de stockage qui est décomposée en blocs de quelques centaines ou milliers d'octets qui sont appelés des secteurs. Sur de nombreux dispositifs de stockage, un *secteur* peut stocker un bloc de 512 octets. Chaque secteur est identifié par une adresse et le dispositif de stockage offre au système d'exploitation une interface simple comprenant deux fonctions :

— `int err=device_read(addr_t addr, sector_t *buf)` où `addr` est l'adresse du secteur dont la lecture est demandée et `buf` un buffer destiné à recevoir le contenu du secteur qui a été lu
— `int err=device_write(addr_t addr, sector_t *buf)` où `addr` est l'adresse d'un secteur et `buf` un buffer contenant le secteur à écrire

Un dispositif de stockage permet donc essentiellement de lire et d'écrire des secteurs entiers. La plupart des implémentations sont optimisées pour pouvoir lire ou écrire plusieurs secteurs consécutifs, mais la plus petite unité de lecture ou d'écriture est le *secteur*, c'est-à-dire un bloc de 512 octets consécutifs. La plupart des programmes ne sont pas prêts à manipuler directement de tels dispositifs de stockage et Unix comme d'autres systèmes d'exploitation contient une interface de plus haut niveau qui permet aux programmes applicatifs d'utiliser des fichiers et des répertoires. Ces fichiers et répertoires sont une abstraction qui est construite par le système d'exploitation au-dessus des secteurs qui sont stockés sur le disque. Cette abstraction est appelée un *système de fichiers* ou *file-system* en anglais. Il existe des centaines de systèmes de fichiers et Linux supporte quelques dizaines de systèmes de fichiers différents.

En simplifiant, un système de fichier Unix s'appuie toujours sur quelques principes de base assez simples. Le premier est qu'un *fichier* est une suite ordonnée d'octets. Un nom est associé à cette suite d'octets et les programmes utilisent ce nom pour accéder au fichier. En pratique, cette suite ordonnée d'octets sera stockée dans un ou plusieurs secteurs. Comme rien ne garantit que la taille d'un fichier sera un multiple du nombre d'octets dans un secteur, le système d'exploitation devra être capable de gérer des secteurs qui sont partiellement remplis. En outre, même si c'est souvent efficace du point de vue des performances, rien ne garantit qu'un fichier sera stocké dans des secteurs consécutifs. Le système de fichiers doit donc pouvoir supporter des fichiers qui sont composés de données qui sont stockées dans des secteurs se trouvant n'importe où sur le dispositif de stockage.

Sous Unix, le lien entre les différents secteurs qui composent un fichier est fait grâce à l'utilisation des inodes. Un *inode* est une structure de données qui est stockée sur le disque et contient les méta-informations qui sont relatives à un fichier. Un inode peut être représenté par une structure de données similaire à la structure ci-dessous [1]. Un *inode* a une taille fixe et une partie du disque (en pratique un ensemble contigu de secteurs, souvent au début du disque) est réservée pour stocker *N* inodes. Connaissant la taille de cette zone, il est possible d'accéder facilement au i^{me} inode du système de fichiers.

```
struct simple_inode {
        uint16 mode;
        uid_t  uid;
        gid_t  gid;
        uint32 size;
        unit32 atime;
        unit32 mtime;
        unit32 ctime;
        uint16 nlinks;
        uint16 zone[10];
};
```

L'*inode* contient les principales méta-données qui sont associées au fichier, à savoir :

- le `mode` qui contient un ensemble de drapeaux binaires avec les permissions associées au fichier
- le `userid` du propriétaire du fichier
- le `groupid` qui identifie le groupe auquel le fichier appartient
- la taille du fichier en bytes
- l'instant de dernier accès au fichier (`atime`)
- l'instant de dernière modification du fichier (`mtime`)
- l'instant de dernier changement d'état du fichier (`ctime`)
- le nombre de liens vers ce fichier (`nlinks`)
- la liste ordonnée des secteurs qui contiennent le fichier (`zone`) [2]

Le lecteur attentif aura noté que parmi les méta-données qui sont associées via un *inode* à un fichier on ne retrouve pas le nom du fichier. Sous Unix, le nom de fichier n'est pas directement associé au fichier lui-même comme sa taille ou ses permissions. Il est stocké dans les répertoires. Un *répertoire* est une abstraction qui permet de regrouper ensemble plusieurs fichiers et/ou répertoires. En pratique, un *répertoire* est un fichier qui a un format spécial. Il contient une suite d'entrées qui contiennent chacune un nom (de fichier ou de répertoire), une indication de type qui permet notamment de distinguer les fichiers des répertoires et le numéro de l'*inode* qui contient les méta-données du fichier ou répertoire. A titre d'exemple, l'extrait ci-dessous [3] est la définition d'une entrée de répertoire dans le système de fichiers de type *ext2* sous Linux [Card+1994]. Cette entrée contient également une indication de longueur. Un répertoire contiendra une entrée par fichier ou répertoire qui y a été placé.

```
/*
 * Structure of a directory entry
 */
#define EXT2_NAME_LEN 255

ext2_dir_entry_2 {
        __le32  inode;                  /* Inode number */
```

1. Cette structure est partiellement inspirée du format des inodes du système de fichiers Minix, voir linux/minix_fs.h.
2. Le champ `zone[10]` permet de stocker dans l'*inode* les références vers les premiers secteurs du fichier et des références vers d'autres blocs qui contiennent eux-aussi des références vers des blocs. Cela permet de stocker une liste de secteurs qui est de taille variable à partir d'un *inode* qui a lui une taille fixe. Une description détaillée des inodes peut se trouver dans une référence sur les systèmes d'exploitation telle que [Tanenbaum+2009].
3. Source : linux/ext2_fs.h

```
        __le16  rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;
        char    name[EXT2_NAME_LEN];    /* File name */
};

/*
 * Ext2 directory file types.  Only the low 3 bits are used.  The
 * other bits are reserved for now.
 */
enum {
        EXT2_FT_UNKNOWN         = 0,
        EXT2_FT_REG_FILE        = 1,
        EXT2_FT_DIR             = 2,
        EXT2_FT_CHRDEV          = 3,
        EXT2_FT_BLKDEV          = 4,
        EXT2_FT_FIFO            = 5,
        EXT2_FT_SOCK            = 6,
        EXT2_FT_SYMLINK         = 7,
        EXT2_FT_MAX             = 8
};
```

Dans un système de fichiers Unix, l'ensemble des répertoires et fichiers est organisé sous la forme d'un arbre. La racine de cet arbre est le répertoire `/`. Il est localisé sur un des dispositifs de stockage du système. Le système de fichiers Unix permet d'intégrer facilement des systèmes de fichiers qui se trouvent sur différents dispositifs de stockage. Cette opération est en général réalisée par l'administrateur système en utilisant la commande mount(8). A titre d'exemple, voici quelques répertoires qui sont montés sur un système Linux.

```
$ df -k
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda1             15116836   9425020   4923912  66% /
tmpfs                  1559516      4724   1554792   1% /dev/shm
/dev/sda2             19840924   4374616  14442168  24% /xendomains
xenstore               1972388        40   1972348   1% /var/lib/xenstored
david:/mnt/student   258547072 188106176  59935296  76% /etinfo/users
david:/mnt/staff     254696800 177422400  64136160  74% /etinfo/users2
```

Dans l'exemple ci-dessus, la première colonne correspond au dispositif de stockage qui contient les fichiers et répertoires. Le premier dispositif de stockage `/dev/sda1/` est un disque local et contient le répertoire racine du système de fichiers. Le système de fichiers `david://mnt/student` est stocké sur le serveur `david` et est monté via mount(8) dans le répertoire `/etinfo/users`. Ainsi, tout accès à un fichier dans le répertoire `/etinfo/users` se fera via le serveur `david`.

Chaque répertoire du système de fichiers contient un ou plusieurs répertoires et un ou plusieurs fichiers. A titre d'exemple, il est intéressant de regarder le contenu de deux répertoires. Le premier est un extrait au contenu du répertoire racine obtenu avec la commande ls(1)

```
$ ls -la /
dr-xr-xr-x.  26 root root 233472 Feb 23 03:18 .
dr-xr-xr-x.  26 root root 233472 Feb 23 03:18 ..
-rw-r--r--    1 root root      0 Feb 13 16:45 .autofsck
-rw-r--r--    1 root root      0 Jul 27  2011 .autorelabel
dr-xr-xr-x.   4 root root   4096 Dec 15 05:50 boot
drwxr-xr-x   19 root root   4160 Mar 22 12:04 dev
drwxr-xr-x. 125 root root  12288 Mar 22 12:04 etc
drwxr-xr-x    4 root root      0 Mar 22 10:22 etinfo
drwxr-xr-x.   2 root root   4096 Jan  6  2011 home
dr-xr-xr-x.  14 root root   4096 Mar 22 03:26 lib
dr-xr-xr-x.  10 root root  12288 Mar 22 03:26 lib64
drwx------.   2 root root  16384 Jul 27  2011 lost+found
...
drwxrwxrwt. 104 root root   4096 Mar 22 12:05 tmp
```

```
drwxr-xr-x.  13 root root  4096 Jul 19  2011 usr
drwxr-xr-x.  23 root root  4096 Jul 27  2011 var
```

Le répertoire racine contient quelques fichiers et des répertoires. Tout répertoire contient deux répertoires spéciaux. Le premier répertoire, identifié par le caractère `.` (un seul point) est un alias vers le répertoire lui-même. Cette entrée de répertoire est présente dans chaque répertoire dès qu'il est créé avec une commande telle que mkdir(1). Le deuxième répertoire spécial est `..` (deux points consécutifs). Ce répertoire est un alias vers le répertoire parent du répertoire courant.

Les méta-données qui sont associées à chaque fichier ou répertoire contienent outre les informations de type les bits de permission. Ceux-ci permettent d'encoder trois types de permissions et d'autorisation :

— `r` : autorisation de lecture
— `w` : autorisation d'écriture ou de modification
— `x` : autorisation d'exécution

Ces bits de permissions sont regroupés en trois blocs. Le premier bloc correspond aux bits de permission qui sont applicables pour les accès qui sont effectués par un processus qui appartient à l'utilisateur qui est propriétaire du fichier/répertoire. Le deuxième bloc correspond aux bits de permission qui sont applicables pour les opérations effectuées par un processus dont l'identifiant de groupe est identique à l'identifiant de groupe du fichier/répertoire mais n'appartient pas à l'utilisateur qui est propriétaire du fichier/répertoire. Le dernier bloc est applicable pour les opérations effectuées par des processus qui appartiennent à d'autres utilisateurs.

Les valeurs de ces bits sont représentés pas les symboles `rwx` dans l'output de la commande ls(1). Les bits de permission peuvent être modifiés en utilisant la commande chmod(1) qui utilise l'appel système chmod(2). Pour qu'un exécutable puisse être exécuté via l'appel système execve(2), il est nécessaire que le fichier correspondant possède les bits de permission `r` et `x`.

Note : Manipulation des bits de permission avec chmod(2)

L'appel système chmod(2) permet de modifier les bits de permission qui sont associés à un fichier. Ceux-ci sont encodés sous la forme d'un entier sur 16 bits.

— `S_IRUSR (00400)` : permission de lecture par le propriétaire
— `S_IWUSR (00200)` : permission d'écriture par le propriétaire
— `S_IXUSR (00100)` : permission d'exécution par le propriétaire
— `S_IRGRP (00040)` : permission de lecture par le groupe hormis le propriétaire
— `S_IWGRP (00020)` : permission d'écriture par le groupe hormis le propriétaire
— `S_IXGRP (00010)` : permission d'exécution par le groupe hormis le propriétaire
— `S_IROTH (00004)` : permission de lecture par tout utilisateur hormis le propriétaire et son groupe
— `S_IWOTH (00002)` : permission d'écriture par tout utilisateur hormis le propriétaire et son groupe
— `S_IXOTH (00001)` : permission d'exécution par tout utilisateur hormis le propriétaire et son groupe

```c
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
```

Ces bits de permissions sont généralement spécifiés soit sous la forme d'une disjonction logique ou sous forme numérique. A titre d'exemple, un fichier qui peut être lu et écrit uniquement par son propriétaire aura comme permissions `00600` ou `S_IRUSR|S_IWUSR`.

Le *nibble* de poids fort des bits de permission sert à encoder des permissions particulières relatives aux fichiers et répertoires. Par exemple, lorsque la permission `S_ISUID (04000)` est associée à un exécutable, elle indique que celui-ci doit s'exécuter avec les permissions du propriétaire de l'exécutable et pas les permissions de l'utilisateur. Cette permission spéciale est utilisée par des programmes comme passwd(1) qui doivent disposer des permissions de l'administrateur système pour s'exécuter correctement (passwd(1) doit modifier le fichier passwd(5) qui appartient à l'administrateur système).

Les exemples ci-dessous présentent le contenu partiel d'un répertoire avec en première colonne le numéro de l'inode associé à chaque fichier/répertoire.

```
$ ls -lai /etinfo/users/obo
total 1584396
24182823 drwx------ 78 obo  stafinfo       4096 Mar 17 00:34 .
       2 drwxr-xr-x 93 root root           4096 Feb 22 11:37 ..
24190937 -rwxr-xr-x  1 obo  stafinfo      11490 Feb 28 00:43 a.out
24183726 -rw-------  1 obo  stafinfo       4055 Mar 22 15:13 .bash_history
24183731 -rw-r--r--  1 obo  stafinfo         55 Sep 18  1995 .bash_profile
24183732 -rw-r--r--  1 obo  stafinfo        101 Aug 28  2003 .bashrc
24183523 drwxr-xr-x  2 obo  stafinfo       4096 Nov 22  2004 bin
24190938 -rw-r--r--  1 obo  stafinfo        346 Feb 13 15:37 hello.c
48365569 drwxr-xr-x  3 obo  stafinfo       4096 Mar  2 09:30 sinf1252
48791553 drwxr-xr-x  2 obo  stafinfo       4096 May 17  2011 src
```

Dans un système Unix, que ce soit au niveau du shell ou dans n'importe quel processus écrit par exemple en langage C, les fichiers peuvent être spécifiés de deux façons. La première est d'indiquer le chemin complet depuis la racine qui permet d'accéder au fichier. Le chemin `/etinfo/users/obo` passé comme argument à la commande ls(1) ci-dessus en est un exemple. Le premier caractère `/` correspond à la racine du système de fichiers et ensuite ce caractère est utilisé comme séparateur entre les répertoires successifs. Ainsi, le fichier `/etinfo/users/obo/hello.c` est un fichier qui a comme nom `hello.c` qui se trouve dans un répertoire nommé `obo` qui lui-même se trouve dans le répertoire `users` qui est dans le répertoire baptisé `etinfo` dans le répertoire racine. La seconde façon de spécifier un nom de fichier est de préciser son nom relatif. Pour éviter de forcer l'utilisateur à spécifier chaque fois le nom complet des fichiers et répertoires auxquels il veut accéder, le kernel maintient dans sa table des processus le *répertoire courant* de chaque processus. Par défaut, lorsqu'un processus est lancé, son répertoire courant est le répertoire à partir duquel le programme a été lancé. Ainsi, lorsque l'utilisateur tape une commande comme `gcc hello.c` depuis son shell, le processus gcc(1) peut directement accéder au fichier `hello.c` qui se situe dans le répertoire courant. Un processus peut modifier son répertoire courant en utilisant l'appel système chdir(2).

```
#include <unistd.h>

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

Cet appel système prend comme argument une chaîne de caractères contenant le nom du nouveau répertoire courant. Ce nom peut être soit un nom complet (commençant par `/`), ou un nom relatif au répertoire courant actuel. Dans ce cas, il est parfois utile de pouvoir référer au répertoire parent du répertoire courant. Cela se fait en utilisant l'alias `..` qui dans chaque répertoire correspond au répertoire parent. Ainsi, si le répertoire courant est `/etinfo/users`, alors le répertoire `../../bin` est le répertoire `bin` se trouvant dans le répertoire racine. Depuis le shell, il est possible de modifier le répertoire courant avec la commande cd(1posix). La commande pwd(1) affiche le répertoire courant actuel.

Il existe plusieurs appels systèmes et fonctions de la librairie standard qui permettent de parcourir le système de fichiers. Les principaux sont :

- l'appel système stat(2) permet de récupérer les méta-données qui sont associées à un fichier ou un répertoire. La commande stat(1) fournit des fonctionnalités similaires depuis le shell.
- les appels systèmes chmod(2) et chown(2) permettent de modifier respectivement le mode (i.e. les permissions), le propriétaire et le groupe associés à un fichier. Les commandes chmod(1), chown(1) et chgrp(1) permettent de faire de même depuis le shell.
- l'appel système utime(2) permet de modifier les timestamps associés à un fichier/répertoire. Cet appel système est utilisé par la commande touch(1)
- l'appel système rename(2) permet de changer le nom d'un fichier ou d'un répertoire. Il est utilisé notamment par la commande rename(1)
- l'appel système mkdir(2) permet de créer un répertoire alors que l'appel système rmdir(2) permet d'en supprimer un
- les fonctions de la librairie opendir(3), closedir(3), et readdir(3) permettent de consulter le contenu de répertoires.

Les fonctions de manipulation des répertoires méritent que l'on s'y attarde un peu. Un répertoire est un fichier qui a une structure spéciale. Ces trois fonctions permettent d'en extraire de l'information en respectant le format d'un répertoire. Pour accéder à un répertoire, il faut d'abord l'ouvrir en utilisant opendir(3). La fonction readdir(3) permet d'accéder aux différentes entrées de ce répertoire et closedir(3) doit être utilisée lorsque l'accès n'est plus nécessaire. La fonction readdir(3) permet de manipuler la structure `dirent` qui est définie dans bits/dirent.h.

```
struct dirent {
     ino_t          d_ino;       /* inode number */
     off_t          d_off;       /* offset to the next dirent */
     unsigned short d_reclen;    /* length of this record */
     unsigned char  d_type;      /* type of file; not supported
                                    by all file system types */
     char           d_name[256]; /* filename */
 };
```

Cette structure comprend le numéro d'inode contenu dans ses deux premiers membres, la longueur de l'entrée, son type et le nom de l'entrée dans le répertoire. Chaque appel à readdir(3) retourne un pointeur vers une structure de ce type.

L'extrait de code ci-dessous permet de lister tous les fichiers présents dans le répertoire `name`.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>

void exit_on_error(char *s) {
  perror(s);
  exit(EXIT_FAILURE);
}

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

  DIR *dirp;
  struct dirent *dp;
  char name[]=".";
  dirp = opendir(name);
  if(dirp==NULL) {
    exit_on_error("opendir");
  }
  while ((dp = readdir(dirp)) != NULL) {
    printf("%s\n",dp->d_name);
  }
  int err = closedir(dirp);
  if(err<0) {
    exit_on_error("closedir");
  }


}
```

La lecture d'un répertoire avec readdir(3) commence au début de ce répertoire. A chaque appel à readdir(3), le programme appelant récupère un pointeur vers une zone mémoire contenant une structure `dirent` avec l'entrée suivante du répertoire ou `NULL` lorsque la fin du répertoire est atteinte. Si une fonction doit relire à nouveau un répertoire, cela peut se faire en utilisant seekdir(3) ou rewinddir(3).

Note : readdir(3) et les threads

La fonction readdir(3) est un exemple de fonction non-réentrante qu'il faut éviter d'utiliser dans une application multithreadée dont plusieurs threads doivent pouvoir parcourir le même répertoire. Ce problème est causé par l'utilisation d'une zone de mémoire `static` afin de stocker la structure dont le pointeur est retourné par readdir(3). Dans une application utilisant plusieurs threads, il faut utiliser la fonction readdir_r(3) :

```
int  readdir_r(DIR *restrict dirp, struct dirent *restrict entry,
               struct dirent **restrict result);
```

Cette fonction prend comme arguments le pointeur `entry` vers un buffer propre à l'appelant qui permet de stocker le résultat de readdir_r(3).

Les appels systèmes link(2) et unlink(2) sont un peu particuliers et méritent une description plus détaillée. Sous Unix, un *inode* est associé à chaque fichier mais l'*inode* ne contient pas le nom de fichier parmi les méta-données qu'il stocke. Par contre, chaque *inode* contient un compteur (`nlinks`) du nombre de liens vers un fichier. Cela permet d'avoir une seule copie d'un fichier qui est accessible depuis plusieurs répertoires. Pour comprendre cette utilisation des liens sur un système de fichiers Unix, considérons le scénario suivant.

```
$ mkdir a
$ mkdir b
$ cd a
$ echo "test" > test.txt
$ cd ..
$ ln a/test.txt a/test2.txt
$ ls -li a
total 16
9624126 -rw-r--r--  3 obo  stafinfo  5 24 mar 21:14 test.txt
9624126 -rw-r--r--  3 obo  stafinfo  5 24 mar 21:14 test2.txt
$ ln a/test.txt b/test3.txt
$ stat --format "inode=%i nlinks=%h" b/test3.txt
inode=9624126 nlinks=3
$ ls -li b
total 8
9624126 -rw-r--r--  3 obo  stafinfo  5 24 mar 21:14 test3.txt
$ echo "complement" >> b/test3.txt
$ ls -li a
total 16
9624126 -rw-r--r--  3 obo  stafinfo  16 24 mar 21:15 test.txt
9624126 -rw-r--r--  3 obo  stafinfo  16 24 mar 21:15 test2.txt
$ ls -li b
total 8
9624126 -rw-r--r--  3 obo  stafinfo  16 24 mar 21:15 test3.txt
$ cat b/test3.txt
test
complement
$ cat a/test.txt
test
complement
$ rm a/test2.txt
$ ls -li a
total 8
9624126 -rw-r--r--  2 obo  stafinfo  16 24 mar 21:15 test.txt
$ rm a/test.txt
$ ls -li a
$ ls -li b
total 8
9624126 -rw-r--r--  1 obo  stafinfo  16 24 mar 21:15 test3.txt
```

Dans ce scénario, deux répertoires sont créés avec la commande mkdir(1). Ensuite, la commande echo(1) est utilisée pour créer le fichier `test.txt` contenant la chaîne de caractères `test` dans le répertoire `a`. Ce fichier est associé à l'*inode* `9624126`. La commande ln(1) permet de rendre ce fichier accessible sous un autre nom depuis le même répertoire. La sortie produite par la commande ls(1) indique que ces deux fichiers qui sont présents dans le répertoire `a` ont tous les deux le même `inode`. Ils correspondent donc aux mêmes données sur le disque. A ce moment, le compteur `nlinks` de l'*inode* `9624126` a la valeur `2`. La commande ln(1) peut être utilisée pour créer un lien vers un fichier qui se trouve dans un autre répertoire [4] comme le montre la création du fichier `test3.txt` dans le répertoire `b`. Ces trois fichiers correspondant au même *inode*, toute modification à l'un des fichiers affecte et est visible dans n'importe lequel des liens vers ce fichier. C'est ce que l'on voit lorsque la commande `echo "complement" >> b/test3.txt` est exécutée. Cette commande affecte immédiatement les trois fichiers. La commande `rm a/test2.txt` efface la référence du fichier sous le nom `a/test2.txt`, mais les deux autres liens restent accessibles. Le fichier ne sera réellement effacé qu'après que le dernier lien vers l'*inode* correspondant aie été supprimé. La commande rm(1) utilise en pratique l'appel système unlink(2)

4. Dans un système de fichiers Unix, un lien ne peut être créé avec ln(1) ou link(2) que lorsque les deux répertoires concernés sont situés sur le même système de fichiers. Si ce n'est pas le cas, il faut utiliser un *lien symbolique*. Ceux-ci peuvent être créés en utilisant l'appel système symlink(2) ou via la commande ln(1) avec l'argument `-s`.

qui en toute généralité décrémente le compteur de liens de l'inode correspondant au fichier et l'efface lorsque ce compteur atteint la valeur `0`.

Une description détaillée du fonctionnement de ces appels systèmes et fonctions de la librairie standard peut se trouver dans les livres de référence sur la programmation en C sous Unix [Kerrisk2010], [Mitchell+2001], [StevensRago2008].

5.2.1 Utilisation des fichiers

Si quelques processus manipulent le système de fichiers et parcourent les répertoires, les processus qui utilisent des données sauvegardées dans des fichiers sont encore plus nombreux. Un système Unix offre deux possibilités d'écrire et de lire dans un fichier. La première utilise directement les appels systèmes open(2), read(2)/ write(2) et close(2). La seconde s'appuie sur les fonctions fopen(3), fread(3)/ fwrite(3) et fclose(3) de la librairie stdio(3). Seuls les appels systèmes sont traités dans ce cours. Des détails complémentaires sur les fonctions de la libraire peuvent être obtenus dans [Kerrisk2010], [Mitchell+2001] ou [StevensRago2008].

Du point de vue des appels systèmes de manipulation des fichiers, un fichier est une séquence d'octets. Avant qu'un processus ne puisse écrire ou lire dans un fichier, il doit d'abord demander au système d'exploitation l'autorisation d'accéder au fichier. Cela se fait en utilisant l'appel système open(2).

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

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

Il existe deux variantes de l'appel système open(2). La première permet d'ouvrir des fichiers existants. Elle prend deux arguments. La deuxième permet de créer un nouveau fichier et d'ensuite l'ouvrir. Elle prend trois arguments. Le premier argument est le nom absolu ou relatif du fichier dont l'ouverture est demandée. Le deuxième argument est un entier qui contient un ensemble de drapeaux binaires qui précisent la façon dont le fichier doit être ouvert. Ces drapeaux sont divisés en deux groupes. Le premier groupe est relatif à l'accès en lecture et/ou en écriture du fichier. Lors de l'ouverture d'un fichier avec open(2), il est nécessaire de spécifier l'un des trois drapeaux d'accès suivants :

- `O_RDONLY` : indique que le fichier est ouvert uniquement en lecture. Aucune opération d'écriture ne sera effectuée sur le fichier.
- `O_WRONLY` : indique que le fichier est ouvert uniquement en écriture. Aucune opération de lecture ne sera effectuée sur le fichier.
- `O_RDWR` : indique que le fichier est ouvert pour des opérations de lecture et d'écriture.

En plus de l'un des trois drapeaux ci-dessus, il est également possible de spécifier un ou plusieurs drapeaux optionnels. Ces drapeaux sont décrits en détails dans la page de manuel open(2). Les plus utiles sont probablement :

- `O_CREAT` : indique que si le fichier n'existe pas, il doit être créé lors de l'exécution de l'appel système open(2). L'appel système creat(2) peut également être utilisé pour créer un nouveau fichier. Lorsque le drapeau `O_CREAT` est spécifié, l'appel système open(2) prend comme troisième argument les permissions du fichier qui doit être créé. Celles-ci sont spécifiées de la même façon que pour l'appel système chmod(2). Si elles ne sont pas spécifiées, le fichier est ouvert avec comme permissions les permissions par défaut du processus définies par l'appel système umask(2)
- `O_APPEND` : indique que le fichier est ouvert de façon à ce que les données écrites dans le fichier par l'appel système write(2) s'ajoutent à la fin du fichier.
- `O_TRUNC` : indique que si le fichier existe déjà et qu'il est ouvert en écriture, alors le contenu du fichier doit être supprimé avant que le processus ne commence à y accéder.
- `O_CLOEXEC` : ce drapeau qui est spécifique à Linux indique que le fichier doit être automatiquement fermé lors de l'exécution de l'appel système execve(2). Normalement, les fichiers qui ont été ouverts par open(2) restent ouverts lors de l'exécution de execve(2).
- `O_SYNC` : ce drapeau indique que toutes les opérations d'écriture sur le fichier doivent être effectuées immédiatement sur le dispositif de stockage sans être mises en attente dans les buffers du noyau du système d'exploitation

Ces différents drapeaux binaires doivent être combinés en utilisant une disjonction logique entre les différents drapeaux. Ainsi, `O_CREAT|O_RDWR` correspond à l'ouverture d'un fichier qui doit à la fois être créé si il n'existe pas et ouvert en lecture et écriture.

Lors de l'exécution de open(2), le noyau du système d'exploitation vérifie si le processus qui exécute l'appel système dispose des permissions suffisantes pour accéder au fichier. Si oui, le système d'exploitation ouvre le fichier et retourne au processus appelant le *descripteur de fichier* correspondant. Si non, le processus récupère une valeur de retour négative et `errno` indique le type d'erreur.

Sous Unix, un *descripteur de fichier* est représenté sous la forme d'un entier positif. L'appel système open(2) retourne toujours le plus petit *descripteur de fichier* disponible. Par convention,

- `0` est le *descripteur de fichier* correspondant à l'entrée standard.
- `1` est le *descripteur de fichier* correspondant à la sortie standard.
- `2` est le *descripteur de fichier* correspondant à la sortie d'erreur standard.

Si l'appel système open(2) échoue, il retourne `-1` comme *descripteur de fichier* et `errno` donne plus de précisions sur le type d'erreur. Il peut s'agir d'une erreur liée aux droits d'accès au fichier (`EACESS`), une erreur de drapeau (`EINVAL`) ou d'une erreur d'entrée sortie lors de l'accès au dispositif de stockage (`EIO`). Le noyau du système d'exploitation maintient une table de l'ensemble des fichiers qui sont ouverts par tous les processus actifs. Si cette table est remplie, il n'est plus possible d'ouvrir de nouveau fichier et open(2) retourne une erreur. Il en va de même si le processus tente d'ouvrir plus de fichiers que le nombre maximum de fichiers ouverts qui est autorisé.

Note : Seul open(2) vérifie les permissions d'accès aux fichiers

Sous Unix, seul l'appel système open(2) vérifie qu'un processus dispose des permissions suffisantes pour accéder à un fichier qui est ouvert. Si les permissions ou le propriétaire d'un fichier change alors que ce fichier est ouvert par un processus, ce processus continue à pouvoir y accéder sans être affecté par la modification de droits. Il en va de même lorsqu'un fichier est effacé avec l'appel système unlink(2). Si un processus utilisait le fichier qui est effacé, il continue à pouvoir l'utiliser même si le fichier n'apparaît plus dans le répertoire.

Toutes les opérations qui sont faites sur un fichier se font en utilisant le *descripteur de fichier* comme référence au fichier. Un *descripteur de fichier* est une ressource limitée dans un système d'exploitation tel que Unix et il est important qu'un processus n'ouvre pas inutilement un grand nombre de fichiers [5] et ferme correctement les fichiers ouverts lorsqu'il ne doit plus y accéder. Cela se fait en utilisant l'appel système close(2). Celui-ci prend comme argument le *descripteur de fichier* qui doit être fermé.

```
#include <unistd.h>

int close(int fd);
```

Tout processus doit correctement fermer tous les fichiers qu'il a utilisé. Par défaut, le système d'exploitation ferme automatiquement les descripteurs de fichiers correspondant `0`, `1` et `2` lorsqu'un processus se termine. Les autres descripteurs de fichiers doivent être explicitement fermés par le processus. Si nécessaire, cela peut se faire en enregistrant une fonction permettant de fermer correctement les fichiers ouverts via atexit(3). Il faut noter que par défaut un appel à execve(2) ne ferme pas les descripteurs de fichiers ouverts par le processus. C'est nécessaire pour permettre au programme exécuté d'avoir les entrées et sorties standard voulues.

Lorsqu'un fichier a été ouvert, le noyau du système d'exploitation maintient outre les références vers l'*inode* du fichier un *offset pointer*. Cet *offset pointer* est la position actuelle de la tête de lecture/écriture du fichier. Lorsqu'un fichier est ouvert, son *offset pointer* est positionné au premier octet du fichier, sauf si le drapeau `O_APPEND` a été spécifié lors de l'ouverture du fichier, dans ce cas l'*offset pointer* est positionné juste après le dernier octet du fichier de façon à ce qu'une écriture s'ajoute à la suite du fichier.

Les deux appels systèmes permettant de lire et d'écrire dans un fichier sont respectivement read(2) et write(2).

```
#include <unistd.h>

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

5. Il y a une limite maximale au nombre de fichiers qui peuvent être ouverts par un processus. Cette limite peut être récupérée avec l'appel système getdtablesize(2).

Ces deux appels systèmes prennent trois arguments. Le premier est le *descripteur du fichier* sur lequel l'opération doit être effectuée. Le second est un pointeur `void *` vers la zone mémoire à lire ou écrire et le dernier est la quantité de données à lire/écrire. Si l'appel système réussit, il retourne le nombre d'octets qui ont été écrits/lus et sinon une valeur négative et la variable `errno` donne plus de précisions sur le type d'erreur. read(2) retourne `0` lorsque la fin du fichier a été atteinte.

Il est important de noter que read(2) et write(2) permettent de lire et d'écrire des séquences contigües d'octets. Lorsque l'on écrit ou lit des chaînes de caractères dans lesquels chaque caractère est représenté sous la forme d'un byte, il est possible d'utiliser read(2) et write(2) pour lire et écrire d'autres types de données que des octets comme le montre l'exemple ci-dessous.

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

void exit_on_error(char *s) {
  perror(s);
  exit(EXIT_FAILURE);
}

int main (int argc, char *argv[])  {
  int n=1252;
  int n2;
  short ns=1252;
  short ns2;
  long nl=125212521252;
  long nl2;
  float f=12.52;
  float f2;
  char *s="sinf1252";
  char *s2=(char *) malloc(strlen(s)*sizeof(char )+1);
  int err;
  int fd;

  fd=open("test.dat",O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
  if(fd==-1) {
    perror("open");
    exit(EXIT_FAILURE);
  }
  if( write(fd, (void *) s, strlen(s)) == -1 )
    exit_on_error("write s");
  if (write(fd, (void *) &n, sizeof(int )) == -1)
    exit_on_error("write n");
  if (write(fd, (void *) &ns, sizeof(short int))==-1)
    exit_on_error("write ns");
  if (write(fd, (void *) &nl, sizeof(long int))==-1)
    exit_on_error("write nl");
  if (write(fd, (void *) &f, sizeof(float))==-1)
    exit_on_error("write f");
  if (close(fd)==-1)
    exit_on_error("close ");

  // lecture
  fd=open("test.dat",O_RDONLY);
  if(fd==-1) {
    perror("open");
    exit(EXIT_FAILURE);
  }
```

```
  printf("Fichier ouvert\n");

  if(read(fd, (void *) s2, strlen(s))==-1)
    exit_on_error("read s");
  printf("Donnée écrite : %s, lue: %s\n",s,s2);

  if(read(fd, (void *) &n2, sizeof(int))==-1)
    exit_on_error("read n");
  printf("Donnée écrite : %d, lue: %d\n",n,n2);

  if(read(fd, (void *) &ns2, sizeof(short))==-1)
    exit_on_error("read ns");
  printf("Donnée écrite : %d, lue: %d\n",ns,ns2);

  if(read(fd, (void *) &nl2, sizeof(long))==-1)
    exit_on_error("read nl");
  printf("Donnée écrite : %ld, lue: %ld\n",nl,nl2);

  if(read(fd, (void *) &f2, sizeof(float))==-1)
    exit_on_error("read f");
  printf("Donnée écrite : %f, lue: %f\n",f,f2);
  err=close(fd);
  if(err==-1){
   perror("close");
    exit(EXIT_FAILURE);
  }

  return(EXIT_SUCCESS);
}
```

Lors de son exécution, ce programme affiche la sortie ci-dessous.

```
Fichier ouvert
Donnée écrite : sinf1252, lue: sinf1252
Donnée écrite : 1252, lue: 1252
Donnée écrite : 1252, lue: 1252
Donnée écrite : 125212521252, lue: 125212521252
Donnée écrite : 12.520000, lue: 12.520000
```

Si il est bien possible de sauvegarder dans un fichier des entiers, des nombres en virgule flottante voire même des structures, il faut être bien conscient que l'appel système write(2) se contente de sauvegarder sur le disque le contenu de la zone mémoire pointée par le pointeur qu'il a reçu comme second argument. Si comme dans l'exemple précédent c'est le même processus qui lit les données qu'il a écrit, il pourra toujours récupérer les données correctement.

Par contre, lorsqu'un fichier est écrit sur un ordinateur, envoyé via Internet et lu sur un autre ordinateur, il peut se produire plusieurs problèmes dont il faut être conscient. Le premier problème est que deux ordinateurs différents n'utilisent pas nécessairement le même nombre d'octets pour représenter chaque type de données. Ainsi, sur un ordinateur équipé d'un ancien processeur [IA32], les entiers sont représentés sur 32 bits (i.e. 4 bytes) alors que sur les processeurs plus récents ils sont souvent représentés sur 64 bits (i.e. 8 bytes). Cela implique qu'un tableau de 100 entiers en 32 bits sera interprété comme un tableau de 50 entiers en 64 bits.

Le second problème est que les fabricants de processeurs ne se sont pas mis d'accord sur la façon dont il fallait représenter les entiers sur 16 et 32 bits en mémoire. Il y a deux techniques qui sont utilisées : *big endian* et *little endian*.

Pour comprendre ces deux techniques, regardons comment l'entier 16 bits `0b1111111100000000` est stocké en mémoire. En *big endian*, le byte `11111111` sera stocké à l'adresse *x* et le byte `00000000` à l'adresse *x+1*. En *little endian*, c'est le byte `00000000` qui est stocké à l'adresse *x* et le byte `11111111` qui est stocké à l'adresse *x+1*. Il en va de même pour les entiers encodés sur 32 bits comme illustré dans les deux figures ci-dessous [6].

6. Source : http://en.wikipedia.org/wiki/Endianness

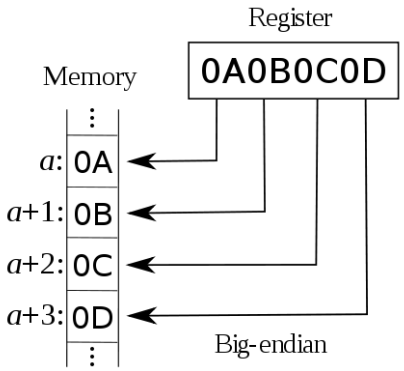


FIGURE 5.1 – Ecriture d'un entier 32 bits en mémoire *big endian*

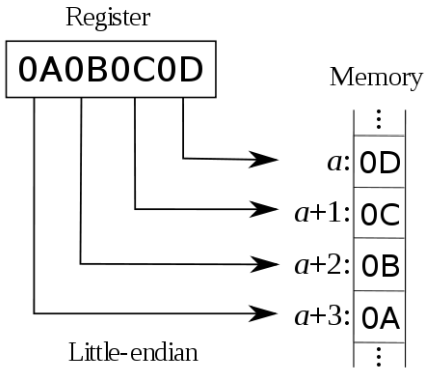


FIGURE 5.2 – Ecriture d'un entier 32 bits en mémoire *little endian*

Pour les nombres en virgule flottante, ce problème ne se pose heureusement pas car tous les processeurs actuels utilisent la même norme pour représenter les nombres en virgule flottant en mémoire.

Les processeurs [IA32] utilisent la représentation *little endian* tandis que les PowerPC utilisent *big endian*. Certains processeurs sont capables d'utiliser les deux représentations.

Il est également possible en utilisant l'appel système lseek(2) de déplacer l'*offset pointer* associé à un *descripteur de fichier*.

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

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

Cet appel système prend trois arguments. Le premier est le *descripteur de fichier* dont l'*offset pointer* doit être modifié. Le second est un entier qui est utilisé pour le calcul de la nouvelle position *offset pointer* et le troisième indique comment l'*offset pointer* doit être calculé. Il y a trois modes de calcul possibles pour l'*offset pointer* :

- `whence==SEEK_SET` : dans ce cas, le deuxième argument de l'appel système indique la valeur exacte du nouvel *offset pointer*
- `whence==SEEK_CUR` : dans ce cas, le nouvel *offset pointer* sera sa position actuelle à laquelle le deuxième argument aura été ajouté
- `whence==SEEK_END` : dans ce cas, le nouvel *offset pointer* sera la fin du fichier à laquelle le deuxième argument aura été ajouté

Note : Fichiers temporaires

Il est parfois nécessaire dans un programme de créer des fichiers temporaires qui sont utilisés pour effectuer des opérations dans le processus sans pour autant être visible dans d'autres processus et sur le système de fichiers. Il est possible d'utiliser open(2) pour créer un tel fichier temporaire, mais il faut dans ce cas prévoir tous les cas d'erreur qui peuvent se produire lorsque par exemple plusieurs instances du même programme s'exécutent au même moment. Une meilleure solution est d'utiliser la fonction de la librairie mkstemp(3). Cette fonction prend comme argument un template de nom de fichier qui se termine par `XXXXXX` et génère un nom de fichier unique et retourne un descripteur de fichier associé à ce fichier. Elle s'utilise généralement comme suit :

```
char template[]="/tmp/sinf1252PROCXXXXXX"

int fd=mkstemp(template);
if(fd==-1)
  exit_on_error("mkstemp");
// template contient le nom exact du fichier généré
unlink(template);
// le fichier est effacé, mais reste accessible
// via son descripteur jusqu'à close(fd)

/ Accès au fichier avec read et write

if(close(fd)==-1)
  exit_on_error("close");
// le fichier n'est plus accessible
```

L'utilisation de unlink(2) permet de supprimer le fichier du système de fichiers dès qu'il a été créé. Ce fichier reste cependant accessible au processus tant que celui-ci dispose d'un descripteur de fichier qui y est associé.

Note : Duplication de descripteurs de fichiers

Dans certains cas il est utile de pouvoir dupliquer un descripteur de fichier. C'est possible avec les appels systèmes dup(2) et dup2(2). L'appel système dup(2) prend comme argument un descripteur de fichier et retourne le plus petit descripteur de fichier libre. Lorsqu'un descripteur de fichier a été dupliqué avec dup(2) les deux descripteurs de fichiers partagent le même *offset pointer* et les mêmes modes d'accès au fichier.

5.2.2 Les pipes

Les appels systèmes de manipulation des fichiers permettent d'accéder à des données sur des dispositifs de stockage, mais ils peuvent aussi être utilisés pour échanger des informations entre processus. Les systèmes d'exploitation de la famille Unix supportent plusieurs mécanismes permettant à des processus de communiquer. Le plus simple est le *pipe*. Un *pipe* est un flux de bytes unidirectionnel qui relie deux processus qui ont un ancêtre commun. L'un des processus peut écrire sur le *pipe* et le second peut lire sur le *pipe*. Un *pipe* est créé en utilisant l'appel système pipe(2).

```
#include <unistd.h>

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

Cet appel système prend comme argument un tableau permettant de stocker deux descripteurs de fichiers. Ces deux descripteurs de fichiers sont utilisés pour respectivement lire et écrire sur le *pipe*. `fd[0]` est le descripteur de fichier sur lequel les opérations de lecture seront effectuées tandis que les opérations d'écriture se feront sur `fd[1]`. Chaque fois qu'une donnée est écrite sur `fd[1]` avec l'appel système `write(fd[1],...)`, elle devient disponible sur le descripteur de fichiers `fd[0]` et peut être lue avec `read(fd[0],...)`. Même si il est possible de créer un *pipe* dans un processus unique, pipe(2) s'utilise en général entre un père et son fils. Le programme ci-dessous illustre cette utilisation de pipes pour permettre à un processus père d'échanger de l'information avec son processus fils.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <stdbool.h>

void exit_on_error(char *s) {
  perror(s);
  exit(EXIT_FAILURE);
}


int main (int argc, char *argv[])  {
  int status;
  pid_t pid;
  int fd[2];
  int err;

  if ( pipe(fd)==-1)
    exit_on_error("pipe");

  pid=fork();

  if (pid==-1) {
    // erreur à l'exécution de fork
    perror("fork");
    exit(EXIT_FAILURE);
  }
  // pas d'erreur
  if (pid==0) {
    int count=0;
    int finished=false;
    int n;
    // processus fils, lecture du pipe
    if(close(fd[1])==-1)
      exit_on_error("close");
    while ( !finished) {
      err=read(fd[0],(void *) &n,sizeof(int));
      if(err==sizeof(int))
```

```
        // entier reçu
        count++;
      else
        if(err==0)
          // fin de fichier
          finished=true;
        else
          exit_on_error("read");
    }
    if(close(fd[0])==-1)
      exit_on_error("close");
    printf("Reçu : %d entiers\n",count);
    return(EXIT_SUCCESS);
  }
  else {
    // processus père
    if(close(fd[0])==-1)
      exit_on_error("close");
    for(int i=0;i<100000;i++) {
      if(write(fd[1],(void *) &i,sizeof(int))==-1)
        exit_on_error("write");
    }
    if( close(fd[1])==-1)
      exit_on_error("close(fd[1])");

    int fils=waitpid(pid,&status,0);
    if(fils==-1) {
      perror("wait");
      exit(EXIT_FAILURE);
    }
    // fils terminé correctement
  }
}
```

Dans cet exemple, le processus père crée d'abord un *pipe*. Ensuite, il crée un fils avec fork(2). Comme l'exécution de fork(2) ne ferme pas les descripteurs de fichiers ouverts, le processus père et le processus fils ont accès au *pipe* via ses deux descripteurs. Le développeur de ce programme a choisi que le processus père écrirait sur le *pipe* tandis que le fils l'utiliserait uniquement en lecture. Le père (resp. fils) ferme donc le descripteur de fichier d'écriture (resp. lecture). C'est une règle de bonne pratique qui permet souvent d'éviter des bugs. Le processus père envoie ensuite des entiers sur `fd[1]` puis ferme ce descripteur de fichier. Le processus fils quant à lui lit les entiers reçus sur le *pipe* via `fd[0]`. Le processus fils peut détecter la fermeture du descripteur `fd[1]` du *pipe* par le père en analysant la valeur de retour de read(2). En effet, read(2) retourne une valeur `0` lorsque la dernière donnée envoyée sur le *pipe* a été lue. Le fils ferme proprement le descripteur de fichier `fd[0]` avant de se terminer.

En pratique, les pipes sont utilisés notamment par le shell. En effet, lorsqu'une commande telle que `cat test.txt | grep "student"` est exécutée, le shell relie via un *pipe* la sortie standard de `cat` avec l'entrée standard de `grep`.

Sur un système Unix, il est fréquent que des processus différents doivent communiquer entre eux et coordonner leurs activités. Dans ce chapitre, nous détaillons plusieurs mécanismes permettant à des processus ayant un ancêtre commun ou non de coordonner leurs activités.

Historiquement, le premier mécanisme de coordination entre processus a été l'utilisation des signaux. Nous avons déjà utilisé des signaux sans les avoir détaillés explicitement pour terminer des processus avec la commande kill(1). Nous décrivons plus en détails les principales utilisations des signaux dans ce chapitre.

Lorsque deux processus ont un ancêtre commun, il est possible de les relier en utilisant un pipe(2). Si ils n'ont pas d'ancêtre commun, il est possible d'utiliser une *fifo* pour obtenir un effet similaire. Une *fifo* peut être créée en utilisant la commande mkfifo(1) ou la fonction mkfifo(3). mkfifo(3) s'utilise comme l'appel système open(2). Pour créer une nouvelle *fifo*, il suffit de lui donner un nom sous la forme d'une chaîne de caractères et tout processus qui connait le nom de la *fifo* pourra l'utiliser.

Les sémaphores que nous avons utilisés pour coordonner plusieurs threads sont également utilisables entre pro-

cessus moyennant quelques différences que nous détaillerons.

Enfin, des processus liés ou non doivent parfois accéder aux mêmes fichiers. Ces accès concurrents, si ils ne sont pas correctement coordonnés, peuvent conduire à des corruptions de fichiers. Nous présenterons les mécanismes de locking qui sont utilisés dans Unix et Linux pour coordonner l'accès à des fichiers.

5.3 Signaux

L'envoi et la réception de signaux est le mécanisme de communication entre processus le plus primitif sous Unix. Un *signal* est une forme d'interruption logicielle [StevensRago2008]. Comme nous l'avons vu précédemment, un microprocesseur utilise les interruptions pour permettre au système d'exploitation de réagir aux événements imprévus qui surviennent. Un *signal* Unix est un mécanisme qui permet à un processus de réagir de façon asynchrone à un événement qui s'est produit. Certains de ces événements sont directement liés au fonctionnement du matériel. D'autres sont provoqués par le processus lui-même ou un autre processus s'exécutant sur le système.

Pour être capable d'utiliser les signaux à bon escient, il est important de bien comprendre comment ceux-ci sont implémentés dans le système d'exploitation.

Il existe deux types de signaux.

— Un *signal synchrone* est un *signal* qui a été directement causé par l'exécution d'une instruction du processus. Un exemple typique de *signal synchrone* est le signal `SIGFPE` qui est généré par le système d'exploitation lorsqu'un processus provoque une exception lors du calcul d'expressions mathématiques. C'est le cas notamment lors d'une division par zéro. La sortie ci-dessous illustre ce qu'il se produit lors de l'exécution du programme `/Fichiers/src/sigfpe.c`.

```
$ ./sigfpe
Calcul de : 1252/0
Floating point exception
```

— Un *signal asynchrone* est un *signal* qui n'a pas été directement causé par l'exécution d'une instruction du processus. Il peut être produit par le système d'exploitation ou généré par un autre processus comme lorsque nous avons utilisé kill(1) dans un shell pour terminer un processus.

Le système Unix fournit plusieurs appels systèmes qui permettent de manipuler les signaux. Un processus peut recevoir des signaux synchrones ou asynchrones. Pour chaque signal, le système d'exploitation définit un traitement par défaut. Pour certains signaux, leur réception provoque l'arrêt du processus. Pour d'autres, le signal est ignoré et le processus peut continuer son exécution. Un processus peut redéfinir le traitement des signaux qu'il reçoit en utilisant l'appel système signal(2). Celui-ci permet d'associer un *handler* ou fonction de traitement de signal à chaque signal. Lorsqu'un *handler* a été associé à un signal, le système d'exploitation exécute ce *handler* dès que ce signal survient. Unix permet également à un processus d'envoyer un signal à un autre processus en utilisant l'appel système kill(2).

Avant d'analyser en détails le fonctionnement des appels systèmes signal(2) et kill(2), il est utile de parcourir les principaux signaux. Chaque *signal* est identifié par un entier positif et signal.h définit des constantes pour chaque signal reconnu par le système. Sous Linux, les principaux signaux sont :

— `SIGALRM`. Ce signal survient lorsqu'une alarme définie par la fonction alarm(3posix) ou l'appel système setitimer(2) a expiré. Par défaut, la réception de ce signal provoque la terminaison du processus.
— `SIGBUS`. Ce signal correspond à une erreur au niveau matériel. Par défaut, la réception de ce signal provoque la terminaison du processus.
— `SIGSEGV`. Ce signal correspond à une erreur dans l'accès à la mémoire, typiquement une tentative d'accès en dehors de la zone mémoire allouée au processus. Par défaut, la réception de ce signal provoque la terminaison du processus.
— `SIGFPE`. Ce signal correspond à une erreur au niveau de l'utilisation des fonctions mathématiques, notamment en virgule flottante mais pas seulement. Par défaut, la réception de ce signal provoque la terminaison du processus.
— `SIGTERM`. Ce signal est le signal utilisé par défaut par la commande kill(1) pour demander la fin d'un processus. Par défaut, le réception de ce signal provoque la terminaison du processus.
— `SIGKILL`. Ce signal permet de forcer la fin d'un processus. Alors qu'un processus peut définir un handler pour le signal `SIGTERM`, il n'est pas possible d'en définir un pour `SIGKILL`. Ce signal est le seul qui ne peut être traité et ignoré par un processus.

— `SIGUSR1` et `SIGUSR2` sont deux signaux qui peuvent être utilisés par des processus sans conditions particulières. Par défaut, la réception d'un tel signal provoque la terminaison du processus.
— `SIGCHLD`. Ce signal indique qu'un processus fils s'est arrêté ou a fini son exécution. Par défaut ce signal est ignoré.
— `SIGHUP`. Aux débuts de Unix, ce signal servait à indiquer que la connexion avec le terminal avait été rompue. Aujourd'hui, il est parfois utilisé par des processus serveurs qui rechargent leur fichier de configuration lorsqu'ils reçoivent ce signal.
— `SIGINT`. Ce signal est envoyé par le shell lorsque l'utilisateur tape *Ctrl-C* pendant l'exécution d'un programme. Il provoque normalement la terminaison du processus.

Une description détaillée des différents signaux sous Unix et Linux peut se trouver dans signal(7), [BryantOHallaron2011], [StevensRago2008] ou encore [Kerrisk2010].

Note : Comment arrêter proprement un processus ?

Trois signaux permettent d'arrêter un processus : `SIGTERM`, `SIGINT` et `SIGKILL`. Lorsque l'on doit forcer l'arrêt d'un processus, il est préférable d'utiliser le signal `SIGTERM` car le processus peut prévoir une routine de traitement de ce signal. Cette routine peut par exemple fermer proprement toutes les ressources (fichiers, ...) utilisées par le processus. Lorsque l'on tape *Ctrl-C* dans le shell, c'est le signal `SIGINT` qui est envoyé au processus qui est en train de s'exécuter. Le signal `SIGKILL` ne peut pas être traité par le processus. Lorsque l'on envoie ce signal à un processus, on est donc certain que celui-ci s'arrêtera. Malheureusement, cet arrêt sera brutal et le processus n'aura pas eu le temps de libérer proprement toutes les ressources qu'il utilisait. Quand un processus ne répond plus, il est donc préférable de d'abord essayer `SIGINT` ou `SIGTERM` et de n'utiliser `SIGKILL` qu'en dernier recours.

5.3.1 Envoi de signaux

Un processus peut envoyer un signal à un autre processus en utilisant l'appel système kill(2).

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

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

Cet appel système prend deux arguments. Le second est toujours le numéro du signal à délivrer ou la constante définissant ce signal dans signal.h. Le premier argument indique à quel(s) processus le signal doit être délivré :
— `pid>0`. Dans ce cas, le signal est délivré au processus ayant comme identifiant `pid`.
— `pid==0`. Dans ce cas, le signal est délivré à tous les processus qui font partie du même groupe de processus [7] que le processus qui exécute l'appel système kill(2).
— `pid==-1`. Dans ce cas, le signal est délivré à tous les processus pour lesquels le processus qui exécute kill(2) a les permissions suffisantes pour leur envoyer un signal.
— `pid<-1`. Dans ce cas, le signal est délivré à tous les processus qui font partie du groupe `abs(pid)`.

Par défaut, un processus ne peut envoyer un signal qu'à un processus qui s'exécute avec les mêmes permissions que le processus qui exécute l'appel système kill(2).

5.3.2 Traitement de signaux

Pour des raisons historiques, il existe deux appels systèmes permettant à un processus de spécifier comment un signal particulier doit être traité. Le premier est l'appel système signal(2). Il prend deux arguments : le numéro du signal dont le traitement doit être modifié et la fonction à exécuter lorsque ce signal est reçu. Le second est sigaction(2). Cet appel système est nettement plus générique et plus complet que signal(2) mais il ne sera pas traité en détails dans ces notes. Des informations complémentaires sur l'utilisation de sigaction(2) peuvent être obtenues dans [Kerrisk2010] ou [StevensRago2008].

7. Chaque processus appartient à groupe de processus. Ce groupe de processus peut être récupéré via l'appel système getpgrp(2). Par défaut, lorsqu'un processus est créé, il appartient au même groupe de processus que son processus père, mais il est possible de changer de groupe de processus en utilisant l'appel système setpgid(2). En pratique, les groupes de processus sont surtout utilisés par le shell. Lorsqu'un utilisateur exécute une commande combinée telle que `cat /tmp/t | ./a.out`, il souhaite pouvoir l'arrêter en tapant sur *Ctrl-C* si nécessaire. Pour cela, il faut pouvoir délivrer le signal `SIGINT` aux processus `cat` et `a.out`.

```c
#include <signal.h>
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

Le premier argument de l'appel système signal(2) est généralement spécifié en utilisant les constantes définies dans signal.h. Le second argument est un pointeur vers une fonction de type `void` qui prend comme argument un entier. Cette fonction est la fonction qui sera exécutée par le système d'exploitation à la réception du signal. Ce second argument peut également être la constante `SIG_DFL` si le signal doit être traité avec le traitement par défaut documenté dans signal(7) et `SIG_IGN` si le signal doit être ignoré. La valeur de retour de l'appel système signal(2) est la façon dont le système d'exploitation gérait le signal passé en argument avant l'exécution de l'appel système (typiquement `SIG_DFL` ou `SIG_IGN`) ou `SIG_ERR` en cas d'erreur.

L'exemple ci-dessous est un programme simple qui compte le nombre de signaux `SIGUSR1` et `SIGUSR2` qu'il reçoit et se termine dès qu'il a reçu cinq signaux.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <stdint.h>
#include <stdio.h>

volatile sig_atomic_t n_sigusr1=0;
volatile sig_atomic_t n_sigusr2=0;

static void sig_handler(int);

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

  if(signal(SIGUSR1,sig_handler)==SIG_ERR) {
    perror("signal");
    exit(EXIT_FAILURE);
  }
  if(signal(SIGUSR2,sig_handler)==SIG_ERR) {
    perror("signal");
    exit(EXIT_FAILURE);
  }

  while( (n_sigusr1+n_sigusr2) <5) {
    // vide
  }

  printf("Fin du processus\n");
  printf("Reçu %d SIGUSR1 et %d SIGUSR2\n",n_sigusr1,n_sigusr2);
  return(EXIT_SUCCESS);
}

static void sig_handler(int signum) {

  if(signum==SIGUSR1) {
    n_sigusr1++;
  }
  else {
    if(signum==SIGUSR2) {
      n_sigusr2++;
    }
    else {
      char *msg="Reçu signal inattendu\n";
      write(STDERR_FILENO,msg,strlen(msg));
      _exit(EXIT_FAILURE);
```

```
    }
  }
}
```

Lors de son exécution, ce programme affiche :

```
$ ./sigusr &
[1] 45602
$ kill -s SIGUSR1 45602
$ kill -s SIGUSR2 45602
$ kill -s SIGUSR2 45602
$ kill -s SIGUSR1 45602
$ kill -s SIGUSR1 45602
$ Fin du processus
Reçu 3 SIGUSR1 et 2 SIGUSR2
[1]+  Done                    ./sigusr
```

Il est intéressant d'analyser le code source du programme ci-dessus. Commençons d'abord par une lecture rapide pour comprendre la logique du programme sans s'attarder sur les détails. La fonction `main` utilise l'appel système signal(2) pour enregistrer un handler pour les signaux `SIGUSR1` et `SIGUSR2`. La fonction `sig_handler` sera exécutée dès réception d'un de ces signaux. Cette fonction prend comme argument le numéro du signal reçu. Cela permet, de traiter plusieurs signaux dans la même fonction. Ensuite, la boucle `while` est une boucle active qui ne se terminera que lorsque la somme des variables `n_sigusr1` et `n_sigusr2` sera égale à 5. Ces deux variables sont modifiées uniquement dans la fonction `sig_handler`. Elles permettent de compter le nombre de signaux de chaque type qui ont été reçus.

Une lecture plus attentive du code ci-dessus révèle plusieurs points importants auxquels il faut être attentif lorsque l'on utilise les signaux.

Tout d'abord, lorsqu'un processus comprend une (ou plusieurs) fonction(s) de traitement de signaux, il y a plusieurs séquences d'instructions qui peuvent être exécutées par le processus. La première est la suite d'instructions du processus lui-même qui démarre à la fonction `main`. Dès qu'un signal est reçu, cette séquence d'instructions est interrompue pour exécuter la séquence d'instructions de la fonction de traitement du signal. Ce n'est que lorsque cette fonction se termine que la séquence principale peut reprendre son exécution à l'endroit où elle a été interrompue.

L'existence de deux ou plusieurs séquences d'instructions peut avoir des conséquences importantes sur le bon fonctionnement du programme et peut poser de nombreuses difficultés d'implémentation. En effet, une fonction de traitement de signal doit pouvoir être exécutée à n'importe quel moment. Elle peut donc démarrer à n'importe quel endroit de la séquence d'instructions du processus. Si le processus et une fonction de traitement de signal accèdent à la même variable, il y a un risque que celle-ci soit modifiée par la fonction de traitement du signal pendant qu'elle est utilisée dans le processus. Si l'on n'y prend garde, ces accès venant de différentes séquences d'instructions peuvent poser des problèmes similaires à ceux posés par l'utilisation de threads. Une routine de traitement de signal est cependant moins générale qu'un thread et les techniques utilisables dans les threads ne sont pas applicables aux fonctions de traitement des signaux. En effet, quand la fonction de traitement de signal démarre, il est impossible de bloquer son exécution sur un mutex pour revenir au processus principal. Celle-ci doit s'exécuter jusqu'à sa dernière instruction.

Lorsque l'on écrit une routine de traitement de signal, plusieurs précautions importantes doivent être prises. Tout d'abord, une fonction de traitement de signal doit manipuler les variables avec précautions. Comme elle est potentiellement exécutée depuis n'importe quel endroit du code, elle ne peut pas s'appuyer sur le stack. Elle ne peut utiliser que des variables globales pour influencer le processus principal. Comme ces variables peuvent être utilisées à la fois dans le processus et la routine de traitement de signal, il est important de les déclarer en utilisant le mot-clé `volatile`. Cela force le compilateur à recharger la valeur de la variable de la mémoire à chaque fois que celle-ci est utilisée. Mais cela ne suffit pas car il est possible que le processus exécute l'instruction de chargement de la valeur de la variable puis qu'un signal lui soit délivré, ce qui provoquera l'exécution de la fonction de traitement du signal. Lorsque celle-ci se terminera le processus poursuivra son exécution sans recharger la valeur de la variable potentiellement modifiée par la fonction de traitement du signal. Face à ce problème, il est préférable d'utiliser uniquement des variables de types `sig_atomic_t` dans les fonctions de traitement de signaux. Ce type permet de stocker un entier. Lorsque ce type est utilisé, le compilateur garantit que tous les accès à la

variable se feront de façon atomique sans accès concurrent possible entre le processus et la fonction de traitement des signaux.

L'utilisation de `sig_atomic_t` n'est pas la seule précaution à prendre lorsque l'on écrit une fonction de traitement des signaux. Il faut également faire attention aux fonctions de la librairie et aux appels systèmes que l'on utilise. Sachant qu'un signal peut être reçu à n'importe quel moment, il est possible qu'un processus reçoive un signal et exécute une fonction de traitement du signal pendant l'exécution de la fonction `fct` de la librairie standard. Si la fonction de traitement du signal utilise également la fonction `fct`, il y a un risque d'interférence entre l'exécution de ces deux fonctions. Ce sera le cas notamment si la fonction utilise un buffer statique ou modifie la variable `errno`. Dans ces cas, la fonction de traitement du signal risque de modifier une valeur ou une zone mémoire qui a déjà été modifiée par le processus principal et cela donnera un résultat incohérent. Pour éviter ces problèmes, il ne faut utiliser que des fonctions "réentrantes" à l'intérieur des fonctions de traitement des signaux. Des fonctions comme printf(3), scanf(3) ne sont pas réentrantes et ne doivent pas être utilisées dans une section de traitement des signaux. La (courte) liste des fonctions qui peuvent être utilisées sans risque est disponible dans la section 2.4 de l'Open Group Base Specification

Ces restrictions sur les instructions qui peuvent être utilisées dans une fonction de traitement des signaux ne sont pas les seules qui affectent l'utilisation des signaux. Ceux-ci souffrent d'autres limitations.

Pour bien les comprendre, il est utile d'analyser comment ceux-ci sont supportés par le noyau. Il y a deux stratégies possibles pour implémenter les signaux sous Unix. La première stratégie est de considérer qu'un signal est un message qui est envoyé depuis le noyau ou un processus à un autre processus. Pour traiter ces messages, le noyau contient une queue qui stocke tous les signaux destinés à un processus donné. Avec cette stratégie d'implémentation, l'appel système kill(2) génère un message et le place dans la queue associée au processus destination. Le noyau stocke pour chaque processus un tableau de pointeurs vers les fonctions de traitement de chacun des signaux. Ce tableau est modifié par l'appel système signal(2). Chaque fois que le noyau réactive un processus, il vérifie si la queue associée à ce processus contient un ou plusieurs messages concernant des signaux. Si un message est présent, le noyau appelle la fonction de traitement du signal correspondant. Lorsque la fonction se termine, l'exécution du processus reprend à l'instruction qui avait été interrompue.

La seconde stratégie est de représenter l'ensemble des signaux qu'un processus peut recevoir sous la forme de drapeaux binaires. En pratique, il y a un drapeau par signal. Avec cette stratégie d'implémentation, l'appel système kill(2) modifie le drapeau correspondant du processus destination du signal (sauf si ce signal est ignoré par le processus, dans ce cas le drapeau n'est pas modifié). L'appel système signal(2) modifie également le tableau contenant les fonctions de traitement des signaux associé au processus. Chaque fois que le noyau réactive un processus, que ce soit après un changement de contexte ou après l'exécution d'un appel système, il vérifie les drapeaux relatifs aux signaux du processus. Si un des drapeaux est vrai, le noyau appelle la fonction de traitement associée à ce signal.

La plupart des variantes de Unix, y compris Linux, utilisent la seconde stratégie d'implémentation pour les signaux. L'avantage principal de l'utilisation de drapeaux pour représenter les signaux reçus par un processus est qu'il suffit d'un bit par signal qui peut être reçu par le processus. La première stratégie nécessite de maintenir une queue par processus et la taille de cette queue varie en fonction du nombre de signaux reçus. Par contre, l'utilisation de drapeaux a un inconvénient majeur : il n'y a pas de garantie sur la délivrance des signaux. Lorsqu'un processus reçoit un signal, cela signifie qu'il y a au moins un signal de ce type qui a été envoyé au processus, mais il est très possible que plus d'un signal ont été envoyés au processus.

Pour illustrer ce problème, considérons le programme ci-dessous qui compte simplement le nombre de signaux `SIGUSR1` reçus.

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

volatile sig_atomic_t n_sigusr1=0;

static void sig_handler(int);

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

  if(signal(SIGUSR1,sig_handler)==SIG_ERR) {
```

```
    perror("signal");
    exit(EXIT_FAILURE);
  }
  int duree=60;
  while(duree>0) {
    printf("Exécution de sleep(%d)\n",duree);
    duree=sleep(duree);
  }
  printf("Fin du processus\n");
  printf("Reçu %d SIGUSR1\n",n_sigusr1);
  return(EXIT_SUCCESS);
}

static void sig_handler(int signum) {

  if(signum==SIGUSR1) {
    n_sigusr1++;
  }
}
```

Depuis un shell, il est possible d'envoyer plusieurs fois le signal `SIGUSR1` rapidement avec le script `/Fichiers/src/nkill.sh`. Ce script prend deux arguments : le nombre de signaux à envoyer et le processus destination.

```
#!/bin/bash
if [ $# -ne 2 ]
then
    echo "Usage: `basename $0` n pid"
    exit 1
fi
n=$1
pid=$2
for (( c=1; c<=$n; c++ ))
do
    kill -s SIGUSR1 $pid
done
```

La sortie ci-dessous présente une exécution de ce script avec le processus `/Fichiers/src/sigusrcount.c` en tâche de fond.

```
$ ./sigusrcount &
[1] 47845
$ Exécution de sleep(60)
$./nkill.sh 10 47845
Exécution de sleep(52)
$ ./nkill.sh 10 47845
Exécution de sleep(46)
$ ./nkill.sh 10 47845
Exécution de sleep(31)
$ Fin du processus
Reçu 3 SIGUSR1
```

Il y a plusieurs points intéressants à noter concernant l'exécution de ce programme. Tout d'abord, même si 30 signaux `SIGUSR1` ont été générés, seuls 3 de ces signaux ont effectivement étés reçus. Les signaux ne sont manifestement pas fiables sous Unix et cela peut s'expliquer de deux façons. Premièrement, les signaux sont implémentés sous la forme de bitmaps. La réception d'un signal modifie simplement la valeur d'un bit dans le bitmap du processus. En outre, durant l'exécution de la fonction qui traite le signal `SIGUSR1`, ce signal est bloqué par le système d'exploitation pour éviter qu'un nouveau signal n'arrive pendant que le premier est traité.

Un deuxième point important à relever est l'utilisation de sleep(3). Par défaut, cette fonction de la librairie permet d'attendre un nombre de secondes passé en argument. Si sleep(3) a mis le processus en attente pendant au moins le temps demandé, elle retourne `0` comme valeur de retour. Mais sleep(3) est un des exemples de fonctions ou appels systèmes qui sont dits *lents*. Un *appel système lent* est un appel système dont l'exécution peut être interrompue

par la réception d'un signal. C'est notamment le cas pour l'appel système read(2) [8]. Lorsqu'un signal survient pendant l'exécution d'un tel appel système, celui-ci est interrompu pour permettre l'exécution de la fonction de traitement du signal. Après l'exécution de cette fonction, l'appel système se termine [9] en retournant une erreur et met `errno` à `EINTR` de façon à indiquer que l'appel système a été interrompu. Le processus doit bien entendu traiter ces interruptions d'appels systèmes si il utilise des signaux.

Nous terminons cette section en analysant deux cas pratiques d'utilisation des signaux. Le premier est relatif aux signaux synchrones et nous développons une fonction de traitement du signal `SIGFPE` pour éviter qu'un programme ne s'arrête suite à une division par zéro. Ensuite, nous utilisons alarm(3posix) pour implémenter un temporisateur simple.

5.3.3 Traitement de signaux synchrones

Le programme ci-dessous prend en arguments en ligne de commande une séquence d'entiers et divise la valeur `1252` par chaque entier passé en argument. Il enregistre la fonction `sigfpe_handler` comme fonction de traitement du signal `SIGFPE`.

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

static void sigfpe_handler(int);

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

  int n=1252;
  void (*handler)(int);

  handler=signal(SIGFPE,sigfpe_handler);
  if(handler==SIG_ERR) {
    perror("signal");
    exit(EXIT_FAILURE);
  }

  for(int i=1;i<argc;i++) {
    char *endptr;
    printf("Traitement de argv[%d]=%s\n",i,argv[i]);
    fflush(stdout);
    long val=strtol(argv[i],&endptr,10);
    if(*endptr=='\0') {
      int resultat=n/(int) val;
      printf("%d/%d=%d\n",n,(int) val,resultat);
    }
    else {
      printf("Argument incorrect : %s\n",argv[i]);
    }
  }
  return(EXIT_SUCCESS);
}

static void sigfpe_handler(int signum) {
  char *msg="Signal SIGFPE reçu\n";
  write(STDOUT_FILENO,msg,strlen(msg));

}
```

8. Les autres appels systèmes lents sont open(2), write(2), sendto(2), recvfrom(2), sendmsg(2), recvmsg(2), wait(2) ioctl(2).
9. L'appel système sigaction(2) permet notamment de spécifier pour chaque signal si un appel système interrompu par ce signal doit être automatiquement redémarré lorsque le signal survient ou non.

Lors de son exécution, ce programme affiche la sortie ci-dessous :

```
$ ./sigfpe2 1 2 aa 0 9
Traitement de argv[1]=1
1252/1=1252
Traitement de argv[2]=2
1252/2=626
Traitement de argv[3]=aa
Argument incorrect : aa
Traitement de argv[4]=0
Signal SIGFPE reçu
Signal SIGFPE reçu
...
```

La fonction `sigfpe_handler` traite bien le signal `SIGPFE` reçu, mais après son exécution, elle tente de recommencer l'exécution de la ligne `int resultat=n/(int) val;`, ce qui provoque à nouveau un signal `SIGPFE`. Pour éviter ce problème, il faut permettre à la fonction de traitement du signal de modifier la séquence d'exécution de la fonction `main` après la réception du signal. Une solution pourrait être d'utiliser l'instruction `goto` comme suit :

```
  if(*endptr=='\0') {
    int resultat=n/(int) val;
    printf("%d/%d=%d\n",n,(int) val,resultat);
    goto fin:
erreur:
    printf("%d/%d=NaN\n",n,(int) val);
fin:
  }
  else {
    printf("Argument incorrect : %s\n",argv[i]);
   }
// ...

static void sigfpe_handler(int signum) {
 goto erreur:
}
```

En C, ce genre de construction n'est pas possible car l'étiquette d'un `goto` doit nécessairement se trouver dans la même fonction que l'invocation de `goto`. La seule possibilité pour faire ce genre de saut entre fonctions en C, est d'utiliser les fonctions setjmp(3) et longjmp(3). Dans un programme normal, ces fonctions sont à éviter encore plus que les `goto` car elles rendent le code très difficile à lire.

```
#include <setjmp.h>

int setjmp(jmp_buf env);

void longjmp(jmp_buf env, int val);
```

La fonction setjmp(3) est équivalente à la déclaration d'une étiquette. Elle prend comme argument un `jmp_buf`. Cette structure de données, définie dans setjmp.h permet de sauvegarder l'environnement d'exécution, c'est-à-dire les valeurs des registres y compris `%eip` et `%esp` au moment où elle est exécutée. Lorsque setjmp(3) est exécutée dans le flot normal des instructions du programme, elle retourne la valeur `0`. La fonction longjmp(3) prend deux arguments. Le premier est une structure de type `jmp_buf` et le second un entier. Le `jmp_buf` est l'environnement d'exécution qu'il faut restaurer lors de l'exécution de longjmp(3) et le second argument la valeur de retour que doit avoir la fonction setjmp(3) correspondante après l'exécution de longjmp(3).

Le programme ci-dessous illustre l'utilisation de setjmp(3) et longjmp(3).

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

jmp_buf label;
```

```
void f() {

  printf("Début fonction f\n");
  if(setjmp(label)==0) {
    printf("Exécution normale\n");
  }
  else {
    printf("Exécution après longjmp\n");
  }
}

void g() {
  printf("Début fonction g\n");
  longjmp(label,1);
  printf("Ne sera jamais affiché\n");
}

int main (int argc, char *argv[])  {
  f();
  g();
  return(EXIT_SUCCESS);
}
```

Le programme débute en exécutant la fonction `f`. Dans cette exécution, la fonction setjmp(3) retourne la valeur `0`. Ensuite, la fonction `main` appelle la fonction `g` qui elle exécute `longjmp(label,1)`. Cela provoque un retour à la fonction `f` à l'endroit de l'exécution de `setjmp(label)` qui cette fois-ci va retourner la valeur `1`. Lors de son exécution, le programme ci-dessus affiche :

```
Début fonction f
Exécution normale
Début fonction g
Exécution après longjmp
```

Avec les fonctions setjmp(3) et longjmp(3), il est presque possible d'implémenter le traitement attendu pour le signal `SIGFPE`. Il reste un problème à résoudre. Lorsque la routine de traitement du signal `SIGFPE` s'exécute, ce signal est bloqué par le système d'exploitation jusqu'à ce que cette fonction se termine. Si elle effectue un longjmp(3), elle ne se terminera jamais et le signal continuera à être bloqué. Pour éviter ce problème, il faut utiliser les fonctions sigsetjmp(3) et siglongjmp(3). Ces fonctions sauvegardent dans une structure de données `sigjmp_buf` non seulement l'environnement d'exécution mais aussi la liste des signaux qui sont actuellement bloqués. La fonction sigsetjmp(3) prend un argument supplémentaire : un entier qui doit être non nul si l'on veut effectivement sauvegarder la liste des signaux bloqués. Lorsque siglongjmp(3) s'exécute, l'environnement et la liste des signaux bloqués sont restaurés.

Le programme ci-dessous présente l'utilisation de sigsetjmp(3) et siglongjmp(3).

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <setjmp.h>

sigjmp_buf buf;

static void sigfpe_handler(int);

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

  int n=1252;
  if(signal(SIGFPE,sigfpe_handler)==SIG_ERR) {
    perror("signal");
    exit(EXIT_FAILURE);
  }
```

```
  for(int i=1;i<argc;i++) {
    char *endptr;
    int r;
    printf("Traitement de argv[%d]=%s\n",i,argv[i]);
    long val=strtol(argv[i],&endptr,10);
    if(*endptr=='\0') {
      r=sigsetjmp(buf,1);
      if(r==0) {
        int resultat=n/(int) val;
        printf("%d/%d=%d\n",n,(int) val,resultat);
      }
      else {
        printf("%d/%d=NaN\n",n,(int) val);
      }
    }
    else {
      printf("Argument incorrect : %s\n",argv[i]);
    }
  }
  return(EXIT_SUCCESS);
}

static void sigfpe_handler(int signum) {
  // ignorer la donnée et passer à la suivante
  siglongjmp(buf,1);
}
```

Lors de son exécution, il affiche la sortie standard suivante.

```
./sigfpe3 1 2 3 0 a 0 3
Traitement de argv[1]=1
1252/1=1252
Traitement de argv[2]=2
1252/2=626
Traitement de argv[3]=3
1252/3=417
Traitement de argv[4]=0
1252/0=NaN
Traitement de argv[5]=a
Argument incorrect : a
Traitement de argv[6]=0
1252/0=NaN
Traitement de argv[7]=3
1252/3=417
```

5.3.4 Temporisateurs

Parfois il est nécessaire dans un programme de limiter le temps d'attente pour réaliser une opération. Un exemple simple est lorsqu'un programme attend l'entrée d'un paramètre via l'entrée standard mais peut remplacer ce paramètre par une valeur par défaut si celui-ci n'est pas entré endéans quelques secondes. Lorsqu'un programme attend une information via l'entrée standard, il exécute l'appel système read(2) directement ou via des fonctions de la librairie comme fgets(3) ou getchar(3). Par défaut, celui-ci est bloquant, cela signifie qu'il ne se terminera que lorsqu'une donnée aura été lue. Si read(2) est utilisé seul, il n'est pas possible de borner le temps d'attente du programme et d'interrompre l'appel à read(2) après quelques secondes. Pour obtenir ce résultat, une possibilité est d'utiliser un signal. En effet, read(2) est un appel système lent qui peut être interrompu par la réception d'un signal. Il y a plusieurs façons de demander qu'un signal soit généré après un certain temps. Le plus général est setitimer(2). Cet appel système permet de générer un signal `SIGALRM` après un certain temps ou périodiquement. L'appel système alarm(3posix) est plus ancien mais plus simple à utiliser que setitimer(2). Nous l'utilisons afin d'illustrer comment un signal peut permettre de limiter la durée d'attente d'un appel système.

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <stdbool.h>

static void sig_handler(int);

int main (int argc, char *argv[])  {
  char c;
  printf("Tapez return en moins de 5 secondes !\n");
  fflush(stdout);
  if(signal(SIGALRM,sig_handler)==SIG_ERR) {
    perror("signal");
    exit(EXIT_FAILURE);
  }
  // sigalrm interrompt les appels système
  if(siginterrupt(SIGALRM,true)<0) {
    perror("siginterrupt");
    exit(EXIT_FAILURE);
  }
  alarm(5);
  int r=read(STDIN_FILENO,&c,1);
  if((r==1)&&(c=='\n')) {
    alarm(0); // reset timer
    printf("Gagné\n");
    exit(EXIT_SUCCESS);
  }
  printf("Perdu !\n");
  exit(EXIT_FAILURE);
}

static void sig_handler(int signum) {
  // rien à faire, read sera interrompu
}
```

Ce programme utilise alarm(3posix) pour limiter la durée d'un appel système read(2). Pour ce faire, il enregistre d'abord une fonction pour traiter le signal `SIGALRM`. Cette fonction est vide dans l'exemple, son exécution permet juste d'interrompre l'appel système read(2). Par défaut, lorsqu'un signal survient durant l'exécution d'un appel système, celui-ci est automatiquement redémarré par le système d'exploitation pour éviter à l'application de devoir traiter tous les cas possibles d'interruption d'appels système. La fonction siginterrupt(3) permet de modifier ce comportement par défaut et nous l'utilisons pour forcer l'interruption d'appels systèmes lorsque le signal `SIGALRM` est reçu. L'appel à `alarm(0)` permet de désactiver l'alarme qui était en cours.

Lors de son exécution, ce programme affiche la sortie suivante.

```
Tapez return en moins de 5 secondes !
Perdu !
```

En essayant le programme ci-dessus, on pourrait conclure qu'il fonctionne parfaitement. Il a cependant un petit défaut qui peut s'avérer genant si par exemple on utilise la même logique pour écrire une fonction `read_time` qui se comporte comme read(2) sauf que son dernier argument est un délai maximal. Sur un système fort chargé, il est possible qu'après l'exécution de `alarm(5)` le processus soit mis en attente par le système d'exploitation qui exécute d'autres processus. Lorsque l'alarme expire, la fonction de traitement de `SIGALRM` est exécutée puis seulement l'appel à read(2) s'effectue. Celui-ci étant bloquant, le processus restera bloqué jusqu'à ce que les données arrivent ce qui n'est pas le comportement attendu.

Pour éviter ce problème, il faut empêcher l'exécution de read(2) si le signal `SIGALRM` a déjà été reçu. Cela peut se réaliser en utilisant sigsetjmp(3) pour définir une étiquette avant l'exécution du bloc contenant l'appel à alarm(3posix) et l'appel à read(2). Si le signal n'est pas reçu, l'appel à read(2) s'effectue normalement. Si par contre le signal `SIGALRM` est reçu entre l'appel à alarm(3posix) et l'appel à read(2), alors l'exécution de siglongjmp(3) dans `sig_handler` empêchera l'exécution de l'appel système read(2) ce qui est bien le comportement

attendu.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <stdbool.h>
#include <setjmp.h>

sigjmp_buf env;

static void sig_handler(int);

int main (int argc, char *argv[])  {
  char c;
  printf("Tapez return en moins de 5 secondes !\n");
  fflush(stdout);
  if(signal(SIGALRM,sig_handler)==SIG_ERR) {
    perror("signal");
    exit(EXIT_FAILURE);
  }
  // sigalrm interrompt les appels système
  if(siginterrupt(SIGALRM,true)<0) {
    perror("siginterrupt");
    exit(EXIT_FAILURE);
  }
  int r=0;
  if(sigsetjmp(env,1)==0) {
    // sig_handler n'a pas encore été appelé
    alarm(5);
    r=read(STDIN_FILENO,&c,1);
  }
  else {
    // sig_handler a déjà été exécuté
    // le délai a déjà expiré, inutile de faire read
  }
  alarm(0); // arrêt du timer
  if((r==1)&&(c=='\n')) {
    printf("Gagné\n");
    exit(EXIT_SUCCESS);
  }
  else {
    printf("Perdu !\n");
    exit(EXIT_FAILURE);
  }
}

static void sig_handler(int signum) {
  siglongjmp(env,1);
}
```

L'appel système alarm(3posix) s'appuie sur setitimer(2), mais les deux types d'alarmes ne doivent pas être combinés. Il en va de même pour sleep(3) qui peut être implémenté en utilisant `SIGALRM`. Linux contient d'autres appels systèmes et fonctions pour gérer différents types de temporisateurs. Ceux-ci sont décrits de façon détaillée dans [Kerrisk2010].

Note : Signaux, threads, fork(2) et execve(2)

Le noyau du système d'exploitation maintient pour chaque processus une structure de données contenant la liste des signaux qui sont ignorés, ont été reçus et les pointeurs vers les fonctions de traitement pour chaque signal. Cette structure de données est associée à chaque processus. La création de threads ne modifie pas cette structure de données et lorsqu'un signal est délivré à un processus utilisant des threads, c'est généralement le thread principal qui recevra et devra traiter le signal. Lors de l'exécution de fork(2), la structure de données relative aux signaux

du processus père est copiée dans le processus fils. Après fork(2), les deux processus peuvent évoluer séparément et le fils peut par exemple modifier la façon dont il traite un signal sans que cela n'affecte le processus père. Lors de l'exécution de execve(2), la structure de données relative aux signaux est réinitialisée avec les traitements par défaut pour chacun des signaux.

5.4 Sémaphores nommés

Nous avons présenté les sémaphores lors de l'étude du fonctionnement des threads et les mécanismes qui permettent de les coordonner. Chaque sémaphore utilise une structure de données qui est dans une mémoire accessible aux différents threads/processus qui doivent se coordonner. Lorsque des sémaphores sont utilisés pour coordonner des threads, cette structure de données est soit stockée dans la zone réservée aux variables globales, soit sur le tas. Lorsque des processus doivent se coordonner, il ne partagent pas nécessairement [10] de la mémoire. Dans ce cas, la fonction sem_init(3) ne peut pas être utilisée pour créer de sémaphore. Par contre, il est possible d'utiliser des sémaphores "nommés" (named semaphores). Ces sémaphores utilisent une zone de mémoire qui est gérée par le noyau et qui peut être utilisée par plusieurs processus. Un *sémaphore nommé* est identifié par un *nom*. Sous Linux, ce nom correspond à un fichier sur un système de fichiers et tout processus qui connait le nom d'un sémaphore et possède les permissions l'autorisant à y accéder peut l'utiliser. Trois appels systèmes sont utilisés pour créer, utiliser et supprimer un sémaphore nommé. Une présentation générale des sémaphores est disponible dans la page de manuel sem_overview(7).

```
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

sem_open(3) permet d'ouvrir ou de créer un *sémaphore nommé*. Tout comme l'appel système open(2), il existe deux variantes de sem_open(3). La première prend 2 arguments et permet d'utiliser un *sémaphore nommé* déjà existant. La second prend quatre arguments et permet de créer un nouveau *sémaphore nommé* et l'initialise à la valeur du quatrième argument. La fonction sem_open(3) s'utilise de la même façon que l'appel système open(2) sauf qu'elle retourne un pointeur vers une structure de type `sem_t` et qu'en cas d'erreur elle retourne `SEM_FAILED` et utilise `errno` pour fournir une information sur le type d'erreur. La fonction sem_close(3) permet à un processus d'arrêter d'utiliser un *sémaphore nommé*. Enfin, la fonction sem_unlink(3) permet de supprimer un sémaphore qui avait été créé avec sem_open(3). Tout comme il est possible d'utiliser l'appel système unlink(2) sur un fichier ouvert par un processus, il est possible d'appeler sem_unlink(3) avec comme argument un nom de sémaphore utilisé actuellement par un processus. Dans ce cas, le sémaphore sera complètement libéré lorsque le dernier processus qui l'utilise aura effectué sem_close(3). Les fonctions sem_post(3) et sem_wait(3) s'utilisent de la même façon qu'avec les sémaphores non-nommés que nous avons utilisé précédemment.

A titre d'exemple, considérons un exemple simple d'utilisation de sémaphores nommés dans lequel un processus doit attendre la fin de l'exécution d'une fonction dans un autre processus pour pouvoir s'exécuter. Avec les threads, nous avions résolu ce problème en initialisant un sémaphore à `0` dans le premier thread alors que le second démarrait par un sem_wait(3). Le première exécute sem_post(3) dès qu'il a fini l'exécution de sa fonction critique.

Le programme ci-dessous illustre le processus qui s'exécute en premier.

```
sem_t *semaphore;

void before() {
  // do something
  for(int j=0;j<1000000;j++) {
  }
  printf("before done, pid=%d\n",(int) getpid());
```

10. Nous verrons dans le prochain chapitre comment plusieurs processus peuvent partager une même zone mémoire.

```
  sem_post(semaphore);
}


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

  int err;

  semaphore=sem_open("lsinf1252",O_CREAT,S_IRUSR | S_IWUSR,0);
  if(semaphore==SEM_FAILED) {
    error(-1,"sem_open");
  }
  sleep(20);
  before();
  err=sem_close(semaphore);
  if(err!=0) {
      error(err,"sem_close");
  }
  return(EXIT_SUCCESS);
}
```

Ce processus commence par utiliser sem_open(3) pour créer un sémaphore qui porte le nom `lsinf1252` et est initialisé à zéro puis se met en veille pendant vingt secondes. Ensuite il exécute la fonction `before` qui se termine par l'exécution de `sem_post(semaphore)`. Cet appel a pour résultat de libérer le second processus dont le code est présenté ci-dessous :

```
sem_t *semaphore;

void after() {
  sem_wait(semaphore);
  // do something
  for(int j=0;j<1000000;j++) {
  }
  printf("after done, pid=%d\n",(int) getpid());
}

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

  // semaphore a été créé par before
  semaphore=sem_open("lsinf1252",0);
  if(semaphore==SEM_FAILED) {
      error(-1,"sem_open");
  }
  after();

  err=sem_close(semaphore);
  if(err!=0) {
      error(err,"sem_close");
  }
  err=sem_unlink("lsinf1252");
  if(err!=0) {
      error(err,"sem_unlink");
  }
  return(EXIT_SUCCESS);
}
```

Ce processus utilise le sémaphore qui a été créé par un autre processus pour se coordonner. La fonction `after` démarre par l'exécution de sem_wait(3) qui permet d'attendre que l'autre processus ait terminé l'exécution de la fonction `before`. Les sémaphores nommés peuvent être créés et supprimés comme des fichiers. Il est donc normal que le sémaphore soit créé par le premier processus et supprimé par le second. Sous Linux, les sémaphores nommés sont créés comme des fichiers dans le système de fichiers virtuel `/dev/shm` :

```
$ ls -l /dev/shm/
-rw------- 1 obo         stafinfo       32 Apr 10 15:37 sem.lsinf1252
```

Les permissions du fichier virtuel représentent les permissions associées au sémaphore. La sortie ci-dessous présente une exécution des deux processus présentés plus haut.

```
$ ./process-sem-before &
[1] 5222
$ ./process-sem-after
before done, pid=5222
after done, pid=5223
[1]+  Done                    ./process-sem-before
```

Il est important de noter que les sémaphores nommés sont une ressource généralement limitée. Lorsqu'il a été créé, un sémaphore nommé utilise des ressources du système jusqu'à ce qu'il soit explicitement supprimé avec sem_unlink(3). Il est très important de toujours bien effacer les sémaphores nommés dès qu'ils ne sont plus nécessaires. Sans cela, l'espace réservé pour ces sémaphores risque d'être rempli et d'empêcher la création de nouveaux sémaphores par d'autres processus.

5.5 Partage de fichiers

Les fichiers sont l'un des principaux moyens de communication entre processus. L'avantage majeur des fichiers est leur persistence. Les données sauvegardées dans un fichier persistent sur le système de fichiers après la fin du processus qui les a écrites. L'inconvénient majeur de l'utilisation de fichiers par rapport à d'autres techniques de communication entre processus est la relative lenteur des dispositifs de stockage en comparaison avec les accès à la mémoire. Face à cette lenteur des dispositifs de stockage, la majorité des systèmes d'exploitation utilisent des buffers qui servent de tampons entre les processus et les dispositifs de stockage. Lorsqu'un processus écrit une donnée sur un dispositif de stockage, celle-ci est d'abord écrite dans un buffer géré par le système d'exploitation et le processus peut poursuivre son exécution sans devoir attendre l'exécution compléte de l'écriture sur le dispositif de stockage. La taille de ces buffers varie généralement dynamiquement en fonction de la charge du système. Les données peuvent y rester entre quelques fractions de seconde et quelques dizaines de secondes. Un processus peut contrôler l'utilisation de ce buffer en utilisant l'appel système fsync(2). Celui-ci permet de forcer l'écriture sur le dispositif de stockage des données du fichier identifié par le descripteur de fichiers passé en argument. L'appel système sync(2) force quant à lui l'écriture de toutes les données actuellement stockées dans les buffers du noyau sur les dispositifs de stockage. Cet appel système est notamment utilisé par un processus système qui l'exécute toutes les trente secondes afin d'éviter que des données ne restent trop longtemps dans les buffers du noyau.

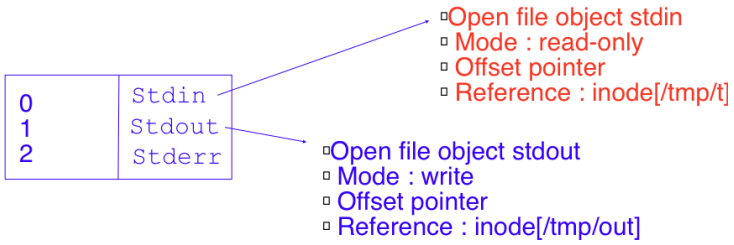
L'utilisation d'un même fichier par plusieurs processus est une des plus anciennes techniques de communication entre processus. Pour comprendre son fonctionnement, il est utile d'analyser les structures de données qui sont maintenues par le noyau du système d'exploitation pour chaque fichier et chaque processus. Comme nous l'avons présenté dans le chapitre précédent, le système de fichiers utilise des inodes pour stocker les méta-données et la liste des blocs de chaque fichier. Lorsqu'un processus ouvre un fichier, le noyau du système d'exploitation lui associe le premier descripteur de fichier libre dans la table des descripteurs de fichiers du processus. Ce descripteur de fichier pointe alors vers une structure maintenue par le noyau qui est souvent appelée un *open file object*. Un *open file object* contient toutes les informations qui sont nécessaires au noyau pour pouvoir effectuer les opérations de manipulation d'un fichier ouvert par un processus. Parmi celles-ci, on trouve notamment :

- le mode dans lequel le fichier a été ouvert (lecture seule, écriture, lecture/écriture). Ce mode est initialisé à l'ouverture du fichier. Le noyau vérifie le mode lors de l'exécution des appels systèmes read(2) et write(2) mais pas les permissions du fichier sur le système de fichiers.
- l'offset pointer qui est la tête de lecture/écriture dans le fichier
- une référence vers le fichier sur le système de fichiers. Dans un système de fichiers Unix, il s'agit généralement du numéro de l'*inode* du fichier ou d'un pointeur vers une structure contenant cet *inode* et des informations comme le dispositif de stockage sur lequel il est stocké.

A titre d'exemple, considérons l'exécution de la commande suivante depuis le shell :

```
$ cat < /tmp/t > /tmp/out
```

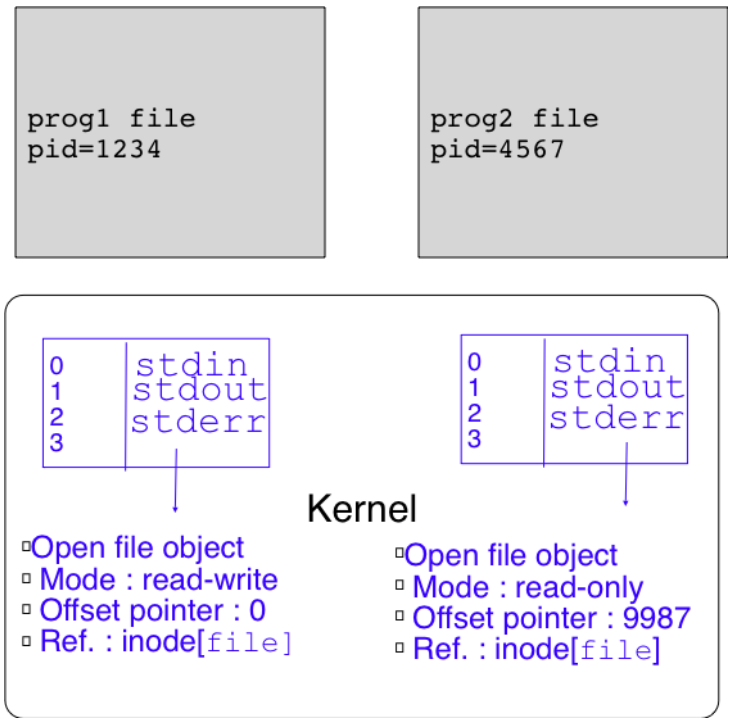
Lors de son exécution, deux open file objects sont créés dans le noyau. Le premier est relatif au fichier `/tmp/t` qui est associé au descripteur `stdin`. Le second est lié au fichier `/tmp/out` et est associé au descripteur `stdout`. Ces open-file objects sont représentés graphiquement dans la figure ci-dessous.



Les open file objects sont également utilisés lorsque plusieurs processus ouvrent le même fichier. Considérons l'exécutions simultanée des deux commandes suivantes :

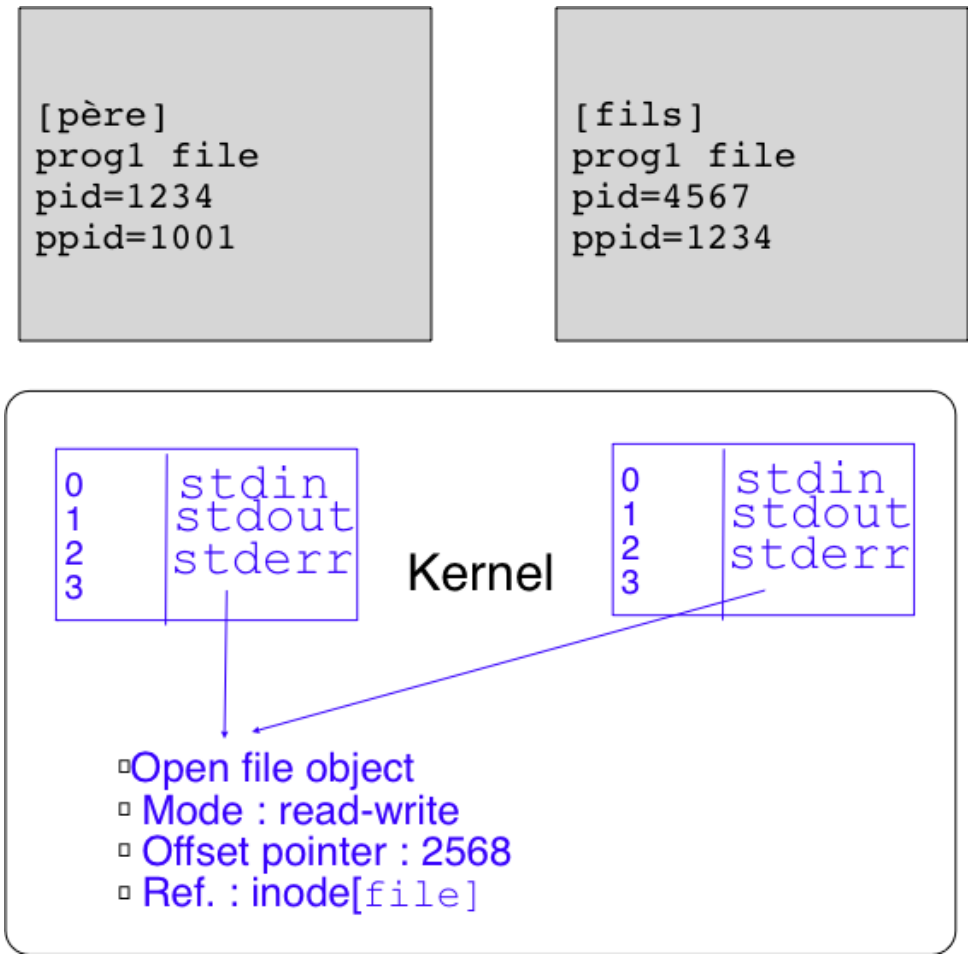
```
$ prog1 file &
$ prog2 file &
```

Lors de l'exécution de ces deux processus, le noyau va attribuer un descripteur de fichiers à chacun d'eux. Si `file` est le premier fichier ouvert par chaque processus, il sera associé au descripteur `3`. Le noyau créera un open-file object pour le fichier `file` utilisé par le processus `prog1` et un autre open-file object pour le fichier `file` utilisé par le processus `prog2`. Ces deux open-file objects référencient le même inode et donc le même fichier, mais ils peuvent avoir des modes et des offset pointers qui sont totalement indépendants. Tous les accès faits par le processus `prog2` sont complètement indépendants des accès faits par le processus `prog1`. Cette utilisation d'un même fichier par deux processus distincts est représentée graphiquement dans la figure ci-dessous.



Sous Unix et Linux, il est important d'analyser également ce qu'il se passe lors de la création d'un processus en utilisant l'appel système fork(2). Imaginons que le processus `prog1` discuté ci-dessous effectue fork(2) après avoir ouvert le fichier `file`. Dans ce cas, le noyau du système d'exploitation va dupliquer la table des descripteurs de fichiers du processus père pour créer celle du processus fils. Le processus père et le processus fils ont donc chacun une table des descripteurs de fichiers qui leur est propre. Cela permet, comme nous l'avons vu lorsque nous avons présenté les pipes, que le fils ferme un de ses descripteurs de fichiers sans que cela n'ait d'impact sur l'utilisation de ce descripteur de fichier par le processus père. Par contre, l'exécution de l'appel système fork(2) ne copie pas les open-file objects. Après exécution de fork(2) le descripteur de fichiers `3` dans le processus père

pointe vers l'open-file object associé au fichier `file` et le même descripteur dans le processus fils pointe vers le même open-file object. Cette situation est représentée schématiquement dans la figure ci-dessous.



Cette utilisation d'un même open-file object par le processus père et le processus fils est une particularité importante de Unix. Elle permet aux deux processus d'écrire des données séquentiellement dans un fichier qui avait été initialement ouvert par le processus père, mais pose régulièrement des problèmes lors de la manipulation de fichiers.

Lorsqu'un fichier est utilisé par plusieurs processus simultanément, il est nécessaire de coordonner les activités de ces processus pour éviter que le fichier ne devienne corrompu. Outres les appels systèmes classiques open(2), read(2), write(2) et close(2), Unix offre plusieurs appels systèmes qui sont utiles lorsque plusieurs processus accèdent au même fichier.

Considérons d'abord un processus père et un processus fils qui doivent lire des données à des endroits particuliers dans un fichier. Pour cela, il est naturel d'utiliser l'appel système lseek(2) pour déplacer l'offset pointer et d'ensuite réaliser la lecture avec read(2). Malheureusement lorsqu'un père et un ou plusieurs fils [11] utilisent ces appels systèmes, il est possible qu'un appel à lseek(2) fait par le fils soit immédiatement suivi d'un appel à lseek(2) fait par le père avant que le fils ne puisse exécuter l'appel système read(2). Dans ce cas, le processus fils ne lira pas les données qu'il souhaitait lire dans le fichier. Les appels systèmes pread(2) et pwrite(2) permettent d'éviter ce problème. Ils complètent les appels systèmes read(2) et write(2) en prenant comme argument l'offset auquel la lecture ou l'écriture demandée doit être effectuée. pread(2) et pwrite(2) garantissent que les opérations d'écriture et de lecture qui sont effectuées avec ces appels systèmes seront atomiques.

```
#include <unistd.h>
ssize_t pread(int fd, void *buf, size_t count, off_t offset);
```

11. Même si il n'a pas été mentionné lors de l'utilisation de threads, ce problème se pose également lorsque plusieurs threads accèdent directement aux données dans un même fichier.

```
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

Ce n'est pas le seul problème qui se pose lorsque plusieurs processus manipulent un même fichier. Considérons un logiciel de base de données qui comprend des processus qui lisent dans des fichiers qui constituent la base de données et d'autres qui modifient le contenu de ces fichiers. Ces opérations d'écritures et de lectures dans des fichiers partagés risquent de provoquer des problèmes d'accès concurrent similaires aux problèmes que nous avons dû traiter lorsque plusieurs threads se partagent une même mémoire. Pour réguler ces accès à des fichiers, Unix et Linux supportent des verrous (locks en anglais) que l'on peut associer à des fichiers. A première vue, un *lock* peut être comparé à un *mutex*. Un *lock* permet à un processus d'obtenir l'accès exclusif à un fichier ou une partie de fichier tout comme un *mutex* est utilisé pour réguler les accès à une variable. En théorie, il existe deux techniques d'utilisation de locks qui peuvent être utilisées sur un système Unix :

— *mandatory locking*. Dans ce cas, les processus placent des locks sur certains fichiers ou zones de fichiers et le système d'exploitation vérifie qu'aucun accès fait aux fichiers avec les appels systèmes standards ne viole ces locks.
— *advisory locking*. Dans ce cas, les processus doivent vérifier eux-mêmes que les accès qu'ils effectuent ne violent pas les locks qui ont été associés aux différents fichiers.

Certains systèmes Unix supportent les deux stratégies de locking, mais la plupart ne supportent que l'*advisory locking*. L'*advisory locking* est la stratégie la plus facile à implémenter dans le système d'exploitation. C'est aussi celle qui donne les meilleures performances. Nous limitons notre description à l'*advisory locking*. Le *mandatory locking* nécessite un support spécifique du système de fichiers qui sort du cadre de ce cours.

Deux appels systèmes sont utilisés pour manipuler les locks qui peuvent être associés aux fichiers : flock(2) et fcntl(2). flock(2) est la solution la plus simple. Cet appel système permet d'associer un verrou à un fichier complet.

```
#include <sys/file.h>
int flock(int fd, int operation);
```

Il prend comme argument un descripteur de fichier et une opération. Deux types de locks sont supportés. Un lock est dit partagé (shared lock, `operation==LOCK_SH`) lorsque plusieurs processus peuvent posséder un même lock vers un fichier. Un lock est dit exclusif (exclusive lock, `operation==LOCK_EX`) lorsqu'un seul processus peut posséder un lock vers un fichier à un moment donné. Il faut noter que les locks sont associés aux fichiers (et donc indirectement aux inodes) et non aux descripteurs de fichiers. Pour retirer un lock associé à un fichier, il faut utiliser `LOCK_UN` comme second argument à l'appel flock(2).

L'appel système fcntl(2) et la fonction lockf(3) sont nettement plus flexibles. Ils permettent de placer des locks sur une partie d'un fichier. lockf(3) prend trois arguments : un descripteur de fichiers, une commande et un entier qui indique la longueur de la section du fichier à associer au lock.

```
#include <unistd.h>
int lockf(int fd, int cmd, off_t len);
```

lockf(3) supporte plusieurs commandes qui sont chacune spécifiées par une constante définie dans unistd.h. La commande `F_LOCK` permet de placer un lock exclusif sur une section du fichier dont le descripteur est le premier argument. Le troisième argument est la longueur de la section sur laquelle le lock doit être appliqué. Par convention, la section s'étend de la position actuelle de l'offset pointer (`pos`) jusqu'à `pos+len-1` si `len` est positif et va de `pos-len` à `pos-1` si `len` est négatif. Si `len` est nul, alors la section concernée par le lock va de `pos` à l'infini (c'est-à-dire jusqu'à la fin du fichier, même si celle-ci change après l'application du lock).

L'appel à lockf(3) bloque si un autre processus a déjà un lock sur une partie du fichier qui comprend celle pour laquelle le lock est demandé. La commande `F_TLOCK` est équivalente à `F_LOCK` avec comme différence qu'elle ne bloque pas si un lock existe déjà sur le fichier mais retourne une erreur. `F_TEST` permet de tester la présence d'un lock sans tenter d'acquérir ce lock contrairement à `F_TLOCK`. Enfin, la commande `F_ULOCK` permet de retirer un lock placé précédemment.

En pratique, lockf(3) doit s'utiliser de la même façon qu'un mutex. C'est-à-dire qu'un processus qui souhaite écrire de façon exclusive dans un fichier doit d'abord obtenir via lockf(3) un lock sur la partie du fichier dans laquelle il souhaite écrire. Lorsque l'appel à lockf(3) réussit, il peut effectuer l'écriture via write(2) et ensuite libérer le lock en utilisant lockf(3). Tout comme avec les mutex, si un processus n'utilise pas lockf(3) avant d'écrire ou de lire, cela causera des problèmes.

L'appel système fcntl(2) est un appel "fourre-tout" qui regroupe de nombreuses opérations qu'un processus peut vouloir faire sur un fichier. L'application de locks est une de ces opérations, mais la page de manuel en détaille de nombreuses autres. Lorsqu'il est utilisé pour manipuler des locks, l'appel système fcntl(2) utilise trois arguments :

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

int fcntl(int fd, int cmd, struct flock* );
```

Le premier argument est le descripteur de fichiers sur lequel le lock doit être appliqué. Le second est la commande. Tout comme lockf(3), fcntl(2) supporte différentes commandes qui sont spécifiées dans la page de manuel. Le troisième argument est un pointeur vers une structure `flock` qui doit contenir au minimum les champs suivants :

```
struct flock {
   ...
   short l_type;    /* Type of lock: F_RDLCK,
                       F_WRLCK, F_UNLCK */
   short l_whence;  /* How to interpret l_start:
                       SEEK_SET, SEEK_CUR, SEEK_END */
   off_t l_start;   /* Starting offset for lock */
   off_t l_len;     /* Number of bytes to lock */
   pid_t l_pid;     /* PID of process blocking our lock
                       (F_GETLK only) */
   ...
   };
```

Cette structure permet de spécifier plus finement qu'avec la fonction lockf(3) la section du fichier sur laquelle le lock doit être placé. L'utilisation de locks force le noyau à maintenir des structures de données supplémentaires pour stocker ces locks et les processus qui peuvent être en attente sur chacun de ces locks. Conceptuellement, cette structure de données est associée à chaque fichier comme représenté dans la figure ci-dessous.

Comme les locks sont associés à des fichiers, le noyau doit maintenir pour chaque fichier ouvert une liste de locks. Celle-ci peut être implémentée sous la forme d'une liste chaînée comme représenté ci-dessus ou sous la forme d'une autre structure de données. Le point important est que le noyau doit mémoriser pour chaque fichier utilisant des locks quelles sont les sections du fichier qui sont concernées par chaque lock, quel est le type de lock, quel est le processus qui possède le lock (pour autoriser uniquement ce processus à le retirer) et enfin une liste ou une queue des processus qui sont en attente sur ce lock.

Sous Linux, le système de fichiers virtuel `/proc` fournit une interface permettant de visualiser l'état des locks. Il suffit pour cela de lire le contenu du fichier `/proc/locks`.

```
cat /proc/locks
1: POSIX  ADVISORY  WRITE 12367 00:17:20628657 0 0
2: POSIX  ADVISORY  WRITE 12367 00:17:7996086 0 0
3: POSIX  ADVISORY  WRITE 12367 00:17:24084665 0 0
4: POSIX  ADVISORY  WRITE 12367 00:17:12634137 0 0
5: POSIX  ADVISORY  WRITE 30677 00:17:14534587 1073741824 1073742335
6: POSIX  ADVISORY  READ  30677 00:17:25756362 128 128
7: POSIX  ADVISORY  READ  30677 00:17:25756359 1073741826 1073742335
8: POSIX  ADVISORY  READ  30677 00:17:25756319 128 128
9: POSIX  ADVISORY  READ  30677 00:17:25756372 1073741826 1073742335
10: POSIX  ADVISORY  WRITE 30677 00:17:25757269 1073741824 1073742335
11: POSIX  ADVISORY  WRITE 30677 00:17:25756354 0 EOF
12: FLOCK  ADVISORY  WRITE 22677 00:18:49578023 0 EOF
13: POSIX  ADVISORY  WRITE 3023 08:01:652873 0 EOF
14: POSIX  ADVISORY  WRITE 3014 08:01:652855 0 EOF
15: POSIX  ADVISORY  WRITE 2994 08:01:652854 0 EOF
16: FLOCK  ADVISORY  WRITE 2967 08:01:798437 0 EOF
17: FLOCK  ADVISORY  WRITE 2967 08:01:797391 0 EOF
18: FLOCK  ADVISORY  WRITE 1278 08:01:652815 0 EOF
```

Dans ce fichier, la première colonne indique le type de lock (`POSIX` pour un lock placé avec fcntl(2) ou lockf(3) et `FLOCK` pour un lock placé avec flock(2)). La deuxième indique si le lock est un *advisory lock* ou un *mandatory*

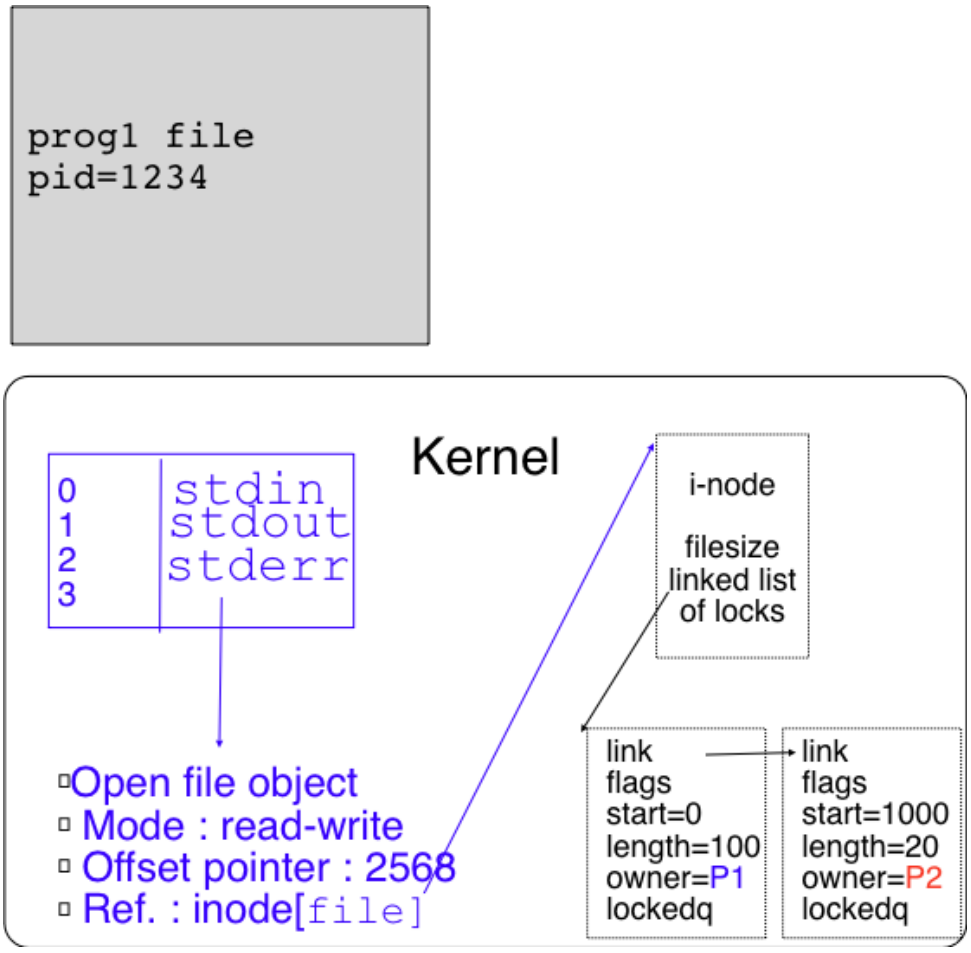


FIGURE 5.3 – Gestion des locks par le kernel

lock. La troisième spécifie si le lock protège l'écriture et/ou la lecture. La cinquième colonne est l'identifiant du processus qui possède le lock. La sixième précise le dispositif de stockage et le fichier concerné par ce lock (le dernier nombre est l'inode du fichier). Enfin, les deux dernières colonnes spécifient la section qui est couverte par le lock avec `EOF` qui indique la fin du fichier.

CHAPITRE 6

Mémoire virtuelle

6.1 La mémoire virtuelle

Le modèle d'interaction entre le processeur et la mémoire que nous avons utilisé jusqu'à présent est le modèle traditionnel. Dans ce modèle, illustré sur la figure ci-dessous, la mémoire est divisée en octets. Chaque octet est identifié par une adresse encodée sur n bits. Une telle mémoire peut donc contenir au maximum 2^n octets de données. Aujourd'hui, les processeurs utilisent généralement des adresses sur 32 ou 64 bits. Avec des adresses sur 32 bits, la mémoire peut stocker $4.294.967.296$ octets de données. Avec des adresses sur 64 bits, la capacité de stockage de la mémoire monte à $18.446.744.073.709.551.616$ octets. Si on trouve facilement aujourd'hui des mémoires avec une capacité de $4.294.967.296$ octets, il n'en existe pas encore qui sont capables de stocker $18.446.744.073.709.551.616$ et il faudra probablement quelques années avant que de telles capacités ne soient utilisables en pratique.

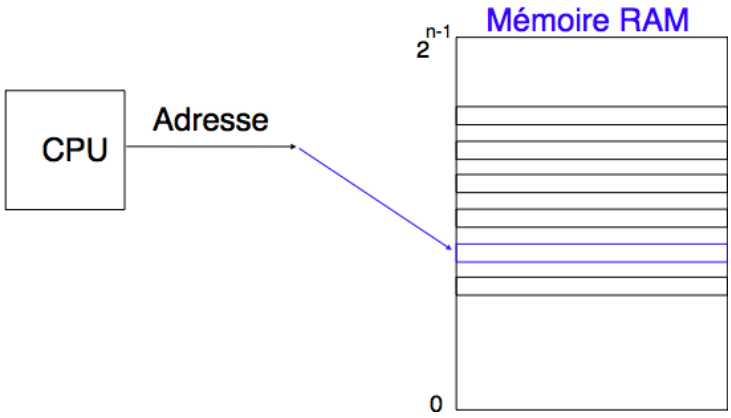


FIGURE 6.1 – Modèle simple d'interaction entre le processeur et la mémoire

Ce modèle correspond au fonctionnement de processeurs simples tels que ceux que l'on trouve sur des systèmes embarqués comme une machine à lessiver. Malheureusement, il ne permet pas d'expliquer et de comprendre le fonctionnement des ordinateurs actuels. Pour s'en convaincre, il suffit de réfléchir à quelques problèmes liés à l'utilisation de la mémoire sur un ordinateur fonctionnant sous Unix.

Le premier problème est lié à l'organisation d'un processus en mémoire. Sous Unix, le bas de la mémoire est réservé au code, le milieu au heap et le haut au stack. Le modèle simple d'organisation de la mémoire ne permet pas facilement de comprendre comment un tel processus peut pouvoir utiliser la mémoire sur un processeur 64 bits qui est placé dans un ordinateur qui ne dispose que de 4 GBytes de mémoire. Avec une telle quantité de mémoire, le sommet de la pile devrait se trouver à une adresse proche de 2^{32} et non 2^{64}.

Un deuxième problème est lié à l'utilisation de plusieurs processus simultanément en mémoire. Lorsque deux processus s'exécutent, ils utilisent nécessairement la même mémoire physique. Si un processus utilise l'adresse x et y place des instructions ou des données, cette adresse ne peut pas être utilisée par un autre processus. Physiquement, ces deux processus doivent utiliser des zones mémoires distinctes. Pourtant, le programme ci-dessous

affiche les adresses de `argc`, de la fonction `main` et de la fonction `printf` de la librairie standard puis effectue `sleep(20);`. Lors de l'exécution de deux instances de ce programmes simultanément, on observe ceci sur la sortie standard.

```
$ ./simple &
[pid=32955] Adresse de argc : 0x7fff5fbfe18c
[pid=32955] Adresse de main : 0x100000e28
[pid-32955] Adresse de printf : 0x7fff8a524f3a
$ ./simple 2 3 4
[pid=32956] Adresse de argc : 0x7fff5fbfe18c
[pid=32956] Adresse de main : 0x100000e28
[pid-32956] Adresse de printf : 0x7fff8a524f3a
```

Manifestement, les deux programmes utilisent exactement les mêmes adresses en mémoire. Pourtant, ces deux programmes doivent nécessairement utiliser des zones mémoires différentes pour pouvoir s'exécuter correctement. Ceci est possible grâce à l'utilisation de la *mémoire virtuelle*. Avec la *mémoire virtuelle*, deux types d'adresses sont utilisées sur le système : les *adresses virtuelles* et les *adresses réelles* ou *physiques*. Une *adresse virtuelle* est une adresse qui est utilisée à l'intérieur d'un programme. Les adresses des variables ou des fonctions de notre programme d'exemple ci-dessus sont des adresses virtuelles. Une *adresse physique* est l'adresse qui est utilisée par des puces de RAM pour les opérations d'écriture et de lecture. Ce sont les adresses physiques qui sont échangées sur le bus auquel la mémoire est connectée. Pour que les programmes puissent accéder aux instructions et données qui se trouvent en mémoire, il est nécessaire de pouvoir traduire les adresses virtuelles en adresses physiques. C'est le rôle du *MMU* ou *Memory Management Unit*. Historiquement, le *MMU* était implémenté sous la forme d'un chip séparé qui était placé entre le processeur (qui utilisait alors des adresses virtuelles) et la mémoire (qui utilise elle toujours des adresses physiques). Aujourd'hui, le *MMU* est directement intégré au processeur pour des raisons de performance, mais conceptuellement son rôle reste essentiel comme nous allons le voir.

6.1.1 La mémoire virtuelle

Le rôle principal du *MMU* est de traduire toute adresse virtuelle en une adresse physique. Avant d'expliquer comment le *MMU* peut être implémenté en pratique, il est utile de passer en revue plusieurs avantages de l'utilisation des adresses virtuelles.

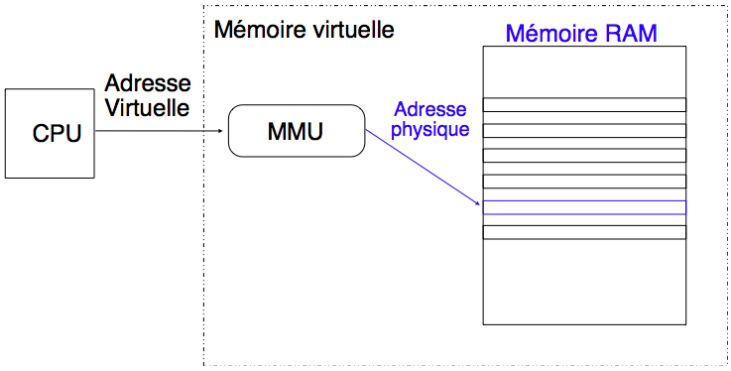


FIGURE 6.2 – *MMU* et *mémoire virtuelle*

Un premier avantage de l'utilisation de la mémoire virtuelle est qu'elle permet de découpler les adresses virtuelles des adresses physiques. Celles-ci ne doivent pas nécessairement être encodées en utilisant le même nombre de bits. La longueur des adresses dépend généralement de l'architecture du processeur et de la taille des registres qu'il utilise. Une organisation possible de la mémoire virtuelle est d'utiliser des adresses virtuelles qui sont encodées sur autant de bits que les adresses physiques, mais ce n'est pas la seule. Il est tout à fait possible d'avoir un ordinateur sur lequel les adresses virtuelles sont plus longues que les adresses physiques. C'est le cas par exemple sur les ordinateurs bon marché qui utilisent une quantité réduite de mémoire RAM. Inversement, la mémoire virtuelle permet à un serveur d'utiliser des adresses physiques qui sont plus longues que les adresses virtuelles. Cela lui permet d'utiliser une capacité de mémoire plus importante que celle autorisée par l'architecture de son processeur. Dans ce cas, un processus ne peut pas utiliser plus de mémoire que l'espace d'adressage virtuel disponible. Mais

ensemble, tous les processus fonctionnant sur l'ordinateur peuvent utiliser tout l'espace d'adressage physique disponible.

Un deuxième avantage de la mémoire virtuelle est qu'elle permet, à condition de pouvoir réaliser une traduction spécifique à chaque processus, de partager efficacement la mémoire entre plusieurs processus tout en leur permettant d'utiliser les mêmes adresses virtuelles. C'est cette particularité de la mémoire virtuelle qui nous a permis dans l'exemple précédent d'avoir deux processus qui en apparence utilisent les mêmes adresses. En effectuant une traduction spécifique à chaque processus, le *MMU* permet d'autres avantages qui sont encore plus intéressants.

Le *MMU* est capable d'effectuer des traductions d'adresses virtuelles qui sont spécifiques à chaque processus. Cela implique qu'en général la traduction de l'adresse x dans le processus $P1$ ne donnera pas la même adresse physique que la traduction de l'adresse x dans le processus $P2$. Par contre, il est tout à fait possible que la traduction de l'adresse w (resp. y) dans le processus $P1$ (resp. $P2$) donne l'adresse physique z dans les deux processus. Comme nous le verrons ultérieurement, cela permet à deux processus distincts de partager de la mémoire. Cette propriété est aussi à la base du fonctionnement des librairies partagées dans un système Unix. Dans notre exemple, la fonction `printf` qui est utilisée par les deux processus fait partie de la librairie standard. Celle-ci doit être chargée en mémoire lors de l'exécution de chacun des processus. Grâce à l'utilisation du *MMU* et de la mémoire virtuelle, une seule copie physique de la librairie standard est chargée en mémoire et tous les processus qui y font appel utilisent les instructions se trouvant dans cette copie physique. Cela permet de réduire fortement la consommation de mémoire lorsque de nombreux processus s'exécutent simultanément, ce qui est souvent le cas sur un système Unix.

Le dernier avantage de l'utilisation de la mémoire virtuelle est qu'il est possible de combiner ensemble la mémoire RAM et un ou des dispositifs de stockage tels que des disques durs ou des disques SSD pour constituer une mémoire virtuelle de plus grande capacité que la mémoire RAM disponible. Pour cela, il suffit, conceptuellement, que le *MMU* soit capable de supporter deux types d'adresses physiques : les adresses physiques en RAM et les adresses physiques qui correspondent à des données stockées sur un dispositif de stockage [1].

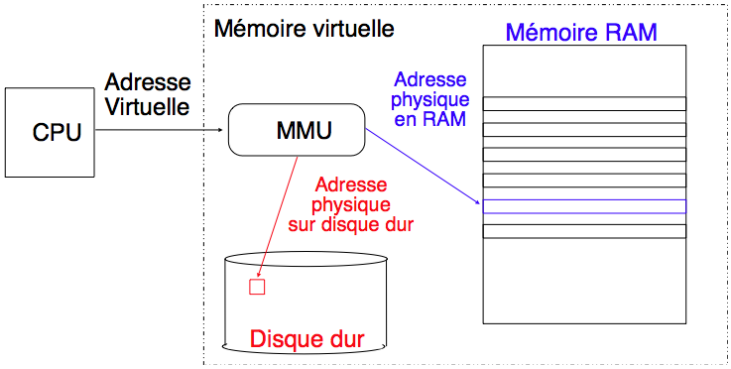


FIGURE 6.3 – Organisation de la *mémoire virtuelle*

Cette possibilité de combiner la mémoire RAM et les dispositifs de stockage offre encore plus de possibilités. Comme nous le verrons, grâce à la mémoire virtuelle, un processus pourra accéder à des fichiers via des pointeurs et des écriture/lectures en mémoire. Le chargement d'un programme pourra s'effectuer en passant par la mémoire virtuelle de façon à charger uniquement les parties du programme qui sont nécessaires en mémoire. Nous verrons également qu'il existe plusieurs appels systèmes qui permettent à des processus de contrôler leur utilisation de la mémoire virtuelle.

6.1.2 Fonctionnement de la mémoire virtuelle

Avant d'analyser comment la mémoire virtuelle peut être utilisée par les processus, il est important de bien comprendre son organisation et les principes de base de fonctionnement du *MMU*. La mémoire virtuelle combine la mémoire RAM et les dispositifs de stockage. Comme la mémoire RAM et les dispositifs de stockage ont des caractéristiques fort différentes, il n'est pas trivial de les combiner pour donner l'illusion d'une mémoire virtuelle unique.

1. En pratique, les adresses sur le disque dur ne sont pas stockées dans le *MMU* mais dans une table maintenue par le système d'exploitation. C'est le noyau qui est responsable des transferts entre le dispositif de stockage et la mémoire RAM.

Au niveau de l'adressage, la mémoire RAM permet d'adresser des octets et supporte des lectures et des écritures à n'importe quelle adresse. La mémoire RAM permet au processeur d'écrire et de lire des octets ou des mots à une position déterminée en mémoire

Un dispositif de stockage (disque dur, CD/DVD, ...) quant à lui contient un ensemble de secteurs. Chaque secteur peut être identifié par une adresse, comprenant par exemple le numéro du plateau, le numéro de la piste et le numéro du secteur sur la piste. Sur un tel dispositif, le secteur est l'unité de transfert de l'information. Cela implique que la moindre lecture/écriture sur un dispositif de stockage nécessite la lecture/écriture d'au moins 512 octets, même pour modifier un seul bit. Enfin, la dernière différence importante entre ces deux technologies est leur temps d'accès. Au niveau des mémoires RAM, les temps d'accès sont de l'ordre de quelques dizaines de nanosecondes. Pour un dispositif de stockage, les temps d'accès peuvent être de quelques dizaines de microsecondes pour un dispositif de type *Solid State Drive* ou *SSD* et jusqu'à quelques dizaines de millisecondes pour un disque dur. Les tableaux ci-dessous présentent les caractéristiques techniques de deux dispositifs de stockage [2] [3] à titre d'exemple.

La mémoire virtuelle utilise elle une unité intermédiaire qui est la *page*. Une *page* est une zone de mémoire contigüe. La taille des pages dépend de l'architecture du processeur et/ou du système d'exploitation utilisé. Une taille courante est de 4096 octets.

```
#include <unistd.h>
int sz = getpagesize();
```

Lorsqu'un programme est chargé en mémoire, par exemple lors de l'exécution de l'appel système execve(2), il est automatiquement découpé en pages. Grâce à la mémoire virtuelle, ces pages peuvent être stockée dans n'importe quelle zone de la mémoire RAM. La seule contrainte est que tous les octets qui font partie de la même page soient stockés à des adresses qui sont contigües. Cette contrainte permet de structurer les adresses virtuelles en deux parties comme représenté dans la figure ci-dessous. Une *adresse virtuelle* est donc un ensemble de bits. Les bits de poids fort servent à identifier la *page* dans laquelle une donnée est stockée. Les bits de poids faible (12 lorsque l'on utilise des pages de 4 KBytes), identifient la position de la donnée par rapport au début de la page.

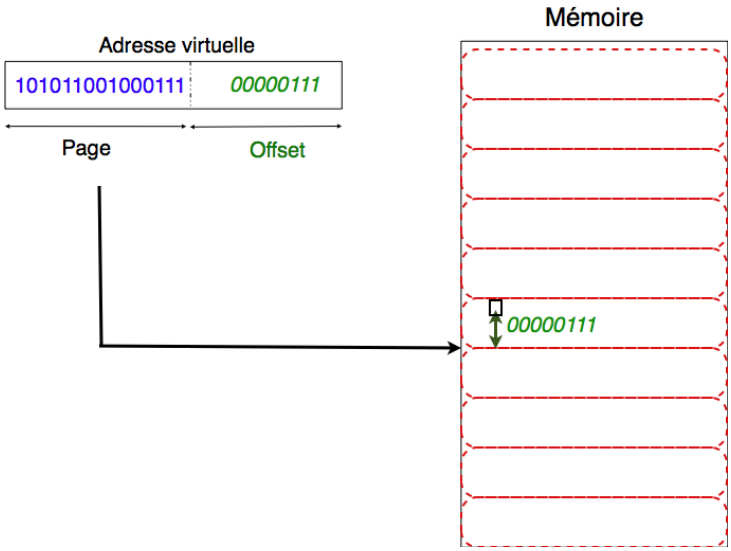


FIGURE 6.4 – Adresse virtuelle

Grâce à cette organisation des adresses virtuelles, il est possible de construire un mécanisme efficace qui permet de traduire une adresse virtuelle en une adresse réelle. La première solution qui a été proposée pour réaliser cette traduction est d'utiliser une *table des pages*. La *table des pages* est stockée en mémoire RAM et contient une ligne pour chaque page appartenant à la mémoire virtuelle. A titre d'exemple, un système utilisant des adresses virtuelles de 32 bits et des pages de 4 KBytes contient $2^{32-12} = 2^{20}$ pages. La table des pages de ce système contient donc 2^{20} lignes. Une ligne de la table des pages contient différentes informations que nous détaillerons par après. Les deux plus importantes sont :

— le *bit de validité* qui indique si la page est actuellement présente en mémoire RAM ou non

2. Source : http://www.intel.com/content/www/us/en/solid-state-drives/ssd-320-specification.html
3. Source : http://www.seagate.com/staticfiles/support/disc/manuals/desktop/Barracuda%207200.11/100507013e.pdf

— l'adresse en mémoire RAM à laquelle la page est actuellement stockée (si elle est présente en mémoire RAM, sinon une information permettant de trouver la page sur un dispositif de stockage)

La table des page est stockée en mémoire RAM comme un tableau en C. L'information correspondant à la page *0* est stockée à l'adresse de début de la table des pages. Cette adresse de début de la table des pages (`P`) est généralement stockée dans un registre du processeur pour être facilement accessible. Si une entrée [4] de la table des pages est encodée en *n* bytes, l'information correspondant à la page *1* sera stockée à l'adresse `P+n`, celle relative à la page *2* à l'adresse `P+2*n`, ... Cette organisation permet d'accéder facilement à l'entrée de la table des pages relative à la page *z*. Il suffit en effet d'y accéder depuis l'adresse `P+z*n`.

Grâce à cette table des pages, il est possible de traduire directement les adresses virtuelles en adresses physiques. Cette traduction est représentée dans la figure ci-dessous. Pour réaliser une traduction, il faut tout d'abord extraire de l'adresse virtuelle le numéro de la page. Celui-ci se trouve dans les bits de poids fort de l'adresse virtuelle. Le numéro de la page sert d'index pour récupérer l'entrée correspondant à cette page dans la table des pages. Cette entrée contient l'adresse en mémoire RAM à laquelle la page débute. Pour finaliser la traduction de l'adresse virtuelle, il suffit de concaténer les bits de poids faible de l'adresse virtuelle avec l'adresse de la page en mémoire RAM. Cette concaténation donne l'adresse réelle à laquelle la donnée est stockée en mémoire RAM. Cette adresse physique permet au processeur d'accéder directement à la donnée en mémoire.

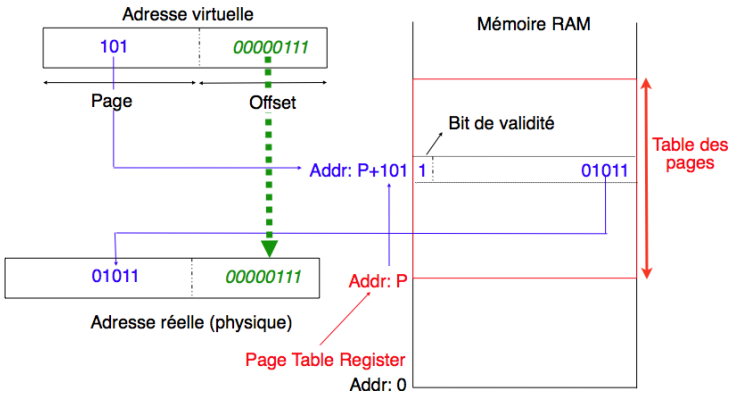


FIGURE 6.5 – Traduction d'adresses avec une table des pages

La table des pages permet de traduire les adresses virtuelles en adresses physiques. Ce faisant, elle introduit un mécanisme d'indirection entre les adresses (virtuelles) qui sont utilisées par les programmes et les adresses (réelles) qui sont utilisées par le hardware. Ce mécanisme d'indirection a de nombreuses applications comme nous le verrons par la suite.

Un point important à mentionner concernant l'utilisation d'un mécanisme de traduction des adresses est qu'il permet de découpler le choix de la taille des adresses (virtuelles) utilisées par les programmes des contraintes matérielles qui sont liées directement aux mémoires RAM utilisées. En pratique, il est très possible d'avoir des systèmes informatiques dans lesquels les adresses virtuelles sont plus longues, plus courtes ou ont la même longueur que les adresses physiques. Sur un ordinateur 32 bits actuel équipé de 4 GBytes de mémoire, il est naturel d'utiliser des adresses virtuelles de 32 bits et des adresses physiques de 32 bits également pour pouvoir accéder à l'ensemble de la mémoire. Dans ce cas, la mémoire virtuelle permet d'accéder à toute la mémoire physique. Aujourd'hui, il existe des serveurs 64 bits. Ceux-ci utilisent des adresses virtuelles de 64 bits, mais aucun ordinateur ne contient *2^64* bytes de mémoire. Par exemple, un serveur disposant de 128 GBytes de mémoire physique pourrait se contenter d'utiliser des adresses physiques de 37 bits. Dans ce cas, la mémoire virtuelle donne l'illusion qu'il est possible d'accéder à plus de mémoire que celle qui est réellement disponible. D'un autre côté, il est aussi possible de construire des serveurs qui utilisent des adresses virtuelles de 32 bits, mais disposent de plus de 4 GBytes de mémoire RAM. Dans ce cas, les adresses physiques pourront être plus longues que les adresses réelles. Quelles que soient les longueurs respectives des adresses virtuelles et physiques, la table des pages, sous le contrôle du système d'exploitation, permettra de réaliser efficacement les traductions entre les adresses virtuelles et les adresses physiques.

Pour bien comprendre la traduction des adresses virtuelles en utilisant la table des pages, considérons un système imaginaire qui utilise des adresses virtuelles encodées sur 7 bits et des adresses physiques qui sont elles enco-

4. Une entrée de la table de pages occupe généralement 32 ou 64 bits suivant les architectures de processeurs.

dées sur 6 bits. La table des pages correspondante est reprise dans le tableau ci-dessous. Comme dans la figure précédente, la ligne du bas du tableau est relative à la page *0*.

Validité	Adresse
true	00
false	—
true	11
false	—
false	—
false	—
true	01
true	10

Cette mémoire virtuelle contient quatre pages. La première couvre les adresses physiques allant de `000000` à `001111`, la seconde de `010000` à `011111`, la troisième de `100000` à `101111` et la dernière de `110000` à `111111`. Les adresses virtuelles elles vont de `0000000` à `1111111`. La traduction s'effectue sur base de la table des pages. Ainsi, l'adresse `1010001` correspond à l'octet `0001` dans la page virtuelle `101`. Sur base la table des pages, cette page se trouve en mémoire RAM (sont bit de validité est vrai) et elle démarre à l'adresse `110000`. L'adresse virtuelle `1010001` est donc traduite en l'adresse réelle `110001`. L'adresse virtuelle `0110111` correspond elle à une page qui n'est pas actuellement en mémoire RAM puisque le bit de validité correspondant à la page `011` est faux.

Si on analyse la table des pages ci-dessus, on peut remarquer que la page contenant les adresses virtuelles les plus hautes se trouve dans la zone mémoire avec les adresses physiques les plus basses. Inversement, la page qui est en mémoire RAM à l'adresse la plus élevée correspond à des adresses virtuelles qui se trouvent au milieu de l'espace d'adressage. Ce découplage entre l'adresse virtuelle et la localisation physique de la page en mémoire est un des avantages importants de la mémoire virtuelle.

La mémoire virtuelle a aussi un rôle important à jouer lorsque plusieurs processus s'exécutent simultanément. Comme indiqué ci-dessus, l'adresse de la table des pages est stockée dans un des registre du processeur. L'utilisation de ce registre permet d'avoir une table des pages pour chaque processus. Pour cela, il suffit qu'une zone de mémoire RAM soit réservée pour chaque processus et que le système d'exploitation y stocke la table des pages du processus. Lors d'un changement de contexte, le système d'exploitation modifie le registre de table des pages de façon à ce qu'il pointe vers la table des pages du processus qui s'exécute. Ce mécanisme est particulièrement utile et efficace.

A titre d'exemple, considérons un système imaginaire utilisant des adresses virtuelles sur 6 bits et des adresses physiques sur 8 bits. Deux processus s'exécutent sur ce système et ils utilisent chacun trois pages, deux pages dans le bas de l'espace d'adressage virtuel qui correspondent à leur segment de code et une page dans le haut de l'espace d'adressage virtuel qui correspond à leur pile. Le premier tableau ci-dessous présente la table des pages du processus `P1`.

Validité	Adresse
true	0011
false	—
true	1001
true	1000

Le processus `P2` a lui aussi sa table des pages. Celle-ci pointe vers des adresses physiques qui sont différentes de celle utilisées par le processus `P1`. L'utilisation d'une table des pages par processus permet à deux processus distincts d'utiliser les mêmes adresses virtuelles.

Validité	Adresse
true	0000
false	—
true	1111
true	1110

Lorsque le processus `P1` s'exécute, c'est sa table des pages qui est utilisée par le processeur pour la traduction des adresses virtuelles en adresses physiques. Ainsi, l'adresse `011101` est traduite en l'adresse `10011101`. Par contre, lorsque le processus `P2` s'exécuté, cette adresse `011101` est traduite grâce à la table des pages de ce processus en l'adresse `11111101`.

Note : Performance de la mémoire virtuelle

Grâce à son mécanisme d'indirection entre les adresses virtuelles et les adresses physiques, la mémoire virtuelle permet de nombreuses applications comme nous le verrons dans les sections qui suivent. Cependant, la mémoire virtuelle peut avoir un impact important au niveau des performances des accès à une donnée en mémoire. Pour cela, il est intéressant d'analyser en détails ce qu'il se passe lors de chaque accès à la mémoire. Pour accéder à une donnée en mémoire, le *MMU* doit d'abord consulter la table des pages pour traduire l'adresse virtuelle en une adresse physique correspondante. Ce n'est qu'après avoir obtenu cette adresse physique que le processeur peut effectuer l'accès à la mémoire RAM. En pratique, l'utilisation d'une table des pages a comme conséquence de doubler le temps d'accès à une donnée en mémoire. Lorsque la mémoire virtuelle a été inventée, ce doublement du temps d'accès à la mémoire n'était pas une limitation car les mémoires RAM étaient nettement plus rapides que les processeurs. Aujourd'hui, la situation est complètement inversée puisque les processeurs sont déjà fortement ralentis par les temps d'accès à la mémoire RAM. Doubler ce temps d'accès aurait un impact négatif sur les performances des processeurs. Pour faire face à ce problème, les processeurs actuels disposent tous d'un *Translation Lookaside Buffer* (*TLB*). Ce *TLB* est en fait une sorte de *mémoire cache* qui permet de stocker dans une mémoire rapide se trouvant sur le processeur certaines lignes de la *table des pages*. Les détails de gestion du *TLB* sortent du cadre de ce cours [HennessyPatterson]. Grâce à l'utilisation du *TLB*, la plupart des traductions des adresses virtuelles en adresses physique peuvent être obtenus sans devoir directement consulter la table des pages.

La table des pages d'un processus contrôle les adresses physiques auxquelles le processus a accès. Pour garantir la sécurité d'un système informatique, il faut bien entendu éviter qu'un processus ne puisse modifier lui-même et sans contrôle sa table des pages. Toutes les manipulations de la table des pages ou du registre de table des pages se font sous le contrôle du système d'exploitation. La modification du registre de table des pages est une opération privilégiée qui ne peut être exécutée que par le système d'exploitation.

En termes de sécurité, une entrée de la table des pages contient également des bits de permission qui sont contrôlés par le système d'exploitation et spécifient quelles opérations peuvent être effectuées sur chaque page. Une entrée de la table des pages contient trois bits de permissions :

- *R* bit. Ce bit indique si le processus peut accéder en lecture à la page se trouvant en mémoire physique.
- *W* bit. Ce bit indique si le processus peut modifier le contenu de la page se trouvant en mémoire physique
- *X* bit. Ce bit indique si la page contient des instructions qui peuvent être exécutées par le processeur ou des données.

Ces bits de protection sont généralement fixés par le système d'exploitation. Par exemple, le segment code qui ne contient que des instructions à exécuter pourra être stocké dans des pages avec les bits *R* et *X* mais pas le bit *W* pour éviter que le processus ne modifie les instructions qu'il exécute. Le stack par contre sera placé dans des pages avec les bits *R* et *W* mais pas le bit *X*. Cette technique est utilisée dans les systèmes d'exploitation récents pour éviter qu'un problème de buffer overflow sur le stack ne conduise à l'exécution d'instructions qui ne font pas partie du processus. Le heap peut utiliser les mêmes bits de protection. Enfin, les pages qui n'ont pas été allouées au processus, notamment celles se trouvant entre le heap et le stack auront toutes leurs bits de protection mis à faux. Cela permet au processeur de détecter les accès à de la mémoire qui n'a pas été allouée au processus. Un tel accès provoquera la génération d'une *segmentation fault* et l'envoi du signal correspondant.

Même si ces bits de protection sont contrôlés par le système d'exploitation, il est parfois utile à un processus de modifier les bits de permissions qui sont associés à certaines de ses pages. Cela peut se faire via l'appel système mprotect(2).

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

int mprotect(const void *addr, size_t len, int prot);
```

Cet appel système prend trois arguments. Le première est un pointeur vers le début de la zone mémoire dont il faut modifier les bits de protection. Le second est la longueur de la zone mémoire concernée et le dernier la protection souhaitée. Celle-ci est spécifiée en utilisant les constantes `PROT_NONE`, `PROT_READ`, `PROT_WRITE` et `PROT_EXEC` qui peuvent être combinées en utilisant une disjonction logique. La protection demandée ne peut pas être plus libérale que la protection qui est déjà fixée par le système d'exploitation. Dans ce cas, le système d'exploitation génère un signal `SIGSEGV`.

6.1.3 Utilisation des dispositifs de stockage

La mémoire virtuelle permet non seulement à des pages d'un processus d'être placées à différents endroits de la mémoire, mais aussi elle permet de combiner la mémoire RAM et les dispositifs de stockage de façon transparente pour les processus.

Une partie des pages qui composent la mémoire virtuelle peut être stockée sur un dispositif de stockage (disque dur, SSD, ...). En pratique, la mémoire RAM peut jouer le rôle d'une sorte de mémoire cache pour la mémoire virtuelle. Les pages qui sont le plus fréquemment utilisées sont placées en mémoire RAM par le système d'exploitation et les pages les moins utilisées sont elles placées sur un dispositif de stockage et ramenées en mémoire RAM lorsqu'elle sont utilisées par le processeur.

Pour bien comprendre cette utilisation de la mémoire virtuelle, il nous faut revenir à la table des pages. Celle-ci comprend autant d'entrées qu'il y a de pages dans l'espace d'adressage d'un processus. Nous avons vu qu'une entrée de cette table pouvait être structurée comme dans la figure ci-dessous.

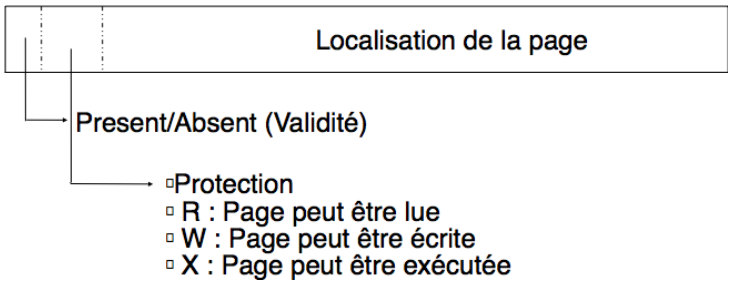


FIGURE 6.6 – Entrée de la table des pages

Le bit de validité indique si la page est présente en mémoire RAM ou non. Lorsque la page est présente en mémoire RAM, les bits de poids faible de l'entrée de la table des pages contiennent l'adresse physique de la page en mémoire RAM. Lorsque le bit de validité a comme valeur *faux*, cela signifie que la page n'existe pas (elle n'a jamais été créée) ou qu'elle est actuellement stockée sur un dispositif de stockage. Si la page n'existe pas, aucun de ses bits de permission n'aura comme valeur *vrai* et tout accès à cette page provoquera une *segmentation fault*. Si par contre la page existe mais se trouve sur un dispositif de stockage, alors l'information de localisation pointera vers une structure de données qui est maintenue par le système d'exploitation et contient la localisation physique de la donnée sur un dispositif de stockage.

Schématiquement, ces informations de localisation des pages peuvent être de deux types. Lorsqu'un dispositif de stockage, ou une partition d'un tel dispositif, est dédié au stockage de pages de la mémoire virtuelle, alors la localisation d'une page est composée de l'identifiant du dispositif et du numéro du secteur sur le dispositif. Ce sera le cas lorsque par exemple une *partition de swap* est utilisée. Sous Linux, le fichier `/proc/swaps` contient la liste des partitions de swap qui sont utilisées pour stocker les pages de la mémoire virtuelle avec leur type, leur taille et leur utilisation. Une telle partition de swap peut être créée avec l'utilitaire mkswap(8). Elle est activée en exécutant la commande swapon(8). Celle-ci est généralement lancée automatiquement lors du démarrage du système.

```
$ cat /proc/swaps
Filename                Type            Size    Used    Priority
/dev/sda3               partition       8193140 444948  -1
```

Outre les partitions de swap, il est également possible de stocker des pages de la mémoire virtuelle dans des fichiers. Dans ce cas, la localisation d'une page comprend le dispositif, l'*inode* du fichier et l'offset à partir duquel la page est accessible dans le fichier. En pratique, les partitions de swap sont un peu plus rapides que les fichiers de swap car les secteurs qui composent une telle partition sont contigus, ce qui n'est pas toujours le cas avec un fichier de swap. D'un autre côté, il est plus facile d'ajouter ou de retirer des fichiers de swap que des partitions de swap sur un dispositif de stockage. En pratique, les deux techniques peuvent être utilisées.

A ce stade, il est utile d'analyser à nouveau le fonctionnement de la mémoire virtuelle. En toute généralité, celle-ci est découpée en pages et comprend une mémoire RAM et un ou plusieurs dispositifs de stockage. Pour simplifier la présentation, nous supposons qu'un seul disque dur est utilisé.

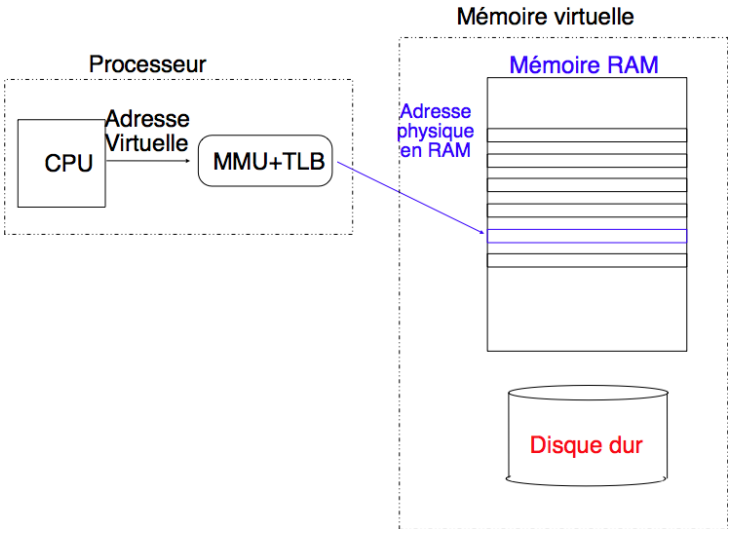


FIGURE 6.7 – La mémoire virtuelle

Les processus utilisent des adresses virtuelles pour représenter les positions des données et des instructions en mémoire virtuelles. Ces adresses virtuelles sont donc utilisées par le CPU chaque fois qu'il doit charger ou sauvegarder une donnée ou une instruction en mémoire. Comme nous l'avons vu, le *MMU* permet de traduire les adresses virtuelles en adresses réelles. Pour des raisons de performance, le *MMU* est intégré directement sur le processeur et il comprend un *TLB* qui sert de cache pour les entrées de la table des pages du processus qui est en train de s'exécuter.

Considérons une opération de lecture faite par le CPU. Pour réaliser cette opération, le CPU fournit l'adresse virtuelle au *MMU*. Celui-ci va consulter le *TLB* pour traduire l'adresse virtuelle demandée. Cette traduction peut nécessiter différentes opérations. Supposons que l'entrée de la table des pages demandées se trouve dans le *TLB*.

- Si le *bit de validité* de la page est *vrai*, la page demandée se situe en mémoire RAM. Dans ce cas, le *MMU* vérifie via les bits de permissions si l'accès demandé (dans ce cas une lecture, mais un raisonnement similaire est valable pour une écriture ou le chargement d'une instruction) est valide.
- Si l'accès est autorisé, le *MMU* retourne l'adresse réelle et le processeur accède aux données.
- Si l'accès n'est pas autorisé, le processeur génère une interruption. Le processus ayant tenté d'accéder à une zone de mémoire ne faisant pas partie de son espace d'adressage virtuel, c'est au système d'exploitation de réagir. Celui-ci enverra un signal segmentation fault, `SIGSEGV`, au processus qui a tenté cet accès.
- Si le *bit de validité* de la page est *faux*, la page demandée ne se trouve pas en mémoire RAM. Deux cas de figure sont possibles :
  - les bits de permission ne permettent aucun accès à la page. Dans ce cas, la page n'existe pas et le *MMU* va générer une interruption qui va provoquer l'exécution d'une routine de traitement

d'interruption du système d'exploitation. Lors du traitement de cette opération, le noyau va envoyer un signal segmentation fault au processus qui a tenté cet accès.

— les bits de permission permettent l'accès à la page. On parle dans ce cas de *page fault*, c'est-à-dire qu'une page nécessaire à l'exécution du processus n'est pas disponible en mémoire RAM. Vu les temps d'accès et la complexité d'accéder à une page sur un disque dur (via une partition, un fichier de swap ou un fichier normal), le *MMU* ne peut pas accéder directement à la donnée sur le disque dur. Le *MMU* va donc générer une interruption qui va forcer l'exécution d'une routine de traitement d'interruption par le noyau. Cette routine va identifier la page manquante et préparer son transfert du disque dur vers la mémoire. Ce transfert peut durer plusieurs dizaines de millisecondes, ce qui est un temps très long par rapport à l'exécution d'instructions par le processeur. Tant que cette page n'est pas disponible en mémoire RAM, le processus ne peut continuer son exécution. Il passe dans l'état bloqué et le noyau effectue un changement de contexte pour exécuter un autre processus. Lorsque la page manquante aura été rapatriée depuis le disque dur en mémoire RAM, le noyau pourra relancer le processus qu'il avait bloqué afin de retenter l'accès mémoire qui vient d'échouer.

Durant son exécution, un système doit pouvoir gérer des pages qui se trouvent en mémoire RAM et des pages qui sont stockées sur le disque dur. Lorsque la mémoire RAM en entièrement remplies de pages, il peut être nécessaire d'y libérer de l'espace mémoire et déplaçant des pages vers un des dispositifs de stockage. C'est le rôle des algorithmes de remplacement de pages.

6.1.4 Stratégies de remplacements de pages

C'est le système d'exploitation qui prend en charge les transferts de pages entre les dispositifs de stockage et la mémoire. Tant que la mémoire RAM n'est pas remplie, ces transferts sont simples, il suffit de ramener une ou plusieurs pages du dispositif de stockage vers la mémoire RAM. En général, le système d'exploitation cherchera à exploiter le principe de localité lors de ces transferts. Lorsqu'une page manque en mémoire RAM, le noyau programmera le chargement de cette page, mais aussi d'autres pages du même processus ayant des adresses proches.

Lorsque la mémoire RAM est remplie et qu'il faut ramener une page depuis un dispositif de stockage, le problème est plus délicat. Pour pouvoir charger cette nouvelle page en mémoire RAM, le système d'exploitation doit libérer de la mémoire. Pour cela, il doit implémenter une *stratégie de remplacement des pages* en mémoire. Cette stratégie définit quelle page doit être préférentiellement retirée de la mémoire RAM et placée sur le dispositif de stockage. Différentes stratégies sont possibles. Elles résultent en général d'un compromis entre la quantité d'information de contrôle qui est stockée dans la table des pages et les performances de la stratégie de remplacement des pages.

Une première stratégie de remplacement de pages pourrait être de sauvegarder les identifiants des pages dans une *file FIFO*. Chaque fois qu'une page est créée par le noyau, son identifiant est placé à la fin de la *file FIFO*. Lorsque la mémoire est pleine et qu'une page doit être retirée de la mémoire RAM, le noyau pourrait choisir la page dont l'identifiant se trouve en tête de la *file FIFO*. Cette stratégie a l'avantage d'être simple à implémenter, mais remettre sur disque la page la plus anciennement créée n'est pas toujours la solution la plus efficace du point de vue des performances. En effet, cette page peut très bien être une des pages les plus utilisées par le processeur. Si elle est remise sur le disque, elle risque de devoir être récupérée peu de temps après.

Au niveau des performances, la meilleure stratégie de remplacement de pages serait de sauvegarder sur le disque dur les pages qui seront utilisées par le processeur d'ici le plus de temps possible. Malheureusement, cette stratégie nécessite de prévoir le futur, une fonctionnalité qui n'existe pas dans les systèmes d'exploitation actuels... Une solution alternative serait de comptabiliser les accès aux différentes pages et de sauvegarder sur disque les pages qui ont été les moins utilisées. Cette solution est séduisante d'une point de vue théorique car en disposant de statistiques sur l'utilisation des pages, le système d'exploitation devrait pouvoir être capable de mieux prédire les pages qui seront nécessaires dans le futur et les conserver en mémoire RAM. Du point de vue de l'implémentation par contre, cette solution est loin d'être réaliste. En effet, pour maintenir un compteur du nombre d'accès à une page, il faut consommer de la mémoire supplémentaire dans chaque entrée de la table des pages. Mais il faut aussi que le *TLB* puisse incrémenter ce compteur lors de chaque accès à une de ces entrées. Cela augmente inutilement la complexité du *TLB*.

Stocker dans le *TLB* l'instant du dernier accès à une page de façon à pouvoir déterminer quelles sont les pages auxquelles le système a accédé depuis le plus longtemps est une autre solution séduisante d'un point de vue théorique. Du point de vue de l'implémentation, c'est loin d'être facilement réalisable. Tout d'abord, pour que cet instant soit utile, il faut probablement disposer d'une résolution d'une milliseconde voire mieux. Une telle

résolution consommera au moins quelques dizaines de bits dans chaque entrée de la table des pages. En outre, le *TLB* devra pouvoir mettre à jour cette information lors de chaque accès.

Face à ces difficultés d'implémentation, la plupart des stratégies de remplacement de pages s'appuient sur deux bits qui se trouvent dans chaque entrée de la table des pages [HennessyPatterson]. Il est relativement facile de supporter ces deux bits dans une implémentation du *TLB* et leur présence n'augmente pas de façon significative la mémoire occupée par une entrée de la table des pages.

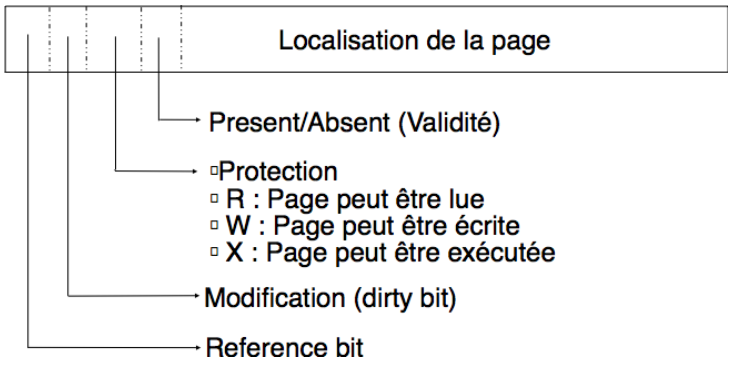


FIGURE 6.8 – Une entrée complète de la table des pages

Outre les bits de validité et de permission, une entrée de la table des pages contient les bits de contrôle suivants :

— le *bit de référence* est mis à vrai par le *MMU* dans le *TLB* à chaque accès à une donnée se trouvant dans la page correspondante, que cet accès soit en lecture ou en écriture

— le *bit de modification* ou *dirty bit* est mis à vrai par le *MMU* chaque fois qu'une opération d'écriture est réalisée dans cette page.

Ces deux bits sont mis à jour par le *MMU* à l'intérieur du *TLB*. Lorsqu'une entrée de la table des pages est retirée du *TLB* pour être remise en mémoire, que ce soit à l'occasion d'un changement de contexte ou parce que le *TLB* est plein, les valeurs de ces deux bits sont recopiées dans l'entrée correspondante de la table des pages. En somme, le *TLB* fonctionne comme une cache en *write-back* pour ces deux bits de contrôle.

Les bits de référence et de modification sur utilisés par la plupart des algorithmes de remplacement de pages. Le bit de référence est généralement utilisé par le système d'exploitation pour déterminer quelles sont les pages auxquelles un processus accède actuellement. Pour cela, le noyau va régulièrement remettre à *faux* les bits de validité des entrées des tables de pages. Lorsque une entrée de la table des pages est chargée dans le *TLB* suite à un *page fault*, son *bit de référence* est mis à *vrai*. Il en va de même chaque fois que le processeur accède à une donnée dans cette page.

La stratégie de remplacement utilisant une *file FIFO* que nous avons mentionné précédemment peut être améliorée en utilisant le *bit de référence*. Plutôt que de remettre sur disque la page dont l'identifiant est en tête de la file il suffit de regarder son *bit de référence*. Si celui-ci a la valeur *faux*, la page n'a pas été utilisée récemment et peut donc être retirée de la mémoire RAM. Sinon, le *bit de référence* est remis à *faux* et l'identifiant de la page est replacé en fin de file. L'algorithme de remplacement de page passe ensuite à la page suivante dans la file et continue jusqu'à trouver suffisamment de pages ayant leur bit de référence mis à *faux*.

Une autre stratégie est de combiner le *bit de référence* et le *dirty bit*. Dans ce cas, le noyau du système d'exploitation va régulièrement remettre à la valeur *faux* tous les bits de référence (par exemple toutes les secondes). Lorsque la mémoire RAM est pleine et qu'il faut libérer de l'espace mémoire pour charger de nouvelles pages, l'algorithme de remplacement de pages va grouper les pages en mémoire en quatre classes.

1. La première classe comprend les pages dont le *bit de référence* et le *bit de modification* ont comme valeur *faux*. Ces pages n'ont pas été utilisées récemment et sont identiques à la version qui est déjà stockée sur disque. Elles peuvent donc être retirée de la mémoire RAM sans nécessiter de transfert vers le disque.
2. La deuxième classe comprend les pages dont le *bit de référence* a comme valeur *faux* mais le *bit de modification* a comme valeur *vrai*. Ces pages n'ont pas été utilisées récemment, mais doivent être transférées vers le disque avant d'être retirées de la mémoire RAM.
3. La troisième classe comprend les pages dont le *bit de référence* a comme valeur *vrai* mais le *bit de modification* a comme valeur *faux*. Ces pages ont été utilisées récemment mais peuvent être retirées de la mémoire RAM sans nécessiter de transfert vers le disque.

4. La dernière classe comprend les pages dont les bits de référence et de modification ont comme valeur *vrai*. Ces pages ont étés utilisées récemment et il faut les transférer vers le disque avant de les retirer de la mémoire RAM.

Si l'algorithme de remplacement de pages doit retirer des pages de la mémoire RAM, il commencera par retirer des pages de la première classe, et ensuite de la deuxième, ...

Des algorithmes plus performants ont été proposés et sont utilisés en pratique. Une description détaillée de ces algorithmes sort du cadre de ce cours d'introduction mais peut être trouvée dans un livre consacré aux systèmes d'exploitation comme [Tanenbaum+2009].

Note : Swapping et pagination

Grâce à l'utilisation de la mémoire virtuelle qui est découpée en pages, il est possible de stocker certaines parties de processus sur un dispositif de stockage plutôt qu'en mémoire RAM. Cette technique de *pagination* permet au système d'exploitation de gérer efficacement la mémoire et de réserver la mémoire RAM aux parties de processus qui sont nécessaires à leur exécution. Grâce à la découpe en pages, il est possible de transférer de petites parties d'un processus temporairement sur un dispositif de stockage. Aujourd'hui, la *pagination* est très largement utilisée, mais ce n'est pas la seule technique qui permette de placer temporairement l'espace mémoire utilisé par un processus sur disque. Le *swapping* est une technique plus ancienne mais qui est encore utilisée en pratique. Le *swapping* est plus radical que la *pagination* puisque cette technique permet au noyau de sauvegarder sur disque la quasi totalité de la mémoire utilisée par un processus. Le noyau fera appel au swapping lorsque la mémoire RAM est surchargée et pour des processus qui sont depuis longtemps bloqués par exemple en attente d'un opération d'entrée/sortie. Lorsque le noyau manque de mémoire, il est plus efficace de sauvegarder un processus complet plutôt que de transférer des pages de différents processus. Un tel processus swappé sera réactivé et ramené en mémoire par le noyau lorsqu'il repassera dans l'état *Running*, par exemple suite à la réussite d'une opération d'entrée/sortie.

6.1.5 Utilisations de la mémoire virtuelle

Comme nous l'avons vu précédemment, la mémoire virtuelle est découpée en pages et elle permet de découpler les adresses utilisées par les processus des adresses physiques. Grâce à la table des pages, il est possible de placer les pages d'un processus à n'importe quel endroit de la mémoire RAM. Mais la mémoire virtuelle permet également d'interagir avec les dispositifs de stockage comme si ils faisaient partie de la mémoire accessible au processus.

6.1.6 Fichiers mappés en mémoire

Lorsqu'un processus Unix veut lire ou écrire des données dans un fichier, il utilise en général les appels systèmes open(2), read(2), write(2) et close(2) directement ou à travers une librairie de plus haut niveau comme la libraire d'entrées/sorties standard. Ce n'est pas la seule façon pour accéder à des données sur un dispositif de stockage. Grâce à la mémoire virtuelle, il est possible de placer le contenu d'un fichier ou d'une partie de fichier dans une zone de la mémoire du processus. Cette opération peut être effectuée en utilisant l'appel système mmap(2). Cet appel système permet de rendre des pages d'un fichier accessibles à travers la table des pages du processus comme illustré dans la figure ci-dessous.

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

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

L'appel système mmap(2) prend six arguments, c'est un des appels systèmes qui utilise le plus d'arguments. Il permet de rendre accessible une portion d'un fichier via la mémoire d'un processus. Le cinquième argument est le descripteur du fichier qui doit être mappé. Celui-ci doit avoir été préalablement ouvert avec l'appel système open(2). Le sixième argument spécifie l'offset à partir duquel le fichier doit être mappé, `0` correspondant au début du fichier. Le premier argument est l'adresse à laquelle la première page du fichier doit être mappée. Généralement, cet argument est mis à `NULL` de façon à laisser le noyau choisir l'adresse la plus appropriée. Si cette adresse est spécifiée, elle doit être un multiple de la taille des pages. Le deuxième argument est la longueur de la zone

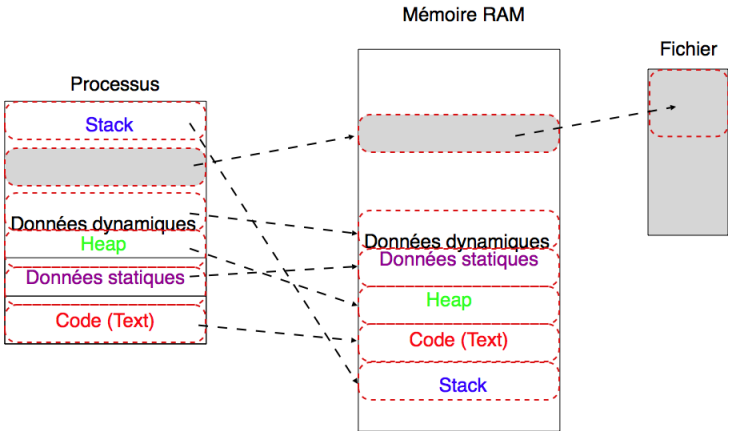


FIGURE 6.9 – Fichiers mappés en mémoire

du fichier qui doit être mappée en mémoire. Le troisième argument contient des drapeaux qui spécifient les permissions d'accès aux données mappées. Cet argument peut soit être `PROT_NONE`, ce qui indique que la page est inaccessible soit une permission classique :

— `PROT_EXEC`, les pages mappées contiennent des instructions qui peuvent être exécutées
— `PROT_READ`, les pages mappées contiennent des données qui peuvent être lues
— `PROT_WRITE`, les pages mappées contiennent des données qui peuvent être modifiées

Ces drapeaux peuvent être combinés avec une disjonction logique. Le quatrième argument est un drapeau qui indique comment les pages doivent être mappées en mémoire. Ce drapeau spécifie comment un fichier qui est mappé par deux ou plusieurs processus doit être traité. Deux drapeaux sont possibles :

— `MAP_PRIVATE`. Dans ce cas, les pages du fichier sont mappées dans chaque processus, mais si un processus modifie une page, cette modification n'est pas répercutée aux autres processus qui ont mappé la même page de ce fichier.
— `MAP_SHARED`. Dans ce cas, plusieurs processus peuvent accéder et modifier la page qui est mappée en mémoire. Lorsqu'un processus modifie le contenu d'une page, la modification est visible aux autres processus. Par contre, le fichier qui est mappé en mémoire n'est modifié que lorsque le noyau du système d'exploitation décide d'écrire les données modifiées sur le dispositif de stockage. Ces écritures dépendent de nombreux facteurs, dont la charge du système. Si un processus veut être sûr des écritures sur disque des modifications qu'il a fait à un fichier mappé un mémoire, il doit exécuter l'appel système msync(2) ou supprimer le mapping via munmap(2).

Ces deux drapeaux peuvent dans certains cas particuliers être combinés avec d'autres drapeaux définis dans la page de manuel de mmap(2).

Lorsque mmap(2) réussit, il retourne l'adresse du début de la zone mappée en mémoire. En cas d'erreur, la constante `MAP_FAILED` est retournée et `errno` est mis à jour en conséquence.

L'appel système msync(2) permet de forcer l'écriture sur disque d'une zone mappée en mémoire. Le premier argument est l'adresse du début de la zone qui doit être écrite sur disque. Le deuxième argument est la longueur de la zone qui doit être écrite sur le disque. Enfin, le dernier contient un drapeau qui spécifie comment les pages correspondantes doivent être écrites sur le disque. La drapeau `MS_SYNC` indique que l'appel msync(2) doit bloquer tant que les données n'ont pas été écrites. Le drapeau `MS_ASYNC` indique au noyau que l'écriture doit être démarrée, mais l'appel système peut se terminer avant que toutes les pages modifiées aient été écrites sur disque.

```
#include <sys/mman.h>
int msync(void *addr, size_t length, int flags);
```

Lorsqu'un processus a fini d'utiliser un fichier mappé en mémoire, il doit d'abord supprimer le mapping en utilisant l'appel système munmap(2). Cet appel système prend deux arguments. Le premier doit être un multiple de la taille d'une page [5]. Le second est la taille de la zone pour laquelle le mapping doit être retiré.

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

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

5. Il est possible d'obtenir la taille des pages utilisée sur un système via les appels sysconf(3) ou getpagesize(2)

A titre d'exemple d'utilisation de mmap(2) et munmap(2), le programme ci-dessous implémente l'équivalent de la commande cp(1). Il prend comme arguments deux noms de fichiers et copie le contenu du premier dans le second. La copie se fait en mappant le premier fichier entièrement en mémoire et en utilisant la fonction memcpy(3) pour réaliser la copie. Cette solution fonctionne avec de petits fichiers. Avec de gros fichiers, elle n'est pas très efficace car tout le fichier doit être mappé en mémoire.

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

int main (int argc, char *argv[]) {
 int file1, file2;
 void *src, *dst;
 struct stat file_stat;
 char dummy=0;

 if (argc != 3) {
   fprintf(stderr,"Usage : cp2 source dest\n");
   exit(EXIT_FAILURE);
 }
 // ouverture fichier source
 if ((file1 = open (argv[1], O_RDONLY)) < 0) {
   perror("open(source)");
   exit(EXIT_FAILURE);
 }

 if (fstat (file1,&file_stat) < 0) {
   perror("fstat");
   exit(EXIT_FAILURE);
 }
 // ouverture fichier destination
 if ((file2 = open (argv[2], O_RDWR | O_CREAT | O_TRUNC, file_stat.st_mode)) < 0) {
   perror("open(dest)");
   exit(EXIT_FAILURE);
 }

 // le fichier destination doit avoir la même taille que le source
 if (lseek (file2, file_stat.st_size - 1, SEEK_SET) == -1) {
   perror("lseek");
   exit(EXIT_FAILURE);
 }

 // écriture en fin de fichier
 if (write (file2, &dummy, sizeof(char)) != 1) {
   perror("write");
   exit(EXIT_FAILURE);
 }

 // mmap fichier source
 if ((src = mmap (NULL, file_stat.st_size, PROT_READ, MAP_SHARED, file1, 0))  == NULL) {
   perror("mmap(src)");
   exit(EXIT_FAILURE);
 }

 // mmap fichier destination
 if ((dst = mmap (NULL, file_stat.st_size, PROT_READ | PROT_WRITE, MAP_SHARED, file2, 0)) == NULL
```

```
    perror("mmap(src)");
    exit(EXIT_FAILURE);
  }

  // copie complète
  memcpy (dst, src, file_stat.st_size);

  // libération mémoire
  if(munmap(src,file_stat.st_size)<0) {
    perror("munmap(src)");
    exit(EXIT_FAILURE);
  }

  if(munmap(dst,file_stat.st_size)<0) {
    perror("munmap(dst)");
    exit(EXIT_FAILURE);
  }
  // fermeture fichiers
  if(close(file1)<0) {
    perror("close(file1)");
    exit(EXIT_FAILURE);
  }

  if(close(file2)<0) {
    perror("close(file2)");
    exit(EXIT_FAILURE);
  }
  return(EXIT_SUCCESS);
}
```

6.1.7 Mémoire partagée

Dans les exemples précédents, nous avons supposé qu'il existait une correspondance biunivoque entre chaque page de la mémoire virtuelle et une page en mémoire RAM. C'est souvent le cas, mais ce n'est pas nécessaire. Il est tout à fait possible d'avoir plusieurs pages de la mémoire virtuelle qui appartiennent à des processus différents mais pointent vers la même page en mémoire physique. Ce partage d'une même page physique entre plusieurs pages de la mémoire virtuelle a plusieurs utilisations en pratique.

Revenons aux threads POSIX. Lorsqu'un processus crée un nouveau thread d'exécution, celui-ci a un accès complet au segment code, aux variables globales et au heap du processus. Par contre, le thread et le processus ont chacun un stack qui leur est propre. Comme nous l'avons indiqué lors de la présentation des threads, ceux-ci peuvent être implémentés en utilisant une libraire ou avec l'aide du système d'exploitation. Du point de vue de la mémoire, lorsqu'une librairie telle que *gnuth* est utilisée pour créer un thread, la librairie réserve une zone de mémoire sur le heap pour ce thread. Cette zone mémoire contient le stack qui est spécifique au thread. Celui-ci a été alloué en utilisant malloc(3) et a généralement une taille fixe. Avec la mémoire virtuelle, il est possible d'implémenter les threads plus efficacement avec l'aide du système d'exploitation. Lors de la création d'un thread, celui-ci va tout d'abord créer une nouvelle table des pages pour le thread. Celle-ci sera initialisée en copiant toutes les entrées de la table des pages du processus, sauf celles qui correspondent au stack. De cette façon, le processus *père* et le thread auront accès aux mêmes segments de code, aux même variables globales et au même heap. Toute modification faite par le processus père à une variable globale ou à une information stockée sur le heap sera immédiatement accessible au thread et inversement. L'entrée de la table des pages du thread correspondant à son stack pointera vers une page qui sera spécifique au thread. Cette page aura été initialisée par le système d'exploitation avec l'argument passé par le processus à la fonction pthread_create(3). La figure ci-dessous illustre ce partage de table des pages après la création d'un thread.

En exploitant intelligemment la table des pages, il est également possible de permettre à deux processus distincts d'avoir accès à la même zone de mémoire physique. Si deux processus peuvent accéder simultanément à la même zone de mémoire, ils peuvent l'utiliser pour communiquer plus efficacement qu'en utilisant des pipes par exemple. Cette technique porte le nom de *mémoire partagée*. Elle nécessite une modification de la table des pages des processus qui veulent partager une même zone mémoire. Pour comprendre le fonctionnement de cette *mémoire*

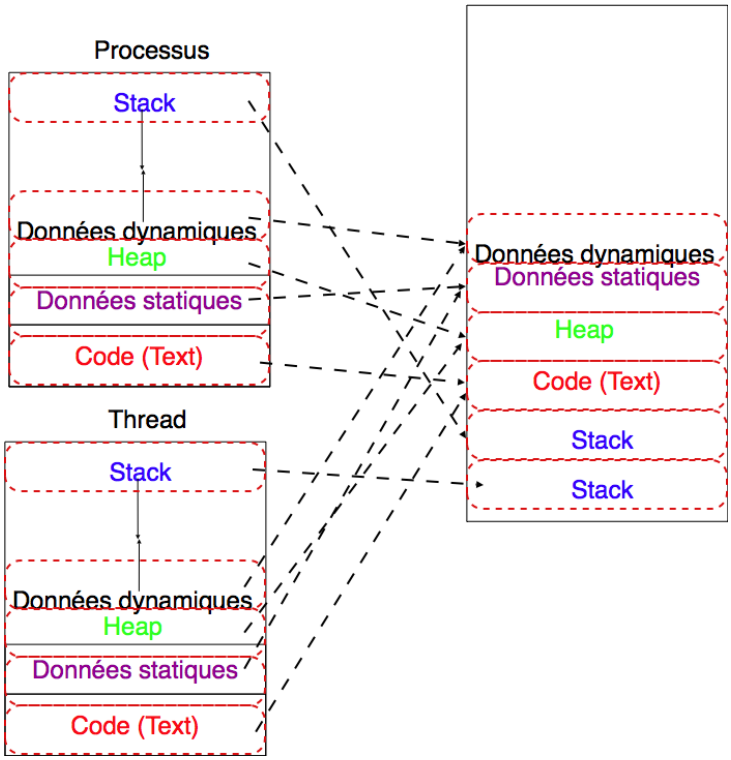


FIGURE 6.10 – Tables des pages après création d'un thread

partagée, considérons le cas de deux processus : *P1* et *P2* qui veulent pouvoir utiliser une page commune en mémoire. Pour cela, plusieurs interactions entre les processus et le système d'exploitation sont nécessaires comme nous allons le voir.

Avant de permettre à deux processus d'accéder à la même page en mémoire physique, il faut d'abord se poser la question de l'origine de cette page physique. Deux solutions sont possibles. La première est de prendre cette page parmi les pages qui appartiennent à l'un des processus, par exemple *P1*. Lorsque la page est partagée, le système d'exploitation peut modifier la table des pages du processus *P2* de façon à lui permettre d'y accéder. La seconde est que le noyau du système d'exploitation fournisse une nouvelle page qui pourra être partagée. Cette page "appartient" au noyau mais celui-ci la rend accessible aux processus *P1* et *P2* en modifiant leurs tables des pages. Linux utilise la seconde technique. Elle a l'avantage de permettre un meilleur contrôle par le système d'exploitation du partage de pages entre différents processus. De plus, lorsqu'une zone de mémoire partagée a été créée via le système d'exploitation, elle survit à la terminaison de ce processus. Une mémoire partagée créée par un processus peut donc être utilisée par d'autres processus.

Sous Linux, la mémoire partagée peut s'utiliser via les appels systèmes shmget(2), shmat(2) et shmdt(2). L'appel système shmget(2) permet de créer un segment de mémoire partagée. Le premier argument de shmget(2) est une clé qui identifie le segment de mémoire partagée. Cette clé est en pratique encodée sous la forme d'un entier qui identifie le segment de mémoire partagée. Elle sert d'identifiant du segment de mémoire partagée dans le noyau. Un processus doit connaître la clé qui identifie un segment de mémoire partagée pour pouvoir y accéder. Le deuxième argument de shmget(2) est la taille du segment. En pratique, celle-ci sera arrondie au multiple entier supérieur de la taille d'une page. Enfin, le troisième argument sont des drapeaux qui contrôlent la création du segment et les permissions qui y sont associées.

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

L'appel système shmget(2) retourne un entier qui identifie le segment de mémoire partagée à l'intérieur du processus si il réussit et `-1` sinon. Il peut être utilisée de deux façons. Un processus peut appeler shmget(2) pour créer un nouveau segment de mémoire partagée. Pour cela, il choisit une clé unique qui identifie ce segment et utilise le drapeau `IPC_CREAT`. Celui-ci peut être combiné avec les drapeaux qui sont supportés par l'appel système

open(2). Ainsi, le fragment de code ci-dessous permet de créer une page de mémoire partagée qui a `1252` comme identifiant et est accessible en lecture et en écriture par tous les processus qui appartiennent au même utilisateur ou au même groupe que le processus courant. Si cet appel à shmget(2) réussit, le segment de mémoire est initialisé à la valeur 0.

```
key_t key=1252;
int shm_id = shmget(key, 4096, IPC_CREAT | S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP );
if (shm_id == -1) {
  perror("shmget");
  exit(EXIT_FAILURE);
}
```

La fonction shmget(2) peut aussi être utilisée par un processus pour obtenir l'autorisation d'accéder à un segment de mémoire partagée qui a été créé par un autre processus. Dans ce cas, le drapeau `IPC_CREAT` n'est pas passé en argument.

Il est important de noter que si l'appel à shmget(2) réussit, cela indique que le processus dispose des permissions pour accéder au segment de mémoire partagée, mais à ce stade il n'est pas accessible depuis la table des pages du processus. Cette modification à la table des pages du processus se fait en utilisant shmat(2). Cet appel système permet d'attacher un segment de mémoire partagée à un processus. Il prend comme premier argument l'identifiant du segment de mémoire retourné par shmget(2). Le deuxième argument est un pointeur vers la zone mémoire via laquelle le segment doit être accessible dans l'espace d'adressage virtuel du processus. Généralement, c'est la valeur `NULL` qui est spécifiée comme second argument et le noyau choisit l'adresse à laquelle le segment de mémoire est attaché dans le processus. Il est aussi possible de spécifier une adresse dans l'espace d'adressage du processus. Le troisième argument permet, en utilisant le drapeau `SHM_RDONLY`, d'attacher le segment en lecture seule. shmat(2) retourne l'adresse à laquelle le segment a été attaché en cas de succès et `(void *) -1` en cas d'erreur.

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

void *shmat(int shmid, const void *shmaddr, int shmflg);

int shmdt(const void *shmaddr);
```

L'appel système shmdt(2) permet de détacher un segment de mémoire qui avait été attaché en utilisant shmat(2). L'argument passé à shmdt(2) doit être l'adresse d'un segment de mémoire attaché préalablement par shmat(2). Lorsqu'un processus se termine, tous les segments auxquels il était attaché sont détachés lors de l'appel à exit(2). Cela n'empêche pas un programme de détacher correctement tous les segments de mémoire qu'il utilise avant de se terminer.

Le fragment de code ci-dessous présente comment un segment de mémoire peut être attaché et détaché après avoir été créé avec shmget(2).

```
void * addr = shmat(shm_id, NULL, 0);
if (addr == (void *) -1) {
  perror("shmat");
  exit(EXIT_FAILURE);
}
// ...
if(shmdt(addr)==-1) {
  perror("shmdt");
  exit(EXIT_FAILURE);
}
```

Note : Attention aux pointeurs en mémoire partagée

Lorsque deux processus partagent le même segment de mémoire partagée, ils ont tous les deux accès directement à la mémoire. Il est ainsi possible de stocker dans cette mémoire un tableau de nombres ou de caractères. Chacun des processus pourra facilement accéder aux données stockées dans le tableau. Il faut cependant être vigilant lorsque l'on veut stocker une structure de données utilisant des pointeurs dans un segment de mémoire partagée. Considérons une liste simplement chaînée. Cette liste peut être implémentée en utilisant une structure contenant

la donnée stockée dans l'élément de la liste (par exemple un entier) et un pointeur vers l'élément suivant dans la liste (et `NULL` en fin de liste). Imaginons que les deux processus ont attaché le segment de mémoire destiné à contenir la liste avec l'appel shmat(2) présenté ci-dessus et que l'adresse retournée par shmat(2) est celle qui correspond au premier élément de la liste. Comme le système d'exploitation choisit l'adresse à laquelle le segment de mémoire partagée est stocké dans chaque processus, l'appel à shmat(2) retourne potentiellement une adresse différente dans les deux processus. Si ils peuvent tous les deux accéder au premier élément de la liste, il n'en sera pas de même pour le second élément. En effet, si cet élément a été créé par le premier processus, le pointeur contiendra l'adresse du second élément dans l'espace d'adressage virtuel du premier processus. Cette adresse ne correspondra en général pas à celle du second élément dans l'espace d'adressage du second processus. Pour cette raison, il est préférable de ne pas utiliser de pointeurs dans un segment de mémoire partagé.

Comme les segments de mémoire partagée sont gérés par le noyau du système d'exploitation, ils persistent après la terminaison du processus qui les a créé. C'est intéressant lorsque l'on veut utiliser des segments de mémoire partagée pour la communication entre plusieurs processus dont certains peuvent se crasher. Malheureusement, le nombre de segments de mémoire partagée qui peuvent être utilisés sur un système Unix est borné. Lorsque la limite fixée par la configuration du noyau est atteinte, il n'est plus possible de créer de nouveau segment de mémoire partagée. Sous Linux ces limites sont visibles dans les fichiers `/proc/sys/kernel/shmmni` (nombre maximum d'identifiants de segments de mémoire partagée) et `/proc/sys/kernel/shmall` (taille totale maximale de la mémoire partagée) ou via shmctl(2). Cet appel système permet de réaliser de nombreuses fonctions de contrôle de la mémoire partagée et notamment la destruction de segments de mémoire partagée qui ont été créés par shmget(2). shmctl(2) s'appuie sur les structures de données qui sont maintenues par le noyau pour les segments de mémoire partagée. Lorsqu'un segment de mémoire partagée est crée, le noyau lui associe une structure de type `shmid_ds`.

```
struct shmid_ds {
  struct ipc_perm shm_perm;    /* Propriétaire et permissions */
  size_t          shm_segsz;   /* Taille du segment (bytes) */
  time_t          shm_atime;   /* Instant de dernier attach */
  time_t          shm_dtime;   /* Instant de dernier detach */
  time_t          shm_ctime;   /* Instant de dernière modification */
  pid_t           shm_cpid;    /* PID du créateur */
  pid_t           shm_lpid;    /* PID du dernier `shmat(2)`_ / `shmdt(2)`_ */
  shmatt_t        shm_nattch;  /* Nombre de processus attachés */
};
```

Ce descripteur de segment de mémoire partagée, décrit dans shmctl(2) contient plusieurs informations utiles. Son premier élément est une structure qui reprend les informations sur le propriétaire et les permissions qui ont été définies ainsi que la taille du segment. Le descripteur de segment comprend ensuite les instants auxquels les dernières opérations shmat(2), shmdt(2) et la dernière modification au segment ont été faites. Le dernier élément contient le nombre de processus qui sont actuellement attachés au segment. L'appel système shmctl(2) prend trois arguments. Le premier est un identifiant de segment de mémoire partagée retourné par shmget(2). Le deuxième est une constante qui spécifie une commande. Nous utiliserons uniquement la commande `IPC_RMID` qui permet de retirer le segment de mémoire partagée dont l'identifiant est passé comme premier argument. Si il n'y a plus de processus attaché au segment de mémoire partagée, celui-ci est directement supprimé. Sinon, il est marqué de façon à ce que le noyau retire le segment dès que le dernier processus s'en détache. shmctl(2) retourne `0` en cas de succès et `-1` en cas d'échec.

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

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Le segment de mémoire partagée qui a été créé dans les exemples précédents peut être supprimé avec le fragment de code ci-dessous.

```
if (shmctl(shm_id, IPC_RMID, 0) != 0) {
   perror("shmctl");
   exit(EXIT_FAILURE);
}
```

En pratique, comme le noyau ne détruit un segment de mémoire partagée que lorsqu'il n'y a plus de processus qui y est attaché, il peut être utile de détruire le segment de mémoire partagée juste après avoir effectué l'appel shmat(2). C'est ce que l'on fera par exemple si un processus père utilise un segment de mémoire partagée pour communiquer avec son processus fils.

La mémoire partagée est utilisée non seulement pour permettre la communication entre processus, mais également avec les librairies partagées. Celles-ci sont chargées automatiquement lors de l'exécution d'un processus qui les utilise. Les instructions qui font partie de ces librairies partagées sont chargées dans la même zone mémoire que celle qui est utilisée pour la mémoire partagée. Sous Linux, cette zone mémoire est située entre le heap et le stack comme illustré dans la figure ci-dessous.

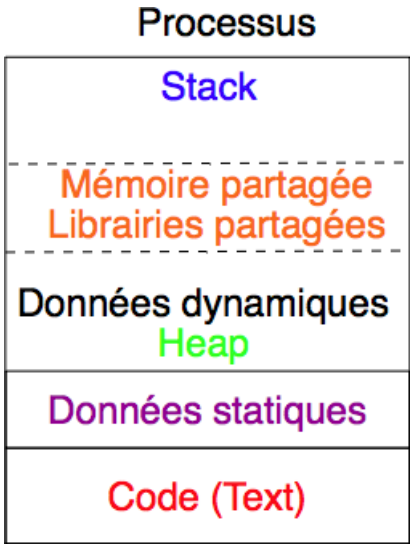


FIGURE 6.11 – Organisation en mémoire d'un processus

Lorsqu'il exécute un processus, le noyau maintient dans les structures de données qui sont relatives à ce processus la liste des segments de mémoire partagée et des librairies partagées qu'il utilise. Sous Linux, cette information est visible via le pseudo-système de fichiers `/proc`. Le fichier `/proc/PID/maps` représente de façon textuelle la table des segments de mémoire qui sont partagés dans le processus `PID`.

Un exemple d'un tel fichier *maps* est présenté ci-dessous. Il contient une carte de l'ensemble des pages qui appartiennent à un processus. Le fichier comprend six colonnes. La première est la zone de mémoire virtuelle. La seconde sont les bits de permission avec *r* pour la permission de lecture, *w* pour l'écriture et *x* pour l'exécution. Le dernier bit de permission est à la valeur *p* lorsque la page est en *copy-on-write* et *s* lorsqu'il s'agit d'un segment de mémoire partagé. Les trois dernières colonnes sont relatives au stockage des pages sur le disque.

```
00400000-00402000 r-xp 00000000 00:1a 49485798      /tmp/a.out
00602000-00603000 rw-p 00002000 00:1a 49485798      /tmp/a.out
3d3f200000-3d3f220000 r-xp 00000000 08:01 268543    /lib64/ld-2.12.so
3d3f41f000-3d3f420000 r--p 0001f000 08:01 268543    /lib64/ld-2.12.so
3d3f420000-3d3f421000 rw-p 00020000 08:01 268543    /lib64/ld-2.12.so
3d3f421000-3d3f422000 rw-p 00000000 00:00 0
3d3f600000-3d3f786000 r-xp 00000000 08:01 269510    /lib64/libc-2.12.so
3d3f786000-3d3f986000 ---p 00186000 08:01 269510    /lib64/libc-2.12.so
3d3f986000-3d3f98a000 r--p 00186000 08:01 269510    /lib64/libc-2.12.so
3d3f98a000-3d3f98b000 rw-p 0018a000 08:01 269510    /lib64/libc-2.12.so
3d3f98b000-3d3f990000 rw-p 00000000 00:00 0
3d3fa00000-3d3fa83000 r-xp 00000000 08:01 269516    /lib64/libm-2.12.so
3d3fa83000-3d3fc82000 ---p 00083000 08:01 269516    /lib64/libm-2.12.so
3d3fc82000-3d3fc83000 r--p 00082000 08:01 269516    /lib64/libm-2.12.so
3d3fc83000-3d3fc84000 rw-p 00083000 08:01 269516    /lib64/libm-2.12.so
7f7c57e42000-7f7c57e45000 rw-p 00000000 00:00 0
7f7c57e60000-7f7c57e61000 rw-s 00000000 00:04 66355276 /SYSV00000000
7f7c57e61000-7f7c57e63000 rw-p 00000000 00:00 0
7fffc479c000-7fffc47b1000 rw-p 00000000 00:00 0           [stack]
```

L'exemple ci-dessus présente la carte de mémoire d'un processus qui utilise trois librairies partagées. Le segment de mémoire partagée se trouve aux adresses virtuelles `7f7c57e60000-7f7c57e61000`. Il est accessible en lecture et en écriture.

6.1.8 Implémentation de fork(2)

La mémoire partagée joue un rôle clé dans l'exécution efficace des appels systèmes fork(2) et execve(2). Considérons d'abord fork(2). Cet appel est fondamental sur un système Unix. Au fil des années, les développeurs de Unix et de Linux ont cherché à optimiser ses performances. Une implémentation naïve de l'appel système fork(2) est de copier physiquement toutes les pages utilisées en mémoire RAM par le processus père. Ensuite, le noyau peut créer une table des pages pour le processus fils qui pointe vers les copies des pages du processus père. De cette façon, le processus père et le processus fils utilisent exactement les mêmes instructions. Ils peuvent donc poursuivre leur exécution à partir des mêmes données en mémoire. Mais chaque processus pourra faire les modifications qu'il souhaite aux données stockées en mémoire. Cette implémentation était utilisée par les premières versions de Unix, mais elle est peu efficace, notamment pour les processus qui consomment beaucoup de mémoire et le shell qui généralement exécute fork(2) et suivi par execve(2). Dans ce scénario, copier l'entièreté de la mémoire du processus père est un gaspillage de ressources.

La mémoire virtuelle permet d'optimiser l'appel système fork(2) et de le rendre nettement plus rapide. Lors de la création d'un processus fils, le noyau du système d'exploitation commence par créer une table des pages pour le processus fils. En initialisant cette table avec les mêmes entrées que celles du processus père, le noyau permet aux deux processus d'accéder aux mêmes instructions et aux mêmes données. Pour les instructions se trouvant dans le segment code et dont les entrées de la table des pages sont généralement en *read-only*, cette solution fonctionne correctement. Le processus père et le processus fils peuvent exécuter exactement les mêmes instructions tout en n'utilisant qu'une seule copie de ces instructions en mémoire.

Malheureusement, cette solution ne fonctionne plus pour les pages contenant les données globales, le stack et le heap. En effet, ces pages doivent pouvoir être modifiées de façon indépendante par le processus père et le processus fils. C'est notamment le cas pour la zone mémoire qui contient la valeur de retour de l'appel système fork(2). Par définition, cette zone mémoire doit contenir une valeur différente dans le processus père et le processus fils. Pour éviter ce problème, le noyau pourrait copier physiquement les pages contenant les variables globales, le heap et le stack. Cela permettrait, notamment dans le cas de l'exécution de fork(2) par le shell d'améliorer les performances de fork(2) sans pour autant compromettre la sémantique de cet appel système. Il existe cependant une alternative à cette copie physique. Il s'agit de la technique du *copy-on-write*.

Sur un système qui utilise *copy-on-write*, l'appel système fork(2) est implémenté comme suit. Lors de l'exécution de fork(2), le noyau copie toutes les entrées de la table des pages du processus père vers la table des pages du processus fils. Ce faisant, le noyau modifie également les permissions de toutes les pages utilisées par le processus père. Les pages correspondant au segment de code sont toutes marquées en lecture seule. Les pages correspondant aux données globales, heap et stack sont marquées avec un statut spécial (*copy-on-write*). Celui-ci autorise les accès en lecture à la page sans restriction. Si un processus tente un accès en écriture sur une de ces pages, le *MMU* interrompt l'exécution du processus et force l'exécution d'une routine d'interruption du noyau. Celle-ci analyse la tentative d'accès à la mémoire qui a échoué. Si la page était en lecture seule (par exemple une page du segment de code), un signal `SIGSEGV` est envoyé au processus concerné. Si par contre la page était marquée *copy-on-write*, alors le noyau alloue une nouvelle page physique et y recopie la page où la tentative d'accès a eu lieu. La table des pages du processus qui a fait la tentative d'accès est modifiée pour pointer vers la nouvelle page avec une permission en lecture et en écriture. La permission de l'entrée de la table des pages de l'autre processus est également modifiée de façon à autoriser les écritures et les lectures. Les deux processus disposent donc maintenant d'une copie différente de cette page et ils peuvent la modifier à leur guise. Cette technique de *copy-on-write* permet de ne copier que les pages qui sont modifiées par le processus père ou le processus fils. C'est un gain de temps appréciable par rapport à la copie complète de toutes les pages.

Dans le pseudo fichier `/proc/PID/maps` présenté avant, le bit *p* indique que les pages correspondantes sont en *copy-on-write*.

6.1.9 Interactions entre les processus et la mémoire

La mémoire virtuelle est gérée par le système d'exploitation. La plupart des processus supportent très bien d'avoir certaines de leurs pages qui sont sauvegardées sur disque lorsque la mémoire est saturée. Cependant, dans certains cas très particuliers, il peut être intéressant pour un processus de contrôler quelles sont ses pages qui restent en mémoire et quelles sont celles qui sont stockées sur le disque dur. Linux fournit plusieurs appels systèmes qui permettent à un processus de monitorer et éventuellement d'influencer la stratégie de remplacement des pages.

```
#include <unistd.h>
#include <sys/mman.h>
int mincore(void *addr, size_t length, unsigned char *vec);
```

L'appel système mincore(2) permet à un processus d'obtenir de l'information sur certaines de ses pages. Il prend comme premier argument une adresse, qui doit être un multiple de la taille des pages. Le deuxième est la longueur de la zone mémoire pour laquelle le processus demande de l'information. Le dernier argument est un pointeur vers une chaîne de caractères qui doit contenir autant de caractères que de pages dans la zone de mémoire analysée. Cette chaîne de caractères contiendra des drapeaux pour chaque page de la zone mémoire concernée. Les principaux sont :

- `MINCORE_INCORE` indique que la page est en mémoire RAM
- `MINCORE_REFERENCED` indique que la page a été référencée
- `MINCORE_MODIFIED` indique que la page a été modifiée

Un exemple d'utilisation de mincore(2) est repris dans le programme `/MemoireVirtuelle/src/mincore.c` ci-dessous.

```
#dfine _BSD_SOURCE

#define SIZE 10*4096

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

  char *mincore_vec;
  size_t page_size = getpagesize();
  size_t page_index;
  char *mem;

  mem=(char *)malloc(SIZE*sizeof(char));
  if(mem==NULL) {
    perror("malloc");
    exit(EXIT_FAILURE);
  }
  for(int i=0;i<SIZE/2;i++) {
    *(mem+i)='A';
  }

  mincore_vec = calloc(sizeof(char), SIZE/page_size);
  if(mincore_vec==NULL) {
    perror("calloc");
    exit(EXIT_FAILURE);
  }
  if(mincore(mem, SIZE, mincore_vec)!=0) {
    perror("mincore");
    exit(EXIT_FAILURE);
  }

  for (page_index = 0; page_index < SIZE/page_size; page_index++) {
    printf("%lu :", (unsigned long)page_index);
    if (mincore_vec[page_index]&MINCORE_INCORE) {
      printf(" incore");
    }
    if (mincore_vec[page_index]&MINCORE_REFERENCED) {
      printf(" referenced_by_us");
    }
```

```
    if (mincore_vec[page_index]&MINCORE_MODIFIED) {
      printf(" modified_by_us");
    }
    printf("\n");
  }
  free(mincore_vec);
  free(mem);
  return (EXIT_SUCCESS);
}
```

Certains processus doivent pouvoir contrôler la façon dont leurs pages sont stockées en mémoire RAM ou sur disque. C'est le cas notamment des processus qui manipulent des clés cryptographiques. Pour des raisons de sécurité, il est préférable que ces clés ne soient pas sauvegardées sur le disque dur. Pour des raisons de performances, certains processus préfèrent également éviter que leurs pages ne soient stockées sur disque dur. Linux comprend plusieurs appels systèmes qui permettent à un processus d'influencer la stratégie de remplacement des pages du noyau.

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

int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);

int mlockall(int flags);
int munlockall(void);
```

L'appel système mlock(2) permet de forcer un ensemble de pages à rester en mémoire RAM tandis que munlock(2) fait l'inverse. L'appel système mlockall(2) quant à lui permet à un processus de demander que tout l'espace d'adressage qu'il utilise reste en permanence en mémoire RAM. mlockall(2) prend comme argument des drapeaux. Actuellement deux drapeaux sont supportés. `MCL_CURRENT` indique que toutes les pages actuellement utilisées par le processus doivent rester en mémoire. `MCL_FUTURE` indique que toutes les pages qui seront utilisées par le processus devront être en mémoire RAM au fur et à mesure de leur allocation. L'appel système madvise(2) permet également à un processus de donner des indications sur la façon dont il utilisera ses pages dans le futur.

Ces appels systèmes doivent être utilisés avec précautions. En forçant certaines de ses pages à rester en mémoire, un processus perturbe le bon fonctionnement de la mémoire virtuelle, ce qui risque de perturber les performances globales du système. En pratique, en dehors d'applications cryptographiques et de processus avec des exigences temps réel qui sortent du cadre de ce cours, il n'y a pas d'intérêt à utiliser ces appels système.

L'utilisation de la mémoire influence fortement les performances des processus. Plusieurs utilitaires sous Linux permettent de mesurer la charge d'un système. Certains offrent une interface graphique. D'autres, comme top(1) ou vmstat(8) s'utilisent directement depuis un terminal. La commande vmstat(8) permet de suivre l'évolution de la charge du système et de la mémoire virtuelle.

```
$ vmstat 5
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
3  0 662260 259360  49044 583808    0    0     1     1    1    2  5  0 95  0  0
3  0 662260 259352  49044 583808    0    0     0     0 6018 4182 61  0 39  0  0
7  1 662260 254856  49044 587328    0    0     0     0 6246 4309 61  0 38  2  0
4  0 662260 221668  49044 608652    0    0     0     0 9731 5520 62  1  9 28  0
3  0 662260 260368  49044 583836    0    0     0     2 8291 4868 70  1 12 17  0
4  0 662260 260548  49044 583836    0    0     0     0 6042 4174 61  0 39  0  0
3  0 662260 260720  49044 583836    0    0     0     0 6003 4242 61  0 39  0  0
```

Lorsqu'elle est utilisée, la commande vmstat(8) retourne de l'information sur l'état du système. Dans l'exemple ci-dessus, elle affiche cet état toutes les 5 secondes. La sortie de vmstat(8) comprend plusieurs informations utiles.

— la colonne `procs` reprend le nombre de processus qui sont dans l'état *Running* ou *Blocked*.
— la colonne `memory` reprend l'information relative à l'utilisation de la mémoire. La colonne *swpd* est la quantité de mémoire virtuelle utilisée. La colonne *free* est la quantité de mémoire RAM qui est actuellement libre. Les colonnes `buff` et `cache` sont relatives aux buffers et à la cache qui sont utilisés par le noyau.

— la colonne `swap` reprend la quantité de mémoire qui a été transférée sur le disque (*so*) ou lue du disque (*si*)
— la colonne `io` reprend le nombre de blocs lus du disque (*bi*) et écrits (*bo*)
— la colonne `system` reprend le nombre d'interruptions (*in*) et de changements de contexte (*cs*)
— la dernière colonne (`cpu`) fournit des statistiques sur l'utilisation du cpu

Enfin, notons que l'appel système getrusage(2) peut être utilisé par un processus pour obtenir de l'information sur son utilisation des ressources du système et notamment les opérations de transfert de pages entre le mémoire RAM et les dispositifs de stockage.

6.1.10 execve(2) et la mémoire virtuelle

Pour terminer ce survol de la mémoire virtuelle, il est intéressant de revenir sur le fonctionnement de l'appel système execve(2). Celui-ci permet de remplacer l'image mémoire du processus en cours d'exécution par un exécutable chargé depuis le disque. L'appel système execve(2) utilise fortement la mémoire virtuelle. Plutôt que de copier l'entièreté de l'exécutable en mémoire, il se contente de créer les entrées correspondants au segment de code et aux variables globales dans la table des pages du processus. Ces entrées pointent vers le fichier exécutable qui est sauvegardé sur le disque. En pratique, celui-ci est découpé en deux parties. La première contient les instructions. Celles-ci sont mappées sous forme de pages sur lesquelles le processus a les permissions d'exécution et de lecture. La seconde partie du fichier correspond à la zone des chaînes de caractères et des variables globales. Celle-ci est mappée dans des pages qui sont accessibles en lecture et en écriture. En pratique, la technique du *copy-on-write* est utilisée pour ne créer une copie spécifique au processus que si celui-ci modifie certaines données en mémoire. Ensuite, les pages relatives au *heap* et au *stack* peuvent être créées.

La sortie ci-dessous présente le contenu de `/proc/PID/maps` lors de l'exécution du programme `/MemoireVirtuelle/src/cp2.c` présenté plus haut. Celui-ci utilise deux fichiers mappés en mémoire avec mmap(2). Ceux-ci apparaissent dans la table des pages du processus, tout comme l'exécutable. Au niveau des permissions, on remarquera que le fichier source est mappé avec des pages en lecture seule tandis que le fichier destination est mappé en lecture/écriture.

```
$ cat /proc/29039/maps
00400000-004ab000 r-xp 00000000 00:19 49479785                           /etinfo/users2/obo/sinf1
006ab000-006ac000 rw-p 000ab000 00:19 49479785                           /etinfo/users2/obo/sinf1
006ac000-006af000 rw-p 00000000 00:00 0
00bc7000-00bea000 rw-p 00000000 00:00 0                                  [heap]
7fc43d5e4000-7fc43d6a4000 rw-s 00000000 00:19 51380745                   /etinfo/users2/obo/sinf1
7fc43d6a4000-7fc43d764000 r--s 00000000 00:19 49479785                   /etinfo/users2/obo/sinf1
7fff5b912000-7fff5b927000 rw-p 00000000 00:00 0                          [stack]
7fff5b9ff000-7fff5ba00000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

6.2 Annexes

6.2.1 Bibliographie

6.2.2 Glossaire

CPU Central Processing Unit

C Langage de programmation permettant d'interagir facilement avec le matériel.

RISC Reduced Instruction Set Computer

CISC Complex Instruction Set Computer

x86 Famille de microprocesseurs développée par intel. Le 8086 est le premier processeur de cette famille. Ses successeurs (286, 386, Pentium, Centrino, Xeon, ...) sont restés compatibles avec lui tout en introduisant chacun de nouvelles instructions et de nouvelles fonctionnalités. Aujourd'hui, plusieurs fabricants développent des processeurs qui supportent le même langage machine que les processeurs de cette famille.

Unix Système d'exploitation développé initialement par AT&T Bell Labs.

gcc Compilateur pour la langage C développé par un groupe de volontaires qui est diffusé depuis http://gcc.gnu.org gcc est utilisé dans plusieurs systèmes d'exploitation de type Unix, comme MacOS, Linux ou FreeBSD. Il existe d'autres compilateurs C. Une liste non-exhaustive est maintenue sur http://en.wikipedia.org/wiki/List_of_compilers#C_compilers

llvm Ensemble de compilateurs pour différents langages de programmation et différents processeurs développé par un groupe de volontaire. llvm est distrubé depuis http://llvm.org/

cpp, préprocesseur Le préprocesseur C est un programme de manipulation de texte sur base de macros qui est utilisé avec le compilateur. Le préprocesseur de *gcc* est http://gcc.gnu.org/onlinedocs/cpp/

microprocesseur, processeur à compléter

CPU Central Processing Unit. Voir *microprocesseur*

stdin Entrée standard sur un système Unix (par défaut le clavier)

stdout Sortie standard sur un système Unix (par défaut l'écran)

stderr Sortie d'erreur standard sur un système Unix (par défaut l'écran)

X11 à compléter

Gnome à compléter

CDE à compléter

shell Interpréteur de commandes sur un système Unix. bash(1) est l'interpréteur de commandes le plus utilisé de nos jours.

bit Plus petite unité d'information. Par convention, un bit peut prendre les valeurs `0` et `1`.

nibble Un bloc de quatre bits consécutifs.

byte, octet Un bloc de huit bits consécutifs.

BSD Unix Variante de Unix développée à l'Université de Californie à Berkeley.

FreeBSD Variante de BSD Unix disponible depuis http://www.freebsd.org

OpenBSD Variante de BSD Unix disponible depuis http://www.openbsd.org

MacOS Système d'exploitation développé par Apple Inc. comprenant de nombreux composantes provenant de *FreeBSD*

Minix Famille de noyaux de systèmes d'exploitation inspiré de *Unix* développée notamment par *Andrew Tanenbaum*. Voir http://www.minix3.org pour la dernière version de Minix.

Linux Noyau de système d'exploitation compatible Unix développé initialement par Linus Torvalds.

Solaris Système d'exploitation compatible Unix développé par Sun Microsystems et repris par Oracle. La version open-source, OpenSolaris, est disponible depuis http://www.opensolaris.org

Application Programming Interface, API Un API est généralement un ensemble de fonctions et de structures de données qui constitue l'interface entre deux composants logiciels qui doivent collaborer. Par exemple, l'API du noyau d'un système Unix est composée de ses appels systèmes. Ceux-ci sont décrits dans la section 2 des pages de manuel (voir intro(2)).

GNU is not Unix, GNU GNU est un projet open-source de la Free Software Foundation qui a permis le développement d'un grand nombre d'utilitaires utilisés par les systèmes d'exploitation de la famille Unix actuellement.

GNU/Linux Nom générique donné à un système d'exploitation utilisant les utilitaires *GNU* notamment et le noyau *Linux* .

Andrew Tanenbaum Andrew Tanenbaum est professeur à la VU d'Amsterdam.

Linus Torvalds Linus Torvalds est le créateur et le mainteneur principal du noyau *Linux*.

Aqua Aqua est une interface graphique spécifique à *MacOS*.

pipe Mécanisme de redirection des entrées-sorties permettant de relier la sortie standard d'un programme à l'entrée standard d'un autre pour créer des pipelines de traitement.

assembleur Programme permettant de convertir un programme écrit en langage d'assemblage dans le langage machine correspondant à un processeur donné.

warning Message d'avertissement émis par un compilateur C. Un *warning* n'empêche pas la compilation et la génération du code objet. Cependant, la plupart des warnings indiquent un problème dans le programme compilé et il est nettement préférable de les supprimer du code.

bit de poids fort Par convention, le bit le plus à gauche d'une séquence de n bits.

bit de poids faible Par convention, bit le plus à droite d'une séquence de n bits.

simple précision Représentation de nombre réels en virgule flottante (type `float` en C). La norme IEEE754 définit le format de ces nombres sur 32 bits.

double précision Représentation de nombre réels en virgule flottante (type `double` en C). La norme IEEE754 définit le format de ces nombres sur 64 bits.

buffer overflow Problème à compléter

garbage collector Algorithme permettant de libérer la mémoire qui n'est plus utilisée notamment dans des langages tels que Java

pointeur à compléter

adresse à compléter

C99 Standard international définissant le langage C [C99]

fichier header à compléter

segmentation fault Erreur à l'exécution à compléter

NOT, négation Opération binaire logique.

AND, conjonction logique Opération binaire logique.

OR, disjonction logique Opération binaire logique.

XOR, ou exclusif Opération binaire logique.

libc Librairie C standard. Contient de nombreuses fonctions utilisables par les programmes écrits en langage C et décrites dans la troisième section des pages de manuel. Linux utilise la librairie GNU glibc qui contient de nombreuses extensions par rapport à la libraire standard.

FSF Free Software Foundation, http://www.fsf.org

buffer overflow à compléter

portée à compléter

portée locale à compléter

portée globale à compléter

debugger à compléter

text, segment text à compléter

segment des données initialisées à compléter

segment des données non-initialisées à compléter

heap, tas à compléter

stack, pile à compléter

etext à compléter

memory leak à compléter

processus Ensemble cohérent d'instructions utilisant une partie de la mémoire, initié par le système d'exploitation et exécuté sur un des processeurs du système. Le système d'exploitation libère les ressources qui lui sont allouées à la fin de son exécution.

pid, process identifier identifiant de processus. Sous Unix, chaque processus est identifié par un entier unique. Cet identifiant sert de clé d'accès à la *table des processus*. Voir getpid(2) pour récupérer l'identifiant du processus courant.

table des processus Table contenant les identifiants (*pid*) de tous les processus qui s'exécutent à ce moment sur un système Unix. Outre les identifiants, cette table contient de nombreuses informations relatives à chaque *processus*. Voir également */proc*

/proc Sous Linux, représentation de l'information stockée dans la *table des processus* sous la forme d'une arborescence directement accessible via les commandes du *shell*. Voir proc(5)

signal mécanisme permettant la communication entre processus. Utilisé notamment pour arrêter un processus via la commande kill(1)

von Neumann Un des inventaires des premiers ordinateurs. A défini l'architecture de base des premiers ordinateurs qui est maintenant connue comme le modèle de von Neumann [Krakowiak2011]

mémoire à compléter

SRAM, static RAM Un des deux principaux types de mémoire. Dans une SRAM, l'information est mémorisée comme la présence ou l'absence d'un courant électrique. Les mémoires SRAM sont généralement assez rapides mais de faible capacité. Elles sont souvent utilisées pour construire des mémoires caches.

DRAM, dynamic RAM Un des deux principaux types de mémoire. Dans une DRAM, l'information est mémorisée comme la présence ou l'absence de charge dans un minuscule condensateur. Les mémoires DRAM sont plus lentes que les *SRAM* mais ont une plus grande capacité.

RAM, Random Access Memory Mémoire à accès aléatoire. Mémoire permettant au processeur d'accéder à n'importe quelle donnée en connaissant son adresse. Voir *DRAM* et *SRAM*.

registre Unité de mémoire intégrée au processeur. Les registres sont utilisés comme source ou destination pour la plupart des opérations effectuées par un processeur.

hiérarchie de mémoire Ensemble des mémoires utilisées sur un ordinateur. Depuis les registres jusqu'à la mémoire virtuelle en passant par la mémoire centrale et les mémoires caches.

mémoire cache Mémoire rapide de faible capacité. La mémoire cache peut stocker des données provenant de mémoires de plus grande capacité mais qui sont plus lentes, et exploite le *principe de localité* en stockant de manière transparente les instructions et les données les plus récemment utilisées. Elle fait office d'interface entre le processeur et la mémoire principale et toutes les demandes d'accès à la mémoire principale passent par la mémoire cache, ce qui permet d'améliorer les performances de nombreux systèmes informatiques.

principe de localité Voir *localité spatiale* et *localité temporelle*.

localité spatiale à compléter

localité temporelle à compléter

lignes de cache à compléter

write through Technique d'écriture dans les mémoires caches. Toute écriture est faite simultanément en mémoire cache et en mémoire principale. Cela garantit la cohérence entre les deux mémoires mais réduit les performances.

write back Technique d'écriture dans les mémoires caches. Toute écriture est faite en mémoire cache. La mémoire principale n'est mise à jour que lorsque la donnée modifiée doit être retirée de la cache. Cette technique permet d'avoir de meilleures performances que *write through* mais il faut faire parfois attention aux problèmes qui pourraient survenir sachant que la mémoire cache et la mémoire principale ne contiennent pas toujours exactement la même information.

eip, pc, compteur de programme, instruction pointer Registre spécial du processeur qui contient en permanence l'adresse de l'instruction en cours d'exécution. Le contenu de ce registre est incrémenté après chaque instruction et modifié par les instructions de saut.

mode d'adressage à compléter

accumulateur Registre utilisé dans les premiers processeurs comme destination pour la plupart des opérations arithmétiques et logiques. Sur l'architecture [IA32], le registre `%eax` est le successeur de cet accumulateur.

bus à compléter

ligne de cache à compléter. Voir notamment [McKenney2005] et [Drepper2007]

write-back à compléter

program counter à compléter

makefile à compléter

fichier à compléter

fichier objet à compléter

linker à compléter

errno à compléter

loi de Moore à compléter

kHz à compléter

MHz à compléter

GHz à compléter

MIPS Million d'instructions par seconde
benchmark à compléter
multi-coeurs à compléter
multi-threadé à compléter
section critique à compléter
exclusion mutuelle à compléter
sureté, safety à compléter
liveness, vivacité à compléter
multitâche, multitasking à compléter
contexte à compléter
changement de contexte à compléter
interruption à compléter
scheduler à compléter
round-robin à compléter
livelock à compléter
opération atomique à compléter
deadlock à compléter
mutex à compléter
problème des philosophes à compléter
appel système à compléter
appel système bloquant à compléter
sémaphore à compléter
problèmes des readers-writers à compléter
inode à compléter
segment de données à compléter
problème des readers-writers à compléter
thread-safe à compléter
loi de Amdahl à compléter
static library, librairie statique à compléter
shared library, librairie dynamique, librairie partagée à compléter
kernel à compléter
mode utilisateur à compléter
mode protégé à compléter
processus père à compléter
processus fils à compléter
processus orphelin à compléter
processus zombie à compléter
filesystem, système de fichiers à compléter
descripteur de fichier à compléter
répertoire à compléter
secteur à compléter
répertoire courant à compléter
offset pointer à compléter
little endian à compléter
big endian à compléter
lien symbolique à compléter

lock à compléter

advisory lock, advisory locking à compléter

mandatory lock, mandatory locking à compléter

open file object à compléter

sémaphore nommé à compléter

appel système lent à compléter

handler à compléter

signal synchrone à compléter

signal asynchrone à compléter

interpréteur à compléter

MMU, Memory Management Unit à compléter

adresse virtuelle à compléter

mémoire virtuelle à compléter

SSD, Solid State Drive Système de stockage de données s'appuyant uniquement sur de la mémoire flash.

stratégie de remplacement de pages à compléter

page à compléter

table des pages à compléter

bit de validité à compléter

TLB, Translation Lookaside Buffer à compléter

Mémoire partagée à compléter

copy-on-write à compléter

adresse physique à compléter

page fault, défaut de page à compléter

file FIFO De "First In, First Out". Le premier élement à entrer dans la file sera le premier à en sortir. (!= LIFO, "Last In First Out")

dirty bit, bit de modification à compléter

reference bit, bit de référence à compléter

swapping à compléter

pagination à compléter

stdio à compléter

fifo à compléter

gnuth à compléter

partition de swap à compléter

stratégie de remplacement de pages à compléter

2^64 à compléter

root à compléter

userid à compléter

[ABS] Cooper, M., *Advanced Bash-Scripting Guide*, 2011, http://tldp.org/LDP/abs/html/

[AdelsteinLubanovic2007] Adelstein, T., Lubanovic, B., *Linux System Administration*, OReilly, 2007, http://books.google.be/books?id=-jYe2k1p5tIC

[Alagarsamy2003] Alagarsamy, K., *Some myths about famous mutual exclusion algorithms*. SIGACT News 34, 3 (September 2003), 94-103. http://doi.acm.org/10.1145/945526.945527

[Amdahl1967] Amdahl, G., *Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities*, Proc. Am. Federation of Information Processing Societies Conf., AFIPS Press, 1967, pp. 483-485, http://dx.doi.org/10.1145/1465482.1465560

[Bashar1997] Bashar, N., *Ariane 5 : Who Dunnit ?*, IEEE Software 14(3) : 15–16. May 1997. doi :10.1109/MS.1997.589224.

[BovetCesati2005] Bovet, D., Cesati, M, *Understanding the Linux Kernel, Third Edition*, O'Reilly, 2005, http://my.safaribooksonline.com/book/operating-systems-and-server-administration/linux/0596005652

[BryantOHallaron2011] Bryant, R. and O'Hallaron, D., *Computer Systems : A programmer's perspective*, Second Edition, Pearson, 2011, http://www.amazon.com/Computer-Systems-Programmers-Perspective-2nd/dp/0136108040/ref=sr_1_1?s=books&ie=UTF8&qid=1329058781&sr=1-1

[C99] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

[Card+1994] Card, R., Ts'o, T., Tweedie, S, *Design and implementation of the second extended filesystem*. Proceedings of the First Dutch International Symposium on Linux. ISBN 90-367-0385-9. http://web.mit.edu/tytso/www/linux/ext2intro.html

[Cooper2011] Cooper, M., *Advanced Bash-Scripting Guide*, http://tldp.org/LDP/abs/html/, 2011

[Courtois+1971] Courtois, P., Heymans, F. and Parnas, D., *Concurrent control with "readers" and "writers"*. Commun. ACM 14, 10 (October 1971), 667-668. http://doi.acm.org/10.1145/362759.362813

[CPP] C preprocessor manual, http://gcc.gnu.org/onlinedocs/cpp/

[CSyntax] Wikipedia, C syntax, http://en.wikipedia.org/wiki/C_syntax

[Dijkstra1965b] Dijkstra, E., *Cooperating sequential processes*, 1965, http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01x

[Dijkstra1965] Dijkstra, E., *Solution of a problem in concurrent programming control*. Commun. ACM 8, 9 (September 1965), 569 http://doi.acm.org/10.1145/365559.365617

[Dijkstra1968] Dijkstra, E., *Go To Statement Considered Harmful*, Communications of the ACM, 11, March 1968, http://www.cs.utexas.edu/~EWD/transcriptions/EWD02xx/EWD215.html Voir aussi [Tribble2005]

[Downey2008] Downey, A., *The Little Book of Semaphores*, Second Edition, Green Tea Press, 2008,

[Drepper2007] Drepper, U., *What every programmer should know about memory*, 2007, http://www.akkadia.org/drepper/cpumemory.pdf

[DrepperMolnar2005] Drepper, U., Molnar, I., *The Native POSIX Thread Library for Linux*, http://www.akkadia.org/drepper/nptl-design.pdf

[Goldberg1991] Goldberg, D., *What every computer scientist should know about floating-point arithmetic*. ACM Comput. Surv. 23, 1 (March 1991), 5-48. http://doi.acm.org/10.1145/103162.103163 ou http://www.validlab.com/goldberg/paper.pdf

[Goldberg+1996] Goldberg, I., Wagner, D., *Randomness and the Netscape Browser*, Dr. Dobb's Journal, January 1996, http://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html

[Gove2011] Gove, D., *Multicore Application Programming for Windows, Linux and Oracle Solaris*, Addison-Wesley, 2011, http://books.google.be/books?id=NF-C2ZQZXekC

[GNUMake] http://www.gnu.org/software/make/manual/make.html

[GNUPTH] Engelschall, R., *GNU Portable Threads*, http://www.gnu.org/software/pth/

[Graham+1982] Graham, S., Kessler, P. and Mckusick, M., *Gprof : A call graph execution profiler*. SIGPLAN Not. 17, 6 (June 1982), 120-126. http://doi.acm.org/10.1145/872726.806987

[HennessyPatterson] Hennessy, J. and Patterson, D., *Computer Architecture : A Quantitative Approach*, Morgan Kauffmann, http://books.google.be/books?id=gQ-fSqbLfFoC

[Honeyford2006] Honeyford, M., *Speed your code with the GNU profiler*, http://www.ibm.com/developerworks/library/l-gnuprof.html

[HP] HP, *Memory technology evolution : an overview of system memory technologies*, http://h20000.www2.hp.com/bc/docs/support/SupportManual/c00256987/c00256987.pdf

[Hyde2010] Hyde, R., *The Art of Assembly Language*, 2nd edition, No Starch Press, http://webster.cs.ucr.edu/AoA/Linux/HTML/AoATOC.html

[IA32] intel, *Intel® 64 and IA-32 Architectures : Software Developer's Manual*, Combined Volumes : 1, 2A, 2B, 2C, 3A, 3B and 3C, December 2011, http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[Kamp2011] Kamp, P., *The Most Expensive One-byte Mistake*, ACM Queue, July 2011, http://queue.acm.org/detail.cfm?id=2010365

[Kerrisk2010] Kerrisk, M., *The Linux Programming Interface*, No Starch Press, 2010, http://my.safaribooksonline.com/book/programming/linux/9781593272203

[Kernighan] Kernighan, B., *Programming in C - A Tutorial*, http://cm.bell-labs.com/cm/cs/who/dmr/ctut.pdf

[KernighanRitchie1998] Kernighan, B., and Ritchie, D., *The C programming language, second edition*, Addison Wesley, 1998, http://cm.bell-labs.com/cm/cs/cbook/

[King2008] King, K., *C programming : a modern approach*, W. W. Norton & company, 2008

[Krakowiak2011] Krakowiak, S., *Le modele d'architecture de von Neumann*, http://interstices.info/le-modele-darchitecture-de-von-neumann

[Leroy] Leroy, X., *The LinuxThreads library*, http://pauillac.inria.fr/~xleroy/linuxthreads/

[McKenney2005] McKenney, P., *Memory Ordering in Modern Microprocessors, Part I*, Linux Journal, August 2005, http://www.linuxjournal.com/article/8211

[Mecklenburg+2004] Mechklenburg, R., Mecklenburg, R. W., Oram, A., *Managing projects with GNU make*, O'Reilly, 2004, http://books.google.be/books?id=rL4GthWj9kcC

[Mitchell+2001] Mitchell, M., Oldham, J. and Samuel, A., *Advanced Linux Programming*, New Riders Publishing, ISBN 0-7357-1043-0, June 2001, http://www.advancedlinuxprogramming.com/

[MorrisThomson1979] Morris, R. and Thompson, K. *Password security : a case history*. Commun. ACM 22, 11 (November 1979), 594-597. http://doi.acm.org/10.1145/359168.359172

[Nemeth+2010] Nemeth, E., Hein, T., Snyder, G., Whaley, B., *Unix and Linux System Administration Handbook*, Prentice Hall, 2010, http://books.google.be/books?id=rgFIAnLjb1wC

[Peterson1981] Peterson, G., *Myths about the mutual exclusion problem*, Inform. Process. Lett. 12 (3) (1981) 115-116

[Stallings2011] Stallings, W., *Operating Systems : Internals and Design Principles*, Prentice Hall, 2011, http://williamstallings.com/OperatingSystems/index.html

[StevensRago2008] Stevens, R., and Rago, S., *Advanced Programming in the UNIX Environment*, Addison-Wesley, 2008, http://books.google.be/books?id=wHI8PgAACAAJ

[Stokes2008] Stokes, J., *Classic.Ars : Understanding Moore's Law*, http://arstechnica.com/hardware/news/2008/09/moore.ars

[Tanenbaum+2009] Tanenbaum, A., Woodhull, A., *Operating systems : design and implementation*, Prentice Hall, 2009

[Toomey2011] Toomey, W., *The Strange Birth and Long Life of Unix*, IEEE Spectrum, December 2011, http://spectrum.ieee.org/computing/software/the-strange-birth-and-long-life-of-unix

[Tribble2005] Tribble, D., *Go To Statement Considered Harmful : A Retrospective*, 2005, http://david.tribble.com/text/goto.html

[Walls2006] Walls, D., *How to Use the restrict Qualifier in C*. Sun Microsystems, 2006, http://developers.sun.com/solaris/articles/cc_restrict.html

Symbols

/proc, **205**
_exit, 134
2^64, **208**

A

accumulateur, **206**
adresse, **205**
adresse physique, **208**
adresse virtuelle, **208**
advisory lock, **208**
advisory locking, **208**
AND, **205**
Andrew Tanenbaum, **204**
API, **204**
appel système, **207**
appel système bloquant, **207**
appel système lent, **208**
Application Programming Interface, **204**
Aqua, **204**
assembleur, **204**
atexit, 133

B

benchmark, **207**
big endian, **207**
bit, **204**
bit de modification, **208**
bit de poids faible, **205**
bit de poids fort, **205**
bit de référence, **208**
bit de validité, **208**
BSD Unix, **204**
buffer overflow, **205**
bus, **206**
byte, **204**

C

C, **203**
C99, **205**
CDE, **204**
changement de contexte, **207**
CISC, **203**
compteur de programme, **206**
conjonction logique, **205**
contexte, **207**
copy-on-write, **208**
cpp, **204**
CPU, **203**, **204**

D

défaut de page, **208**
deadlock, **207**
debugger, **205**
descripteur de fichier, **207**
dirty bit, **208**
disjonction logique, **205**
double précision, **205**
DRAM, **206**
dynamic RAM, **206**

E

eip, **206**
errno, **206**
etext, **205**
exclusion mutuelle, **207**
execve, 136
exit, 133

F

fichier, **206**
fichier header, **205**
fichier objet, **206**
fifo, **208**
file FIFO, **208**
filesystem, **207**
fork, 129
FreeBSD, **204**
FSF, **205**

G

garbage collector, **205**
gcc, **204**
GHz, **206**
Gnome, **204**
GNU, **204**
GNU is not Unix, **204**
GNU/Linux, **204**
gnuth, **208**

H

handler, **208**
heap, **205**
hiérarchie de mémoire, **206**

I

init, 128
inode, **207**
instruction pointer, **206**
interpréteur, **208**
interruption, **207**

K

kernel, 127, **207**
kHz, **206**

L

libc, **205**
librairie dynamique, **207**
librairie partagée, **207**
librairie statique, **207**
lien symbolique, **207**
ligne de cache, **206**
lignes de cache, **206**
linker, **206**
Linus Torvalds, **204**
Linux, **204**
little endian, **207**
livelock, **207**
liveness, **207**
llvm, **204**
localité spatiale, **206**
localité temporelle, **206**
lock, **208**
loi de Amdahl, **207**
loi de Moore, **206**

M

mémoire, **206**
mémoire cache, **206**
Mémoire partagée, **208**
mémoire virtuelle, **208**
MacOS, **204**
makefile, **206**
mandatory lock, **208**
mandatory locking, **208**
memory leak, **205**
Memory Management Unit, **208**
MHz, **206**
microprocesseur, **204**
Minix, **204**
MIPS, **207**
MMU, **208**
mode d'adressage, **206**
mode protégé, 128, **207**
mode utilisateur, 128, **207**
multi-coeurs, **207**
multi-threadé, **207**
multitâche, **207**
multitasking, **207**
mutex, **207**

N

négation, **205**
nibble, **204**
NOT, **205**

O

octet, **204**
offset pointer, **207**
opération atomique, **207**
open file object, 173, **208**
OpenBSD, **204**
OR, **205**
ou exclusif, **205**

P

page, **208**
page fault, **208**
pagination, **208**
partition de swap, **208**
pc, **206**
pid, **205**
pile, **205**
pipe, **204**
pointeur, **205**
portée, **205**
portée globale, **205**
portée locale, **205**
préprocesseur, **204**
principe de localité, **206**
problème des philosophes, **207**
problème des readers-writers, **207**
problèmes des readers-writers, **207**
process identifier, **205**
processeur, **204**
processus, **205**
processus fils, 129, **207**
processus orphelin, **207**
processus père, 129, **207**
processus zombie, **207**
program counter, **206**

R

répertoire, **207**
répertoire courant, **207**
RAM, **206**
Random Access Memory, **206**
reference bit, **208**
registre, **206**
RFC
    RFC 20, 14, 20, 21
RISC, **203**
root, **208**
round-robin, **207**

S

sémaphore, **207**
sémaphore nommé, **208**
safety, **207**
scheduler, **207**
secteur, **207**
section critique, **207**
segment de données, **207**
segment des données initialisées, **205**
segment des données non-initialisées, **205**
segment text, **205**
segmentation fault, **205**
sem_close, 171
sem_open, 171
sem_unlink, 171
shared library, **207**
shell, **204**
sig_atomic_t, 162
SIG_DLF, 161
SIG_IGN, 161
SIGALRM, 168
siginterrupt, 168
signal, **205**
signal asynchrone, **208**
signal synchrone, **208**
simple précision, **205**
Solaris, **204**
Solid State Drive, **208**
SRAM, **206**
SSD, **208**
stack, **205**
static library, **207**
static RAM, **206**
stderr, **204**
stdin, **204**
stdio, **208**
stdout, **204**
stratégie de remplacement de pages, **208**
sureté, **207**
swapping, **208**
système de fichiers, **207**

T

table des pages, **208**
table des processus, 140, **205**
tas, **205**
text, **205**
thread-safe, **207**
TLB, **208**
Translation Lookaside Buffer, **208**

U

Unix, **203**
userid, **208**

V

vivacité, **207**
von Neumann, **205**

W

wait, 133
waitpid, 133
warning, **204**
write back, **206**
write through, **206**
write-back, **206**

X

X11, **204**
x86, **203**
XOR, **205**