LEPL1503 : Introduction au langage C

Version 2021

O. Bonaventure, G. Detal, C. Paasch

févr. 16, 2022

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 . . . 35
2.4 Unions et énumérations . . . 38
2.5 Organisation de la mémoire . . . 40
2.6 Compléments de C . . . 52

3 Systèmes Multiprocesseurs **65**
3.1 Utilisation de plusieurs threads . . . 65
3.2 Communication entre threads . . . 72
3.3 Coordination entre threads . . . 76
3.4 Les sémaphores . . . 83
3.5 Compléments sur les threads POSIX . . . 87

4 Gestion des fichiers **93**
4.1 Gestion des utilisateurs . . . 93
4.2 Systèmes de fichiers . . . 94
4.3 Annexes . . . 108

Bibliographie **115**

Index **117**

CHAPITRE 1

Introduction

1.1 Introduction

Les systèmes informatiques jouent un rôle de plus en plus important dans notre société. Depuis les premiers calculateurs à la fin de la seconde guerre mondiale, 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

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.

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
- 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 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

Les systèmes informatiques peuvent prendre différentes formes, allant de minuscules systèmes embarqués à de gigantesques supercalculateurs. Les *raspberry pi* sont un exemple d'un système embarqué. Il s'agit de nano-ordinateurs, de la taille d'une carte de crédit. Possédant les mêmes composants que décrits ci-dessus, ils fonctionnent de la même façon que des systèmes plus imposants comme les ordinateurs personnels que l'on utilise au quotidien, seulement avec moins de ressources.

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 AT&T Bell Laboratories, Unix a été ensuite développé par d'autres entreprises. C'est aujourd'hui une marque déposée par `The Open group`, voir https://www.opengroup.org/membership/forums/platform/unix
- *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 piles de 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 de 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 simples qui permettent de résoudre un grand nombre de problèmes courants. Certains de ces utilitaires sont chargés automatiquement lors du démarrage de la machine. La

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.

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* en anglais - *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 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'appuie 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.

Pour vous permettre de mettre vos apprentissages en pratique, vous recevrez durant le quadrimestre un raspberry pi. Il est possible d'installer différents systèmes d'exploitation sur celui-ci. Nous utiliserons raspbian qui est lui aussi une variante de Unix.

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

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.

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 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 uniques ou dupliquées (requiert que le fichier d'entrée soit trié, car ne compare que les lignes consécutives)
- 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 utilitaires 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* 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,

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

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 :

```
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="LEPL"
COURS=1503
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 : LEPL
Liste des arguments : LEPL 1503
```

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 : LEPL
Liste des arguments : LEPL 1503
```

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).

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.

```
#!/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 efficacement 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!*.

```
/**************************************
 * Hello.c
 *
 * Programme affichant sur la sortie
```

(suite sur la page suivante)

(suite de la page précédente)

```
 * 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. À 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 "LEPL1503"
  ```

- 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 :

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

  — `<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
```

— 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 déboguer 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` : utilisés lors de la déclaration d'une variable 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 booléennes 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 langage 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.

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

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

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++)
```

(suite sur la page suivante)

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[])`)

(suite de la page précédente)

```
    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
argument[3] : abcdef
argument[4] : lepl1503
```

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 (typiquement 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.

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

```
/**************************************
 * 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. À titre d'exemple, voici un petit programme utilisant printf(3)

```

  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);

```

(suite sur la page suivante)

(suite de la page précédente)

```
  // 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
L
E
P
L
1
5
0
3
```

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 (une table ASCII simple est disponible à cette adresse). 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.

```
/**************************************
 * 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
// exemple simplifié, voir isdigit dans la librarire standard
// pour une solution complète
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);
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
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

Dans la version en-ligne 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.

Il existe de nombreux livres consacrés au langage C. La référence la plus classique est *[KernighanRitchie1998]*, mais certains éléments commencent à dater. Un tutoriel intéressant a été publié par Brian Kernighan *[Kernighan]*. *[King2008]* propose une présentation plus moderne du langage C.

2.2 Types de données

Dans les sections précédentes, 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 détail 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 $b_0 ... b_i ... b_{n-1}$ peut représenter le nombre entier $\sum_{i=0}^{n-1} b_i \times 2^i$. Par convention, le bit b_{n-1}, associé au facteur du plus grand indice 2^{n-1}, 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 $b_{n-1} ... b_i ... b_0$. À titre d'exemple, la suite de bits `0101` correspond à l'entier non signé représentant la valeur cinq. Le bit de poids fort de cette séquence de quatre bits (ou *nibble*) est `0`. Le bit de poids faible est `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

Écrire 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 (optimale) 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 !!
```

(suite sur la page suivante)

(suite de la page précédente)

```
//  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 la réalisation 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 [2] :

— 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 [1].
— Les *f* bits de poids faible servent à stocker la partie fractionnaire du nombre réel.

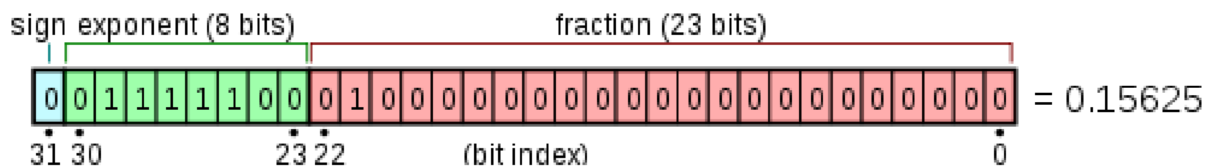

Fig. 1 – Exemple de nombre en virgule flottante (simple précision). (source : Wikipedia)

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

#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 symbole. Dans l'exemple ci-dessus, la chaîne `10` est associée au symbole `N`. Lors de chaque compilation, le préprocesseur remplace toutes les occurrences de `N` par `10`. Cela permet au compilateur de ne traiter que des tableaux de taille fixe.

2. Source : https://en.wikipedia.org/wiki/Single-precision_floating-point_format

1. 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]*.

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` et 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 *[RFC20]*. La table de caractères ASCII définit une correspondance entre des séquences de bits et des caractères. *[RFC20]* contient la table des caractères ASCII représentés sur 7 bits. À 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*. De nombreux détails sur la table ASCII sont disponibles sur la page Wikipedia : https://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange

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. À 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 en 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.

À titre d'exemple, la fonction toupper(3) qui est implémentée dans les versions récentes 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 de ce document, 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 être initialisée 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. À 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 `ArrayIndexOutOfBoundsException`. 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 (langage de bas niveau) 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 [3]). 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 libérer 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 une 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 zones permettant chacune 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.

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';
```

(suite sur la page suivante)

3. 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 https://docs.oracle.com/javase/6/docs/technotes/tools/solaris/java.html

(suite de la page précédente)

```
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)
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
  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 à l'adresse %p\n", tab[i], &(tab[i]));
}
```

```
1020304 est à l'adresse 0x7fff5fbff750
5060708 est à l'adresse 0x7fff5fbff754
90A0B0C est à 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 à 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 stockée 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'en est 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 à 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 à l'adresse 0x7fff5fbff750
3 est à l'adresse 0x7fff5fbff751
2 est à l'adresse 0x7fff5fbff752
1 est à 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`. Le pointeur `ptr_char` a été incrémenté 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. À 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 objet 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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
// 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`, elle, définit 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 système 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 [4]. La première sert à mesurer le nombre de secondes et de microsecondes qui se sont écoulées depuis le premier janvier 1970. Cette structure, baptisée `struct timeval` est définie dans sys/time.h comme suit :

4. 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 système tels que gettimeofday(2) pour notamment récupérer l'heure courante ou les appels de manipulation de temporisateurs (*timers* en anglais) 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 [5] ou les formats de paquets qui sont échangés sur le réseau [6].

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 `Entier` comme alias pour le type `int` et `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

Renommer des 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. À 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 types de données plus courts. Par exemple, il est très courant de remplacer le types `unsigned` par les abréviations ci-dessous.

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

5. 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.

6. Parmi les exemples simples, on peut citer la structure `struct ipv6hdr` qui correspond à l'entête du protocole IP version 6 et est définie dans linux/ipv6.h.

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

```
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 faciliter la compréhension d'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;
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
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 compilé. 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 [7].

```c
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 valeurs. 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.

```c

struct fraction quart;
```

(suite sur la page suivante)

7. 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.

(suite de la page précédente)

```
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 représentations 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 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. À 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.

```c
#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 accessible 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.

```c
float g;   // variable globale
```

(suite sur la page suivante)

(suite de la page précédente)

```
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 `./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;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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 {
```

(suite sur la page suivante)

(suite de la page précédente)

```
  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 occurrences 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 préférera la placer 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.

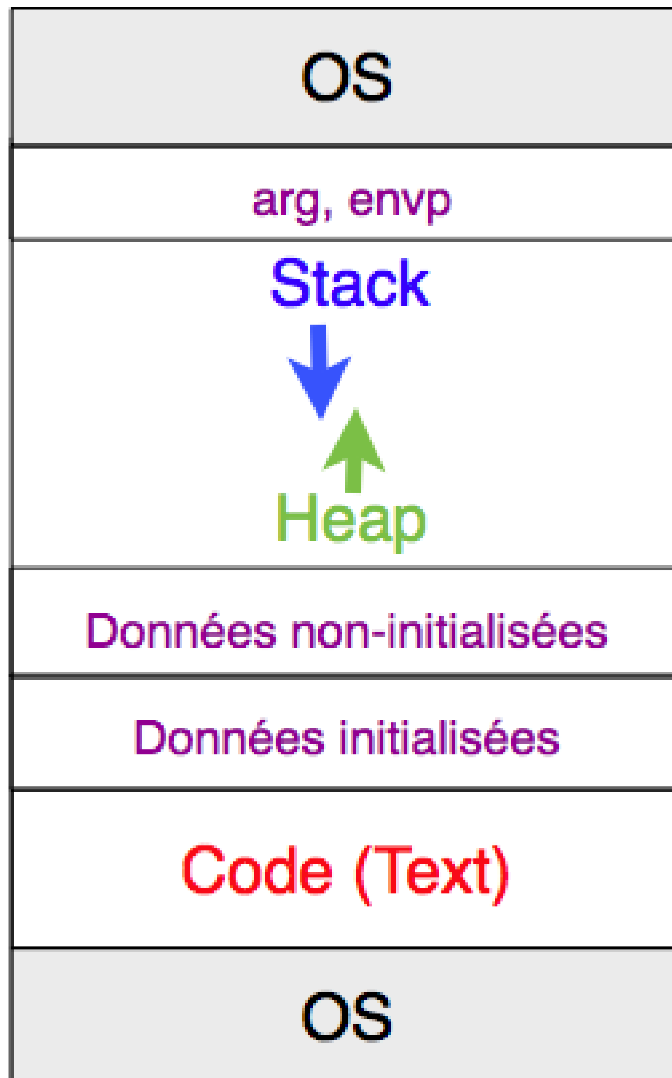

Fig. 2 – Organisation d'un programme Linux en mémoire

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 [3]. 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

3. 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.

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*.

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)sizeof(double));
```

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 [1] 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) {
  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);
```

(suite sur la page suivante)

1. 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.

(suite de la page précédente)

```
  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. Cette conversion 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) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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 (`./S3-src/stack.c`) illustre une 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) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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 `./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 [5]. La librairie standard contient cependant d'autres fonctions permettant d'allouer de la mémoire mais aussi de modifier des allocations antérieures. 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);
}
```

5. 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 [6] :

- `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.

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

6. 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.

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 `./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);
```

(suite sur la page suivante)

(suite de la page précédente)

```
  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) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
  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 débogage. L'exemple ci-dessous est une application qui supporte trois niveaux de débogage. 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 débogage utilise pourtant toujours le même appel pour afficher l'information de débogage : `(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. `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 [1].

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 référence peut être affichée par printf(3).

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

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

(suite sur la page suivante)

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

(suite de la page précédente)

```
  }
  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 de 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 le 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

Dès que le programme que l'on développe en C est un programme non trivial d'une certaine taille, 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, etc. Ces modules peuvent en quelque sorte être comparés aux classes d'un programme Java.

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
```

(suite sur la page suivante)

(suite de la page précédente)

```
 *
 **************************************/

#include "min.h"
#include <stdio.h>
#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 extérieurs sont reprises dans le *fichier header*, c'est-à-dire que les fonctions utilisées uniquement pour des opérations internes ne sont pas présentes.

— 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

```
/**************************************
 * 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.

La seconde partie d'un module est son *fichier source*, qui contient l'implémentation en C des fonctions définies dans le *fichier header* correspondant, ainsi que toute variable ou fonction supplémentaire, qui ne sont donc pas visibles sur l'interface externe. Le *fichier source* contient donc le code C correspondant à l'interface externe.

Un *fichier source* doit contenir une directive `#include` pour inclure le *fichier header* correspondant, et porte également le même nom, avec l'extension `.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 petit exemple 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`.

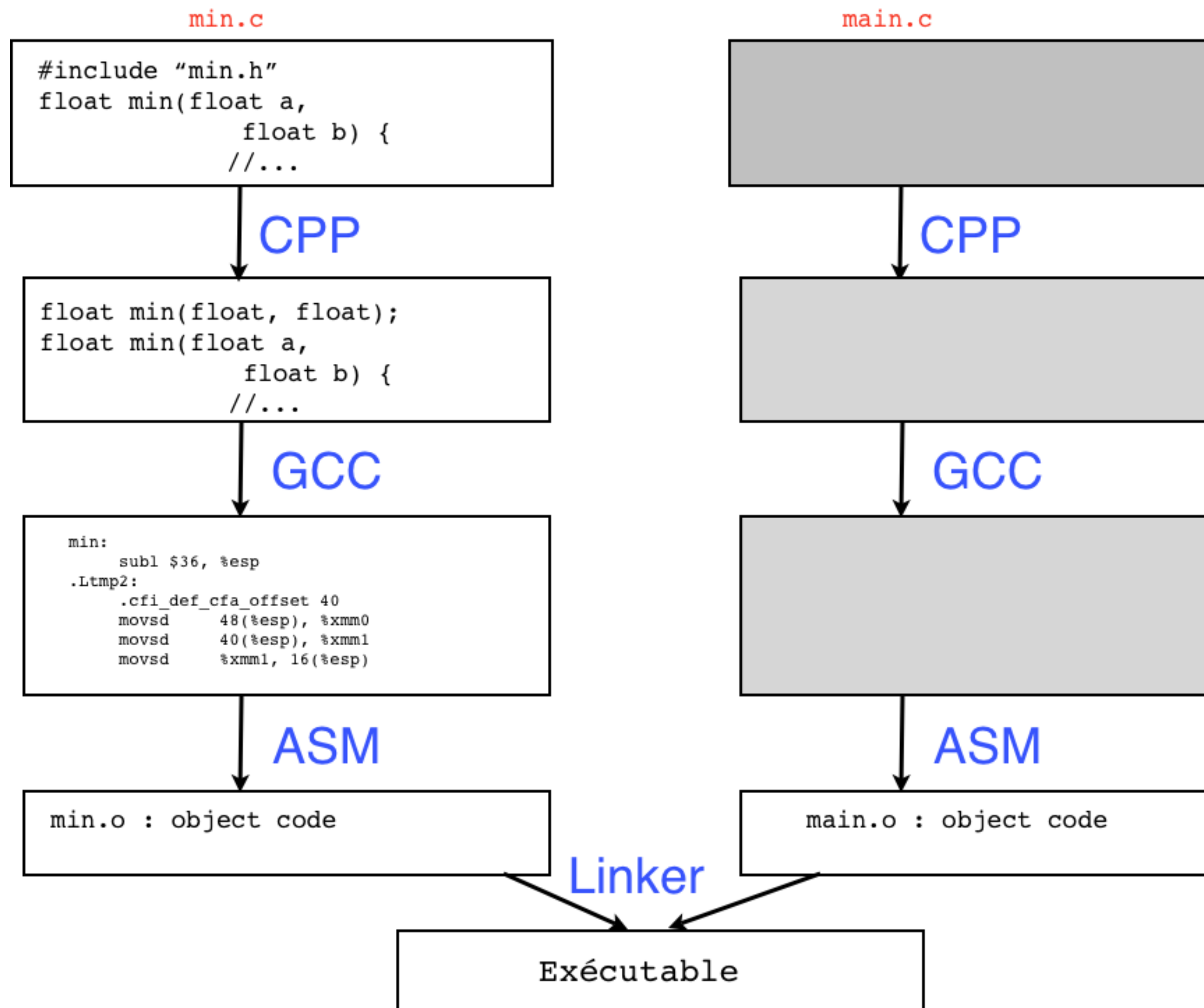

Fig. 3 – Etapes de compilation

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 `static`. Lorsque `static` 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`. 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 lié 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

`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, cela 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.

Il faut noter que `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 `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;
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
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 accessible depuis n'importe quel module. Sa signature est reprise dans le *fichier header* `module.h`. La fonction `min` par contre est déclarée comme étant `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 lié 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 pas 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

Systèmes Multiprocesseurs

3.1 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 micro-électronique 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 de 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 puce 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 difficile 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

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

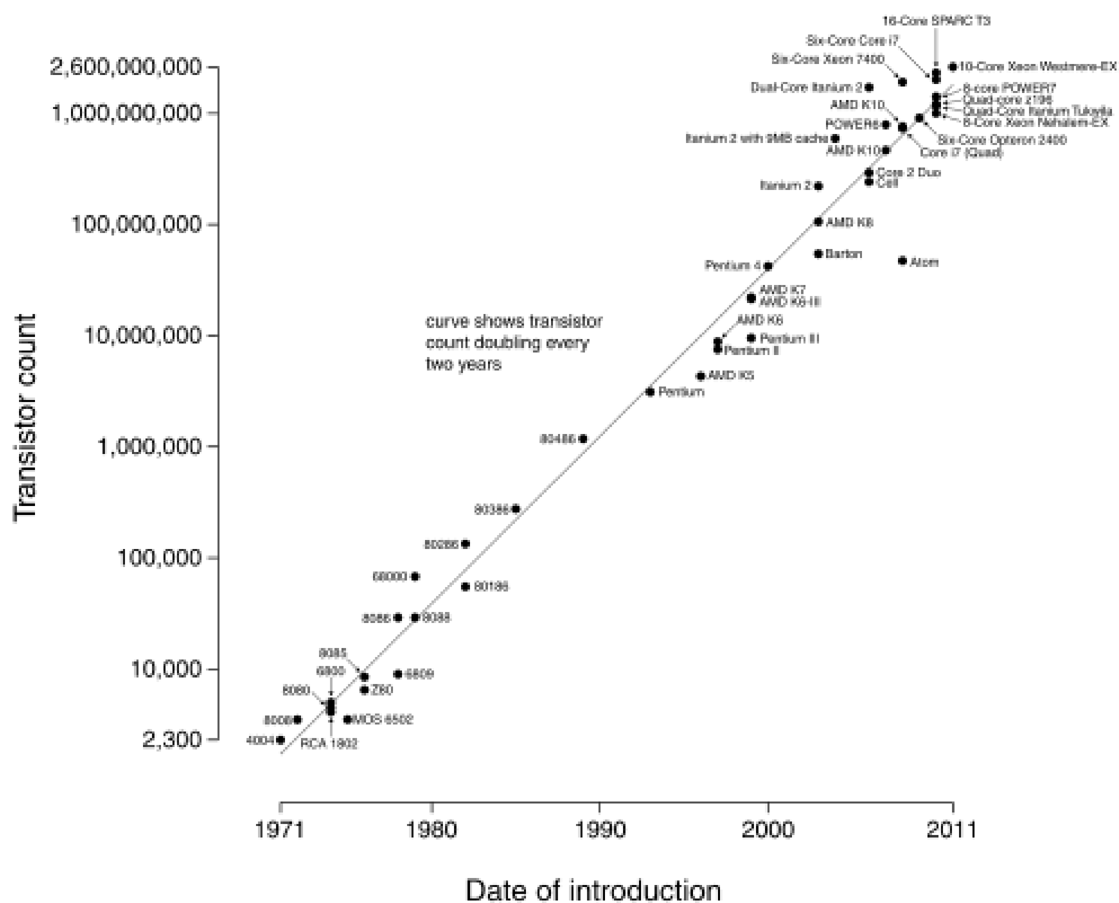

Fig. 1 – Evolution du nombre de transistors par microprocesseur

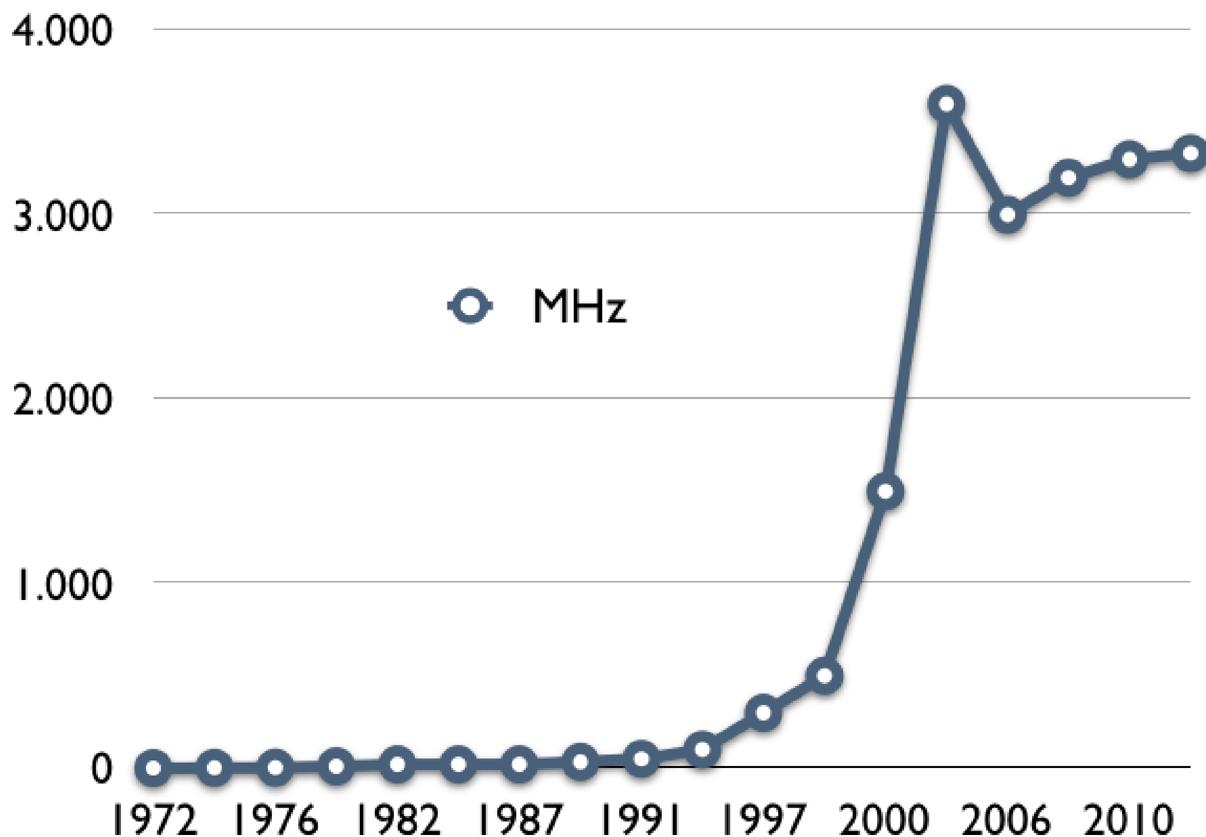

Fig. 2 – Evolution de la vitesse d'horloge des microprocesseurs

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.

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 reproductible. 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

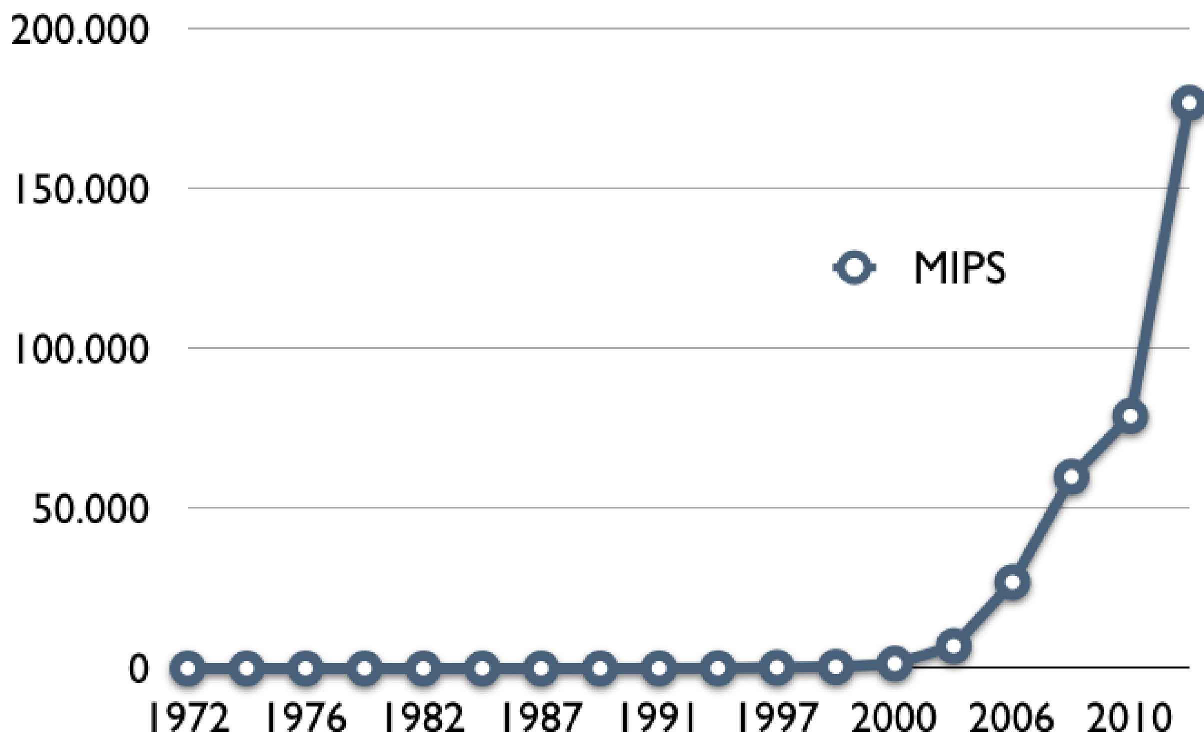

Fig. 3 – Evolution des performances des microprocesseurs en MIPS

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;
}
```

Pour qu'un processeur puisse exécuter cette séquence d'instructions, il faut 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, des zones de mémoire très rapide (mais peu nombreuses) se trouvant sur le processeur qui permettent de stocker entre autres : l'adresse de l'instruction à exécuter, des résultats intermédiaires obtenus durant l'exécution d'un instruction ou encore des informations sur la pile.

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 croître 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 comment implémenter 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.

3.1.1 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 convertir 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);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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.

3.2 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 (c'est-à-dire les constantes, les variables globales ou encore les chaînes de caractère) 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.

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 [1].

1. Le programme complet est accessible via `./S5-src/pthread-test.c`

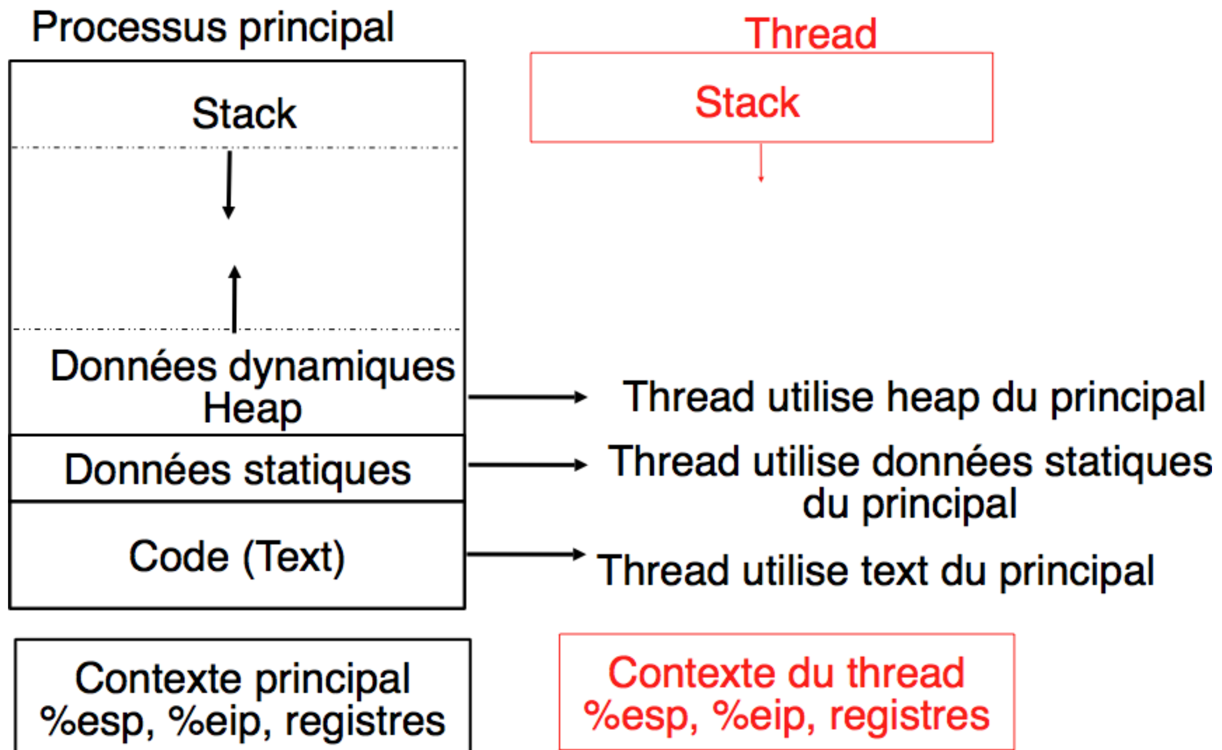

Fig. 4 – Organisation de la mémoire après la création d'un thread POSIX

```
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++) {
   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);
```

(suite sur la page suivante)

(suite de la page précédente)

```
}
```

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 [2] (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;
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)
```

(suite sur la page suivante)

2. Le programme complet est accessible via `./S6-src/pthread.c`

(suite de la page précédente)

```
    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
```

3.3 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.

3.3.1 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 *sûreté* (*safety* en anglais). Dans un programme découpé en threads, une propriété de *sûreté* 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[3] 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[4]. 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 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.

3. 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.
4. Les appels systèmes sont décrits dans la section 2 des pages de manuel tandis que la section 3 décrit les fonctions de la librairie.

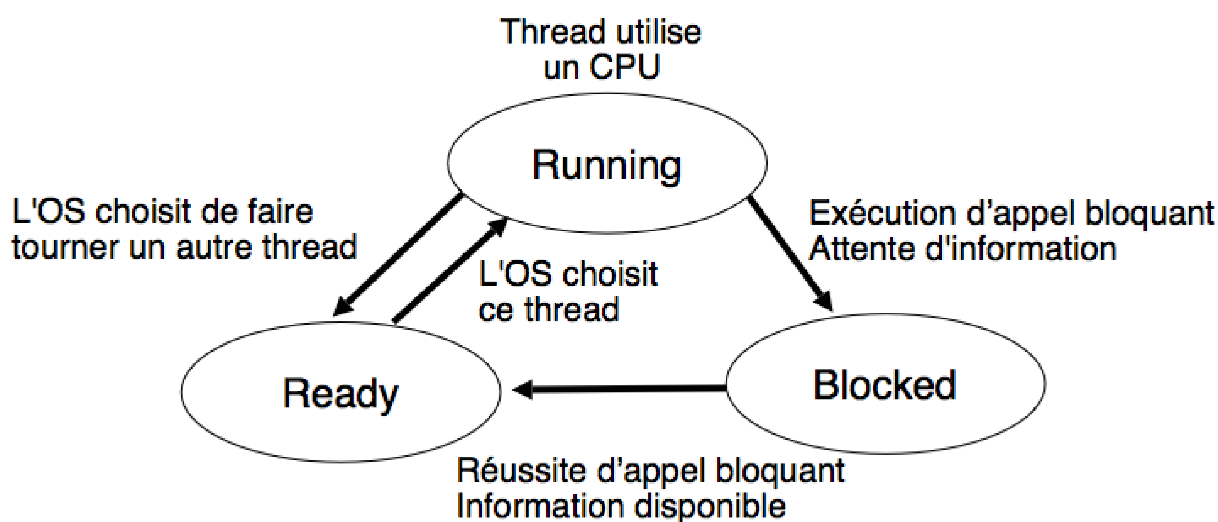

Fig. 5 – Etats d'un thread d'exécution

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 libérera. 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 le langage C, 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 librairie 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 [5].

5. 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

Coordination par Mutex

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.

des interruptions peut être particulièrement coûteuse en termes de performances dans un environnement multiprocesseurs.

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);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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 [6]. 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.

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 sûreté et la propriété de vivacité. Pour la propriété de sûreté, 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).

6. 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.

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.

3.4 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.

3.4.1 Sémaphores POSIX

La librairie POSIX comprend une implémentation des sémaphores [1] 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 :

```c
#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 cette 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.

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.

1. 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 les es sémaphores *System V*. Les principales fonctions pour les sémaphores *System V* sont semget(3posix), semctl(3posix) et semop(3posix).

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 livre 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.

3.4.2 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 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.

3.4.3 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 places comme illustré dans la figure ci-dessous.

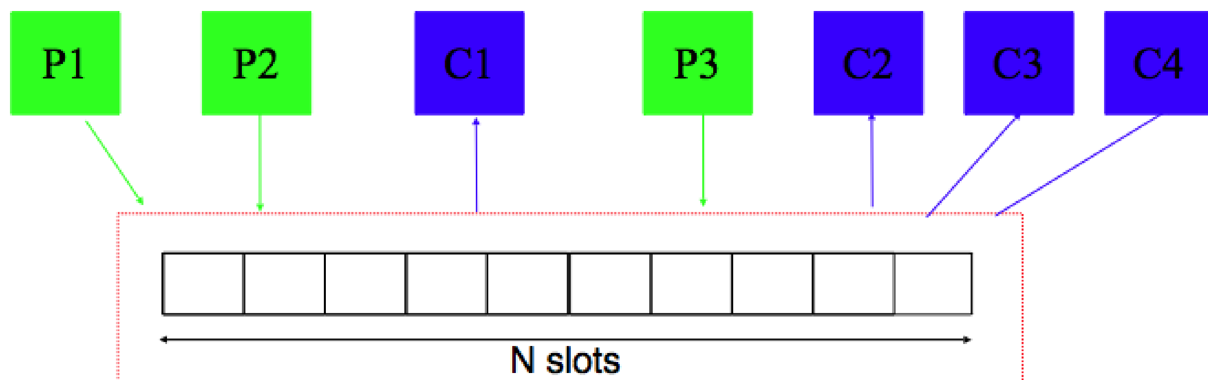

Fig. 6 – Problème des producteurs-consommateurs

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 une place 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 places qui sont vides dans le buffer partagé. Ce sémaphore doit être initialisé à la taille du buffer puisque celui-ci est initialement vide. Le second sémaphore est baptisé `full` dans le pseudo-code ci-dessous. Sa valeur représente le nombre de places du buffer qui sont occupées. Il doit être initialisé à la valeur `0`.

```
// Initialisation
#define N 10 // places dans le 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 une place 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)
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
  int item;
  while(true)
  {
    item=produce(item);
    sem_wait(&empty); // attente d'une place libre
    pthread_mutex_lock(&mutex);
     // section critique
     insert_item();
    pthread_mutex_unlock(&mutex);
    sem_post(&full); // il y a une place remplie 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'une nouvelle place est disponible dans le buffer.

```
// Consommateur
void consumer(void)
{
 int item;
 while(true)
 {
   sem_wait(&full); // attente d'une place remplie
   pthread_mutex_lock(&mutex);
    // section critique
    item=remove(item);
   pthread_mutex_unlock(&mutex);
   sem_post(&empty); // il y a une place libre en plus
 }
}
```

De nombreux programmes découpés en threads fonctionnent avec un ensemble de producteurs et un ensemble de consommateurs.

3.5 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 découpés en threads en langage C.

3.5.1 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 librairie 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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 est utilisée comme 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();
```

(suite sur la page suivante)

(suite de la page précédente)

```
  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 préférera évidemment d'utiliser le qualificatif `__thread` plutôt que d'utiliser une 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.

3.5.2 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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 [3]. 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;
```

(suite sur la page suivante)

3. Cette implémentation est adaptée de https://opensource.apple.com/source/gcc/gcc-926/libiberty/strerror.c et est dans le domaine public.

(suite de la page précédente)

```
 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).

```
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;
  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.

CHAPITRE 4

Gestion des fichiers

4.1 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 formatter 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.

4.2 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.

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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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 contiennent, 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

```
#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.

```
$ ls -lai /etinfo/users/obo
total 1584396
drwx------ 78 obo  stafinfo      4096 Mar 17 00:34 .
drwxr-xr-x 93 root root          4096 Feb 22 11:37 ..
-rwxr-xr-x  1 obo  stafinfo     11490 Feb 28 00:43 a.out
-rw-------  1 obo  stafinfo      4055 Mar 22 15:13 .bash_history
-rw-r--r--  1 obo  stafinfo        55 Sep 18  1995 .bash_profile
-rw-r--r--  1 obo  stafinfo       101 Aug 28  2003 .bashrc
drwxr-xr-x  2 obo  stafinfo      4096 Nov 22  2004 bin
-rw-r--r--  1 obo  stafinfo       346 Feb 13 15:37 hello.c
drwxr-xr-x  3 obo  stafinfo      4096 Mar  2 09:30 sinf1252
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 noyau 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 `..`. Dans chaque répertoire, cet alias 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 dates de création/modification associées 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 de l'inode, c'est-à-dire la métadonnée qui contient les informations relatives au fichier/répertoire, la position de l'entrée `dirent` qui suite, 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) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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 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--  2 obo  stafinfo  5 24 mar 21:14 test.txt
9624126 -rw-r--r--  2 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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) 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]*.

4.2.1 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).

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

```

(suite sur la page suivante)

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`.

(suite de la page précédente)

```
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 l'ouvre ensuite. 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_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 .. - `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).

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 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);
```

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 contiguës 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);
}
```

(suite sur la page suivante)

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).

(suite de la page précédente)

```
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){
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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].

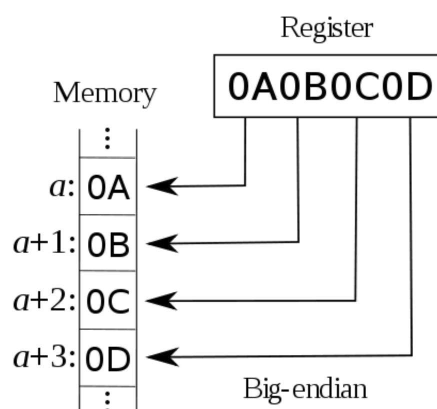

Fig. 1 – Ecriture d'un entier 32 bits en mémoire en *big endian*

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.

6. Source : https://en.wikipedia.org/wiki/Endianness

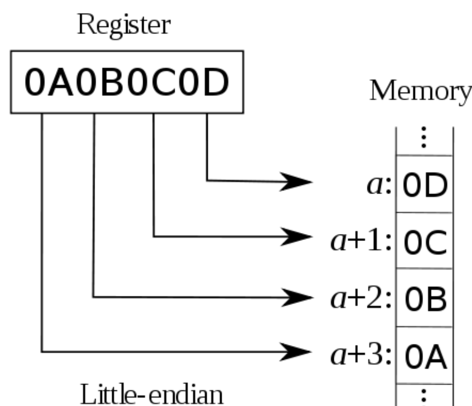

Fig. 2 – Ecriture d'un entier 32 bits en mémoire en *little endian*

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 modèle 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)
```

(suite sur la page suivante)

(suite de la page précédente)

```
// 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.

4.2.2 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 un fichier accessibles directement dans la mémoire du processus.

```
#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. Le deuxième argument est la longueur de la zone 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, le fichier est mappé dans chaque processus, mais si un processus modifie une page, cette modification n'est pas répercutée aux autres processus qui ont mappé le même 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. Le 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 [7]. 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);
```

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);
```

(suite sur la page suivante)

7. Il est possible d'obtenir la taille des pages utilisée sur un système via les appels sysconf(3) ou getpagesize(2)

(suite de la page précédente)

```
  }

  // 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))  ==␣
↪(void *)(-1)) {
    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)) == (void *)(-1)) {
    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);
}
```

4.3 Annexes

4.3.1 Bibliographie

4.3.2 Glossaire

/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)

adresse Position d'un donnée en mémoire.

AND

conjonction logique Opération binaire logique.

Andrew Tanenbaum Andrew Tanenbaum est professeur à la VU d'Amsterdam.

appel système Fonction primitive fournie par le noyau du système d'exploitation et pouvant être appelée directement par les programmes applicatifs.

appel système bloquant Appel système qui ne retourne par de résultat immédiat. Dans ce cas, le noyau du système d'exploitation sélectionne un autre processus via le *scheduler* en attendant que le résultat de l'appel système soit disponible.

appel système lent Un appel système lent est un appel système qui peut attendre un temps indéfini pour se terminer. Par exemple, l'appel read(2) sur l'entrée standard ne retournera de résultat que lorsque l'utilisateur aura pressé une touche sur le clavier.

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)).

Aqua Aqua est une interface graphique spécifique à *MacOS*. Voir la page Wikipédia.

assembleur Programme permettant de convertir un programme écrit en langage d'assemblage dans le langage machine correspondant à un processeur donné.

benchmark Ensemble de programmes permettant d'évaluer les performances d'un système informatique.

big endian Ordre dans lequel les octets correspondants à des mots de plusieurs octets sont stockés en mémoire. Voir https://fr.wikipedia.org/wiki/Boutisme#Gros-boutisme

bit Plus petite unité d'information. Par convention, un bit peut prendre les valeurs `0` et `1`.

bit de poids faible Par convention, bit le plus à droite d'une séquence de n bits.

bit de poids fort Par convention, le bit le plus à gauche d'une séquence de n bits.

BSD Unix Variante de Unix développée à l'Université de Californie à Berkeley.

buffer overflow Erreur dans laquelle un programme informatique cherche à stocker plus de données en mémoire que la capacité de la zone réservée en mémoire. Donne généralement lieu à des problèmes, parfois graves, de sécurité. https://en.wikipedia.org/wiki/Buffer_overflow

byte

octet Un bloc de huit bits consécutifs.

C Langage de programmation permettant d'interagir facilement avec le matériel.

C99 Standard international définissant le langage C *[C99]*

changement de contexte Passage de l'exécution du programme A au programme B.

CISC Complex Instruction Set Computer

contexte Structure de données maintenue pas le noyau du système d'exploitation qui contient toutes les informations nécessaires pour poursuivre l'exécution d'un programme.

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 https://gcc.gnu.org/onlinedocs/cpp/

CPU Central Processing Unit. Voir *microprocesseur*.

deadlock Voir https://en.wikipedia.org/wiki/Deadlock

debugger Logiciel

descripteur de fichier Identifiant (entier) retourné par le noyau du système d'exploitation lors de l'ouverture d'un fichier par l'appel système open(2).

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.

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.

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.

errno Variable globale mise à jour par certains appels systèmes et fonctions de la librairie standard en cas d'erreur. Voir errno(3)

exclusion mutuelle Zone d'un programme multithreadé qui ne peut pas être exécutée par plus d'un thread à la fois.

fichier Une séquence composée d'un nombre entier d'octets stockée sur un dispositif de stockage. Un fichier est identifié par son nom et sa position dans l'arborescence du système de fichiers.

fichier header Fichier contenant des signatures de fonctions, des déclarations de types de données, des variables globales, permettant d'utiliser une librairie ou un API.

fichier objet Fichier résultat de la compilation d'une partie de programme. Ce fichier contient les instructions en langage machine à exécuter ainsi que les informations relatives aux différents symboles (variables, fonctions, ...) qui y sont définis.

fichier source Fichier contenant l'implémentation des fonctions définies dans le *fichier header* correspondant, ainsi que de potentielles autres variables ou fonctions utiles.

FreeBSD Variante de BSD Unix disponible depuis https://www.freebsd.org

FSF Free Software Foundation, https://www.fsf.org

garbage collector Algorithme permettant de libérer la mémoire qui n'est plus utilisée notamment dans des langages tels que Java

gcc Compilateur pour la langage C développé par un groupe de volontaires qui est diffusé depuis https://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 https://en.wikipedia.org/wiki/List_of_compilers#C_compilers

GHz Mesure de fréquence en milliards de répétitions par seconde.

Gnome Environnement graphique utilisé par de nombreuses distributions Linux. Voir https://en.wikipedia.org/wiki/GNOME

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* .

heap

tas Partie de la mémoire d'un programme gérée par malloc(3) et free(3).

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.

inode structure de données utilisée par le système de fichiers Unix pour représenter un fichier/répertoire

interruption Signal extérieur (horloge, opération d'entrée/sortie, ...) qui force le processeur à arrêter l'exécution du programme en cours pour exécuter une routine du système d'exploitation et traiter l'interruption.

kHz Mesure de fréquence en milliers de répétitions par seconde.

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.

lien symbolique Unix supporte deux types de liens. Les liens durs créés par ln(1) et les liens symboliques créés par ln(1) avec l'argument `-s`.

linker Editeur de liens. Partie du compilateur c permettant de combiner plusieurs fichiers objet en un exécutable.

Linus Torvalds Linus Torvalds est le créateur et le mainteneur principal du noyau *Linux*.

Linux Noyau de système d'exploitation compatible Unix développé initialement par Linus Torvalds.

little endian Ordre dans lequel les octets correspondants à des mots de plusieurs octets sont stockés en mémoire. Voir https://fr.wikipedia.org/wiki/Boutisme#Petit-boutisme

livelock Voir https://en.wikipedia.org/wiki/Deadlock#Livelock

liveness

vivacité Propriété d'un programme informatique. Dans le problème de l'exclusion mutuelle, une propriété de vivacité est qu'un thread qui souhaite entrer en section critique finira par y accéder.

llvm Ensemble de compilateurs pour différents langages de programmation et différents processeurs développé par un groupe de volontaire. `llvm` est distribué depuis https://llvm.org/

loi de Amdahl Voir https://fr.wikipedia.org/wiki/Loi_d%27Amdahl

loi de Moore Voir https://fr.wikipedia.org/wiki/Loi_de_Moore

MacOS Système d'exploitation développé par Apple Inc. comprenant de nombreux composantes provenant de *FreeBSD*.

makefile Fichier décrivant la façon dont make(1) doit compiler un programme.

memory leak Fuite de mémoire. Erreur concernant un programme qui a alloué de la mémoire avec malloc(3) et ne l'utilise plus sans avoir fait appel à free(3)

mémoire Dispositif électronique permettant de stocker

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.

MHz Mesure de fréquence en millions de répétitions par seconde.

microprocesseur

processeur Unité centrale de l'ordinateur qui exécute les instructions en langage machine et interagit avec la mémoire.

Minix Famille de noyaux de systèmes d'exploitation inspiré de *Unix* développée notamment par *Andrew Tanenbaum*. Voir https://www.minix3.org pour la dernière version de Minix.

MIPS Million d'instructions par seconde

multi-threadé

multi-coeurs Processeur contenant plusieurs unités permettant d'exécuter simultanément des instructions de programmes différents.

multitâche

multitasking Capacité d'exécuter plusieurs programmes simultanément.

multithreadé Programme utilisant plusieurs threads.

mutex Primitive de synchronisation permettant d'empêcher que deux threads accèdent simultanément à une même section critique.

nibble Un bloc de quatre bits consécutifs.

NOT

négation Opération binaire logique.

offset pointer Position de la tête de lecture associée à un fichier ouvert.

OpenBSD Variante de BSD Unix disponible depuis https://www.openbsd.org

opération atomique Opération ne pouvant être interrompue.

OR

disjonction logique Opération binaire logique.

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.

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.

pointeur Adresse d'une variable ou fonction en mémoire.

portée Zone d'un programme dans laquelle une variable est déclarée.

portée globale Une variable ayant une portée globale est accessible dans tout le programme.

portée locale Une variable ayant une portée locale est accessible uniquement dans le bloc dans laquelle elle est définie.

principe de localité principe de fonctionnement de la mémoire indiquant que lorsqu'un programme accède à une adresse à un temps t, il accédera encore à des adresses proches dans les prochains instants

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.

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*.

raspberry pi Systèmes informatiques développés par la Raspberry Pi Foundation, voir https://www.raspberrypi.org

registre Unité de mémoire intégrée au processeur. Les registres sont utilisés comme source ou destination pour la plupart des opérations effectuées par un processeur.

répertoire Branche de l'arborescence du système de fichiers. Un répertoire contient un ou plusieurs fichiers.

répertoire courant Répertoire dans lequel l'appel système open(2) cherchera à ouvrir les fichiers do

RISC Reduced Instruction Set Computer

root Racine de l'arborescence des fichiers mais aussi utilisateur ayant les privilèges les plus élevés sur un ordinateur utilisant Unix.

round-robin Voir Round-robin sur Wikipédia

scheduler Ordonnanceur. Algorithme utilisé par le noyau du système d'exploitation pour sélectionner le prochain programme à exécuter après une interruption d'horloge ou un appel système bloquant.

section critique Partie de programme ne pouvant pas être exécutée simultanément par deux threads différents.

segment de données Partie de la mémoire comprenant les segments des données initialisées et non-initialisées

segment des données initialisées Partie de la mémoire d'un programme contenant les données initialisées dans le code source du programme ainsi que les chaînes de caractères.

segment des données non-initialisées Partie de la mémoire d'un programme contenant les données (tableaux notamment) qui sont déclarés mais pas explicitement initialisés dans le code source du programme.

segmentation fault Erreur à l'exécution causée par un accès à une adresse mémoire non-autorisée pour le programme.

sémaphore Primitive de synchronisation permettant notamment l'exclusion mutuelle. Voir notamment *[Downey2008]*

shared library

librairie dynamique

librairie partagée Lorsqu'un librairie est dynamiquement liée à un programme exécutable, le code de celui-ci ne contient pas les instructions de la librairie, mais celle-ci est automatiquement chargée lors de chaque exécution du programme. Cela permet d'avoir une seule copie de chaque librairie. C'est la solution utilisée par défaut sous Linux.

shell Interpréteur de commandes sur un système Unix. bash(1) est l'interpréteur de commandes le plus utilisé de nos jours.

signal mécanisme permettant la communication entre processus. Utilisé notamment pour arrêter un processus via la commande kill(1)

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.

Solaris Système d'exploitation compatible Unix développé par Sun Microsystems et repris par Oracle.

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.

SSD

Solid State Drive Système de stockage de données s'appuyant uniquement sur de la mémoire flash.

stack

pile Partie de la mémoire d'un programme contenant les variables locales et adresses de retour des fonctions durant leur exécution.

static library

librairie statique Une librairie est statiquement liée à un programme exécutable lorsque tout son code est intégré dans l'exécutable. Voir les arguments `static` dans gcc(1)

stderr Sortie d'erreur standard sur un système Unix (par défaut l'écran)

stdin Entrée standard sur un système Unix (par défaut le clavier)

stdout Sortie standard sur un système Unix (par défaut l'écran)

sûreté

safety Propriété d'un programme informatique. Dans le problème de l'exclusion mutuelle, une propriété de sûreté est que deux threads ne seront jamais dans la même section critique.

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*

text

segment text Partie de la mémoire d'un programme contenant les instructions en langage machine à exécuter.

thread-safe Une fonction est dite thread-safe si elle peut être simultanément exécutée sans contrainte par différents threads d'un même programme.

Unicode Norme d'encodage de caractères supportant l'ensemble des langues écrites, voir notamment https://en.wikipedia.org/wiki/Unicode

Unix Système d'exploitation développé initialement par AT&T Bell Labs.

userid Identifiant d'utilisateur. Sous Unix, un entier unique est associé à chaque utilisateur.

von Neumann Un des inventeurs des premiers ordinateurs. A défini l'architecture de base des premiers ordinateurs qui est maintenant connue comme le modèle de von Neumann *[Krakowiak2011]*

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.

X11 Interface graphique développée au MIT pour Unix. Voir https://en.wikipedia.org/wiki/X_Window_System

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.

XOR

ou exclusif Opération binaire logique.

Bibliographie

[ABS] Cooper, M., *Advanced Bash-Scripting Guide*, 2011, https://tldp.org/LDP/abs/html/

[AdelsteinLubanovic2007] Adelstein, T., Lubanovic, B., *Linux System Administration*, OReilly, 2007, https://books.google.be/books?id=-jYe2k1p5tIC

[Bashar1997] Bashar, N., *Ariane 5 : Who Dunnit ?*, IEEE Software 14(3) : 15–16. May 1997. https://www.computer.org/csdl/magazine/so/1997/03/s3015/13rRUyft7B4

[C99] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

[Cooper2011] Cooper, M., *Advanced Bash-Scripting Guide*, https://tldp.org/LDP/abs/html/, 2011

[CPP] C preprocessor manual, https://gcc.gnu.org/onlinedocs/cpp/

[Dijkstra1965b] Dijkstra, E., *Cooperating sequential processes*, 1965, https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html

[Dijkstra1965] Dijkstra, E., *Solution of a problem in concurrent programming control*. Commun. ACM 8, 9 (September 1965), 569 https://dl.acm.org/doi/10.1145/365559.365617

[Downey2008] Downey, A., *The Little Book of Semaphores*, Second Edition, Green Tea Press, 2008, https://greenteapress.com/wp/semaphores/

[DrepperMolnar2005] Drepper, U., Molnar, I., *The Native POSIX Thread Library for Linux*, https://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. https://dl.acm.org/doi/10.1145/103162.103163 ou https://www.validlab.com/goldberg/paper.pdf

[Gove2011] Gove, D., *Multicore Application Programming for Windows, Linux and Oracle Solaris*, Addison-Wesley, 2011, https://books.google.be/books?id=NF-C2ZQZXekC

[GNUPTH] Engelschall, R., *GNU Portable Threads*, https://www.gnu.org/software/pth/

[IA32] intel, *Intel® 64 and IA-32 Architectures : Software Developer's Manual*, Combined Volumes : 1, 2A, 2B, 2C, 3A, 3B and 3C, December 2011, https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

[Kamp2011] Kamp, P., *The Most Expensive One-byte Mistake*, ACM Queue, July 2011, https://queue.acm.org/detail.cfm?id=2010365

[Kerrisk2010] Kerrisk, M., *The Linux Programming Interface*, No Starch Press, 2010, https://man7.org/tlpi/

[Kernighan] Kernighan, B., *Programming in C - A Tutorial*, https://www.bell-labs.com/usr/dmr/www/ctut.pdf

[KernighanRitchie1998] Kernighan, B., and Ritchie, D., *The C programming language, second edition*, Addison Wesley, 1998, https://fr.wikipedia.org/wiki/The_C_Programming_Language

[King2008] King, K., *C programming : a modern approach*, W. W. Norton & company, 2008

[Krakowiak2011] Krakowiak, S., *Le modele d'architecture de von Neumann*, https://interstices.info/le-modele-darchitecture-de-von-neumann

[Leroy] Leroy, X., *The LinuxThreads library*, https://www.lix.polytechnique.fr/~liberti/public/computing/parallel/threads/linuxthreads/linuxthreads-index.html

[McKenney2005] McKenney, P., *Memory Ordering in Modern Microprocessors, Part I*, Linux Journal, August 2005, https://www.linuxjournal.com/article/8211

[Mitchell+2001] Mitchell, M., Oldham, J. and Samuel, A., *Advanced Linux Programming*, New Riders Publishing, ISBN 0-7357-1043-0, June 2001, https://www.advancedlinuxprogramming.com/

[Nemeth+2010] Nemeth, E., Hein, T., Snyder, G., Whaley, B., *Unix and Linux System Administration Handbook*, Prentice Hall, 2010, https://books.google.be/books?id=rgFIAnLjb1wC

[RFC20] Cerf, V., *ASCII format for Network Interchange*, RFC20, October 1969, https://datatracker.ietf.org/doc/html/rfc20.html

[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, https://books.google.be/books?id=wHI8PgAACAAJ

[Stokes2008] Stokes, J., *Classic.Ars : Understanding Moore's Law*, https://arstechnica.com/gadgets/2008/09/moore/

[Tanenbaum+2009] Tanenbaum, A., Woodhull, A., *Operating systems : design and implementation*, Prentice Hall, 2009, https://www.pearson.com/us/higher-education/program/Tanenbaum-Operating-Systems-Design-and-Implementation-3rd-Edition/PGM228096.html

[Walls2006] Walls, D., *How to Use the restrict Qualifier in C*. Sun Microsystems, 2006, https://www.oracle.com/solaris/technologies/solaris10-cc-restrict.html

Index

Symbols

/proc, 108

A

adresse, 108
AND, 108
Andrew Tanenbaum, 108
API, 108
appel système, 108
appel système bloquant, 108
appel système lent, 108
Application Programming Interface, 108
Aqua, 108
assembleur, 108

B

benchmark, 108
big endian, 108
bit, 108
bit de poids faible, 108
bit de poids fort, 108
BSD Unix, 108
buffer overflow, 108
byte, 108

C

C, 108
C99, 108
changement de contexte, 108
CISC, 108
compteur de programme, 109
conjonction logique, 108
contexte, 108
cpp, 108
CPU, 108

D

deadlock, 109
debugger, 109
descripteur de fichier, 109
disjonction logique, 111
double précision, 109
DRAM, 109
dynamic RAM, 109

E

eip, 109
errno, 109
exclusion mutuelle, 109

F

fichier, 109
fichier header, 109
fichier objet, 109
fichier source, 109
FreeBSD, 109
FSF, 109

G

garbage collector, 109
gcc, 109
GHz, 109
Gnome, 109
GNU, 109
GNU is not Unix, 109
GNU/Linux, 109

H

heap, 109
hiérarchie de mémoire, 109

I

inode, 109
instruction pointer, 109
interruption, 110

K

kHz, 110

L

libc, 110
librairie dynamique, 112
librairie partagée, 112
librairie statique, 112
lien symbolique, 110
linker, 110

Linus Torvalds, 110
Linux, 110
little endian, 110
livelock, 110
liveness, 110
llvm, 110
loi de Amdahl, 110
loi de Moore, 110

M

mémoire, 110
mémoire cache, 110
MacOS, 110
makefile, 110
memory leak, 110
MHz, 110
microprocesseur, 110
Minix, 110
MIPS, 110
multi-coeurs, 110
multi-threadé, 110
multitâche, 110
multitasking, 110
multithreadé, 110
mutex, 110

N

négation, 111
nibble, 110
NOT, 110

O

octet, 108
offset pointer, 111
opération atomique, 111
OpenBSD, 111
OR, 111
ou exclusif, 113

P

pc, 109
pid, 111
pile, 112
pipe, 111
pointeur, 111
portée, 111
portée globale, 111
portée locale, 111
préprocesseur, 108
principe de localité, 111
process identifier, 111
processeur, 110
processus, 111

R

répertoire, 111
répertoire courant, 111
RAM, 111
Random Access Memory, 111
raspberry pi, 111
registre, 111
RISC, 111
root, 111
round-robin, 111

S

sémaphore, 111
sûreté, 112
safety, 112
scheduler, 111
section critique, 111
segment de données, 111
segment des données initialisées, 111
segment des données non-initialisées, 111
segment text, 112
segmentation fault, 111
shared library, 111
shell, 112
signal, 112
simple précision, 112
Solaris, 112
Solid State Drive, 112
SRAM, 112
SSD, 112
stack, 112
static library, 112
static RAM, 112
stderr, 112
stdin, 112
stdout, 112

T

table des processus, 112
tas, 109
text, 112
thread-safe, 112

U

Unicode, 112
Unix, 112
userid, 112

V

vivacité, 110
von Neumann, 112

W

warning, 112

X

X11, 112
x86, 112
XOR, 113