Licence Mention Informatique, 3ème année

Programmation des systemes – vol 1

Giuseppe Lipari

version 0.6

Table des matières

Preface

Ce document [1] est le support pour le cours de Programmation de Système au 3ème année de Licence Informatique à l'Université de Lille.

Le cours comprend des exercices (Travaux Dirigées) et des projets (Travaux Pratiques) qui sont distribués sur des documents separés.

Après une introduction générale aux systèmes d'exploitation (chapitre 1), à leur utilisation et à leur rôle dans les systèmes d'information, on aborde l'étude de l'interface POSIX. Dans le chapitre 2 on s'intéresse à la structure d'un processus, c'est-à-dire d'un programme en execution dans un système d'exploitation. Dans le chapitre 3 on étudie l'interface POSIX pour les systèmes de fichiers. Le dernier chapitre 4 traite la concurrence avec les POSIX threads et les problèmes de synchronisation via les sémaphores.

Ces arguments seront répris et approfondis dans la deuxième partie du cours (vol 2).

Chaque chapitre se termine par une série d'exercices que l'étudiant peut utiliser pour vérifier son niveau de compréhension de la matière et pour préparer les examens.

Il se peut que ce document contienne des erreurs. Vous pouvez signaler les erreurs en écrivant un email à giuseppe <dot> lipari <at> univ-lille <dot> fr en mentionnant le numéro de version que vous trouverez sur le frontispice.

Chapitre 1

Introduction

1.1 Algorithme, programme, processus, thread

On définit **algorithme** une suite finie d'opérations permettant de résoudre une classe de problèmes. Un algorithme est décrit par un formalisme approprié (appelé *langage de programmation*), de sorte qu'il puisse être exécuté sur un ordinateur. Le codage d'un algorithme, exprimé par un langage de programmation, est appelé **programme**.

Un **processus** est un programme en exécution. Si on lance le même programme plusieurs fois, on obtient plusieurs processus. Un processus peut être de type séquentiel ou peut se composer de plusieurs activités qui s'exécutent "en même temps". Dans ce dernier cas, chaque activité d'un processus est appelée **thread**.

Un processus est caractérisé par un ensemble de ressources privées, y compris l'espace mémoire. Un processus ne peut pas accéder à l'espace mémoire d'un autre processus.

Inversement, les **threads** d'un même processus partagent diverses ressources, y compris l'espace mémoire du processus auxquels ils appartiennent, de sorte qu'ils peuvent communiquer via la mémoire commune.

Dans la figure 1.1 nous voyons 3 processus en exécution sur un système d'exploitation. Les processus n. 15 et 21 sont de processus séquentiels, et le processus n. 42 contient 3 threads. Tous les threads du processus 42 partagent la mémoire entre eux. Par contre, le `threadA()` ne peut pas utiliser la mémoire des processus 15 et 21.

1.2 Système d'exploitation

Un **système d'exploitation** est un programme qui gère les ressources matérielles de la machine, y compris le processeur et la mémoire.

Un système d'exploitation réalise principalement 4 types de tâches :

1. gestion des fichiers (implantation, organisation et désignation) ;
2. gestion de la mémoire (idem) ;
3. gestion des processus et threads (gestion, création, coopération) ;
4. gestion des périphériques d'entrées-sorties et du matériel.

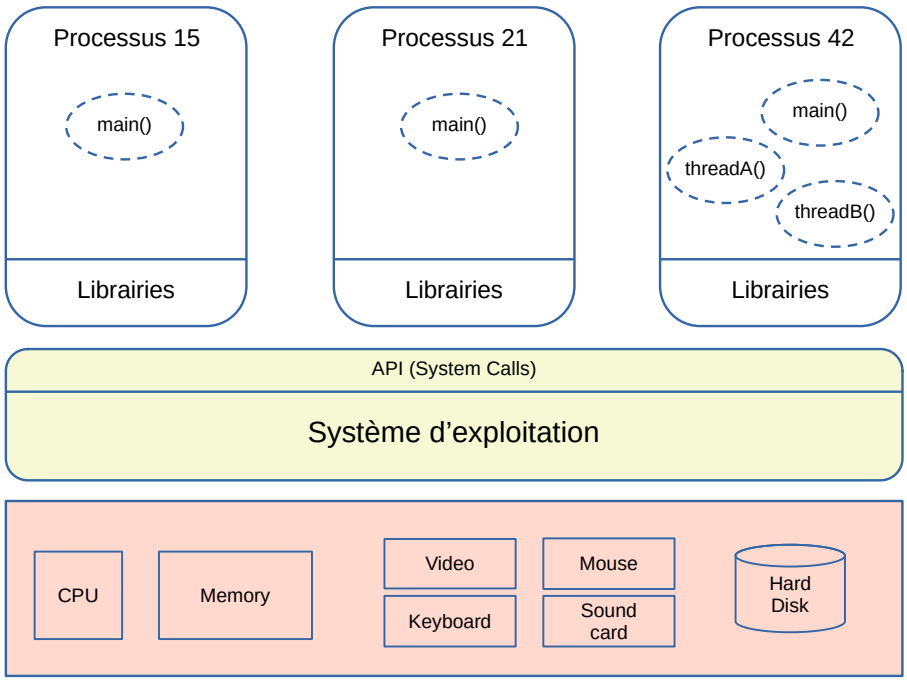


FIGURE 1.1 – Structure d'un système d'exploitation avec 3 processus.

1.2.1 Abstraction

Une **abstraction** est la généralisation d'un concept, ou d'un objet, ou d'un procédé, qui consiste dans l'élimination des détails "inutiles" en gardant les éléments essentiels.

Le matériel d'un ordinateur est souvent compliqué à gérer directement par le programmeur. Le système d'exploitation fourni une *abstraction* de la machine pour faciliter la tache aux programmeurs.

Par exemple, un programme peut sauvegarder ses données sur un disque dur ou une clé USB. Les détails matériels de deux dispositifs sont très différents, et le programmer directement demande un effort considérable. De plus, chaque modèle de disque dur a des caractéristiques différentes et nécessite d'un programme pilote différent.

Le système d'exploitation simplifie l'accès au stockage de masse en fournissant **l'abstraction de système de fichier**. Il propose ainsi une API (*Application Programming Interface*) qui permet d'ouvrir un fichier, lire et écrire des données, et fermer le fichier ensuite. Cette abstraction ne dépend pas du type de mémoire de masse qu'on utilise : ainsi l'interface est la même si on utilise un disque dur ou bien une clé USB. C'est la tâche du système d'exploitation d'implémenter l'interface pour les différents périphériques disponibles sur la machine.

1.2.2 Multiprogrammation

Le système d'exploitation fourni aussi un support pour la *multiprogrammation*, c'est-à-dire la possibilité d'exécuter plusieurs processus en même temps.

Par exemple, supposons d'utiliser une machine avec 1 seul processeur. Dans les anciens systèmes (par exemple le système MS-DOS), un seul programme à la fois pouvait s'exécuter, et on devait attendre sa termination avant de pouvoir lancer un nouveau programme.

Dans les systèmes avec multiprogrammation (e.g. Linux, Windows, MacOS, ...) les système d'exploitation permet de lancer en exécution un nombre élévé de processus, même quand le nombre de processeurs est limité. En particulier, le système d'exploitation utilise un *ordonnanceur* pour alterner l'exécution de tous les processus sur les processeurs disponibles et donner ainsi l'impression à l'utilisateur que tous les processus avancent "en même temps". De cette façon, il est possible d'exécuter $N > 4$ processus sur seulement 4 processeurs en alternant les 4 processus à exécuter pendant un intervalle de temps dit *system tick*.

Le **niveau de parallélisme** d'un système est le nombre de ressources d'exécution (processeurs) disponibles. Deux processus (out thread) s'exécutent **en parallèle** s'ils sont exécutés au même instant sur deux processeurs.

Le **niveau de concurrence** d'un système est le nombre de processus (ou de thread) qui sont actives en même temps dans le système. Les processus peuvent s'exécuter en parallèle, ou ils peuvent alterner leur exécution sur les processeurs.

Exemple 1

Par exemple, on peut avoir 50 processus actifs dans un système Linux sur une machine avec 4 processeurs : dans ce cas, on dit que le niveau de concurrence est 50, et que le niveau de parallélisme est de 4 ; les 50 processus alterne leur exécution sur les 4 processeurs tous les 10 milliseconds. Pour voir les processus actifs dans un système Linux, il suffit de saisir la commande :

```
ps axu
```

sur le terminal.

1.3 Mode utilisateur et mode superviseur

Le système d'exploitation met à disposition du programmeur un ensemble de fonctions pour manipuler et interagir avec la machine : lancer un processus, écrire des données sur la mémoire de masse, etc. Ces fonctions sont spéciales, parce qu'ils manipulent directement la machine. Normalement, un programmeur ne peut pas manipuler directement la machine, sinon un processus non autorisé risque de compromettre les données et même le bon fonctionnement du système.

Par exemple, un processus non autorisé ne peut pas manipuler directement la fréquence de fonctionnement du processeur : si non un programme quelconque, téléchargé via Internet, pourra ralentir ou même arrêter la machine. Également, un processus ne peut pas écrire directement dans la mémoire de l'interface vidéo ; il ne peut pas écrire à un certaine position dans le disque dur ; etc.

Pour prévenir l'accès indiscriminé au matériel, les processeurs modernes possèdent deux modes de fonctionnement : le mode utilisateur et le mode superviseur.

En **mode superviseur** (*supervisor mode* en anglais) le processeur peut accéder en lecture et écriture à toute la mémoire physique de la machine ; il peut écrire dans les registres des périphériques ; il peut exécuter toutes les instructions de son jeu. Le code du système d'exploitation s'exécute normalement en mode superviseur parce que le système d'exploitation doit avoir plein contrôle de toute la machine.

En **mode utilisateur** (*user mode* en anglais) le processeur peut accéder à une partie de la mémoire (la mémoire virtuelle, dont on parlera dans les chapitres à venir) ; il ne peut pas accéder directement aux registres des périphériques ; il y a des instructions qui ne sont pas exécutables. Pour de raisons de sécurité, les processus s'exécutent habituellement en mode utilisateur, pour ne pas compromettre le bon fonctionnement de la machine.

À chaque fois qu'un processus souhaite accéder à la machine, il le fait par le biais du système d'exploitation : le processus fait un **appel système** (*system call* ou *syscall* en anglais) pour exécuter une fonction du système d'exploitation. Un appel système est différent d'un normal appel de fonction : lors d'un appel système le processeur change mode de fonctionnement (*mode switch* en anglais) pour passer du mode utilisateur au mode superviseur. Ainsi, la fonction peut accéder aux périphériques pour réaliser une entrée ou une sortie de données. Quand la fonction se termine, le processeur retourne dans le code du processus en mode utilisateur.

1.4 Syscall et libcall

Souvent, l'ensemble des *syscall* exportées par le système d'exploitation, également appelé interface ou plus exactement *Application Programming Interface* (API), s'avère être trop bas niveau pour être utilisable de manière simple par l'utilisateur. Pour cette raison, des bibliothèques sont parfois proposées qui implémentent des fonctionnalités de plus haut niveau basées sur les appels système. En règle générale, une bibliothèque fournit des appels de bibliothèque (*library call*, ou plus brièvement *libcall*) qui exportent une interface plus puissante et plus simple à utiliser que celle fournie par les appels système. Vous devez garder à l'esprit qu'un appel de bibliothèque va exécuter le code de la bibliothèque en mode utilisateur, puis éventuellement appeler un ou plusieurs syscall qui exécuteront en mode superviseur.

Dans la figure 1.1 on remarque que les bibliothèques sont contenues dans l'espace de mémoire des processus : il s'agit de code qui s'exécute en mode utilisateur. Une fonction de librairie peut ensuite faire

un appel système pour exécuter du code privilégié dans la mémoire du noyau du système d'exploitation.

Un exemple typique pour clarifier ce qui vient d'être dit est la bibliothèque standard du langage C (`libc`) : entre autres choses, elle implémente les flux d'entrée/sortie, proposant un moyen simple et puissant d'accéder aux fichiers. Un fichier peut être ouvert via la libcall `fopen()` et peut être utilisé en écriture via la libcall `fprintf()`, qui permet d'écrire des nombres et des chaînes en effectuant automatiquement la conversion de format. Ces fonctions sont basées sur certains appels système (qui dans les systèmes Unix sont `open()` et `write()`) qui permettent d'accéder au fichier sans utiliser des tampons et sans conversion de format. Un utilisateur peut choisir d'accéder à un fichier via les appels système ou les appels de `libc`, mais les deux types d'accès ne peuvent pas être arbitrairement mélangés (par exemple, vous ne pouvez pas faire un `fprintf()` sur un fichier ouvert avec `open()`).

1.5 Opérations atomiques

Une **opération atomique** est une opération (ou une séquence d'instructions) qui s'exécute entièrement sans pouvoir être interrompue avant la fin de son déroulement. Se terme dérive du grec ancien *atomos* qui signifie « que l'on ne peut diviser ».

Les opération atomiques sont nécessaire pour gérer correctement les processus concurrents. Il peut s'avérer que deux processus essayent de faire des opérations sur la même ressource en même temps. Si les deux opérations ne sont pas correctement synchronisées, des erreurs logiques peuvent se produire. On verra cette problématique dans le chapitre 4. Pour l'instant, on se limite à remarquer que les syscalls sont des opérations atomiques.

1.6 Notes générales sur Unix

Dans ce qui suit, nous analyserons les syscall disponibles sur un système Unix (comme Linux), et nous verrons en particulier comment créer différents flux d'exécution au sein d'un programme, comment les synchroniser et comment réaliser la communication entre les threads et les processus. Ces syscall dépendent du système d'exploitation, et nous nous référerons ici aux systèmes d'exploitation Unix-like (Linux en particulier), selon la norme POSIX.

Bien que le support de multiprogrammation fourni par le système soit exploitable en utilisant différents langages de programmation, nous utiliserons ici le langage `C`. Nous supposons que le lecteur est familier avec les bases de ce langage et avec les bibliothèques standard qu'il fournit. Dans tous les cas, une brève introduction sur la structure d'un programme C sera faite dans la section 2 et une brève présentation de la bibliothèque I/O standard au dans la section 3.7.

Toutes les *syscall* retournent un entier, qui en cas de valeur négative signale une condition d'erreur : il est alors utile de vérifier que cette valeur est supérieure ou égale à 0 chaque fois qu'on invoque une syscall, en utilisant par exemple un morceau de code comme le suivant :

```
int res;

res = syscall(...);
if (res < 0) {
  perror("Error calling syscall");
  exit(-1);
}
// Program continues here...
```

La fonction `perror()` envoie sur la sortie d'erreur standard (typiquement le terminal) un message d'erreur composé de la chaîne passée comme paramètre suivi d'une description de la raison pour laquelle le dernier appel syscall a échoué.

Les prototypes des systèmes d'appels et les définitions des constantes et des structures de données nécessaires sont contenus dans certains en-têtes de fichiers système (généralement dans le répertoire `/usr/include/sys`) ; pour pouvoir utiliser un syscall donné, il faut donc inclure les en-têtes appropriés. Les systèmes Unix fournissent la commande `man`, qui donne une brève description de la sémantique d'un syscall spécifié, ainsi que la syntaxe et la liste des en-têtes à inclure pour utiliser ce syscall. Toujours se référer aux pages de manuel pour inclure les fichiers corrects et utiliser une syscall avec la bonne syntaxe.

Chapitre 2

Processus

2.1 Programme et Processus

Un **programme** est un fichier exécutable résidant sur le disque. Il peut être exécuté à l'aide d'une commande *shell* ou en appelant une fonction `exec()`. Un **processus** est une instance du progamme qui s'exécute [1]. Chaque processus a un identifiant unique dans le système, appelé **process ID**, qui est un entien non négatif.

Si nous lançons deux fois le même programme, nous obtenons deux processus distincts, chacun avec son propre ID de processus. Un processus peut obtenir son ID en invoquant la primitive `getpid()`. Dans le chapitre 4, nous présenterons mieux le concept de processus dans le contexte de la multiprogrammation. Dans ce chapitre, nous allons nous concentrer sur la mise en page de la mémoire d'un processu, et ses interactions avec le système d'exploitation.

Étant donné que le langage le plus utilisé dans les systèmes Unix est le C, et que tous les prototypes des appels système sont exprimés en C, il vaut la peine de faire quelques références sur la structure d'un programme écrit en C.

2.2 Structure d'un programme C

Chaque programme écrit en C a un point d'entrée qui est la fonction `main`. Son prototype est :

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

Le premier paramètre représente le nombre d'arguments passés au programme sur la ligne de commande, tandis que le second paramètre est le pointeur vers un tableau de chaînes contenant les arguments réels. Le premier argument (contenu dans la chaîne `argv[0]`) est toujours le nom du programme. Donc `argc` vaut toujours au moins 1.

Un processus peut se terminer de 5 façons différentes, trois sont normales et deux sont anormales :

- Terminaison normale :
  - en exécutant l'inscription return du main,
  - en appelant `exit()`,

1. Parfois un processus est appelé /task/ : comme task est un terme plus générique, nous préférons d'utiliser le terme processus.

  - en appelant `_exit()`.
- Terminaison anormale :
  - en appelant directement `abort()`,
  - en recevant un signal non géré.

Les prototypes des deux fonctions `exit()` sont énumérés ci-dessous :

```
#include <stdlib.h>
void exit(int status);
```

```
#include <unistd.h>
void _exit(int status);
```

La `exit()` fait un "nettoyage" de l'état du processus avant de quitter (généralement, elle ferme tous les fichiers et descripteurs ouverts), tandis que `_exit()` revient brutalement au système. Il est également possible d'installer des fonctions à appeler avant la sortie via la fonction `atexit()`.

```
#include <stdlib.h>

void atexit(void (*func)(void));
```

Voici un exemple d'utilisation :

```
#include <stdlib.h>

void myexit(void);

int main()
{
  printf("Hello world\n");
  atexit(myexit);
  printf("Just before exiting ...\n");
  return 0;
}
```

La séquence des appels effectués lors du lancement d'un processus et pendant son exécution est résumée dans la Figure 2.1 : lorsqu'un processus est lancé, une procédure d'initialisation (non visible par l'utilisateur) appelée *C Startup Routine* est exécuté. Ensuite, la fonction `main()` est appelée : si cette fonction quitte avec `return`, vous revenez à *C Startup Routine*, qui appelle `exit()` en lui transmettant la valeur de retour obtenue par le `main()`.

La fonction `main()` (ou l'une des fonctions appelées par `main()`) peut à son tour invoquer directement la fonction `exit()` ou `_exit()`. La différence est que `exit()` fait un certain nombre de choses, parmi lesquelles invoquer les *handlers*, c'est-à-dire les fonctions qui ont été installées avec la `atexit()`. Enfin, les `_exit()` et `exit()` retournent au noyau, qui effectue un travail de nettoyage supplémentaire, en supprimant presque toutes les structures internes liées au processus. Dans le chapitre 3, nous verrons qu'en réalité certaines structures restent toujours présentes jusqu'à ce que le processus parent appelle la `wait()`.

2.3 Variables d'environnement

Chaque interpréteur dispose d'un moyen de définir des variables d'environnement. Sur `bash` :

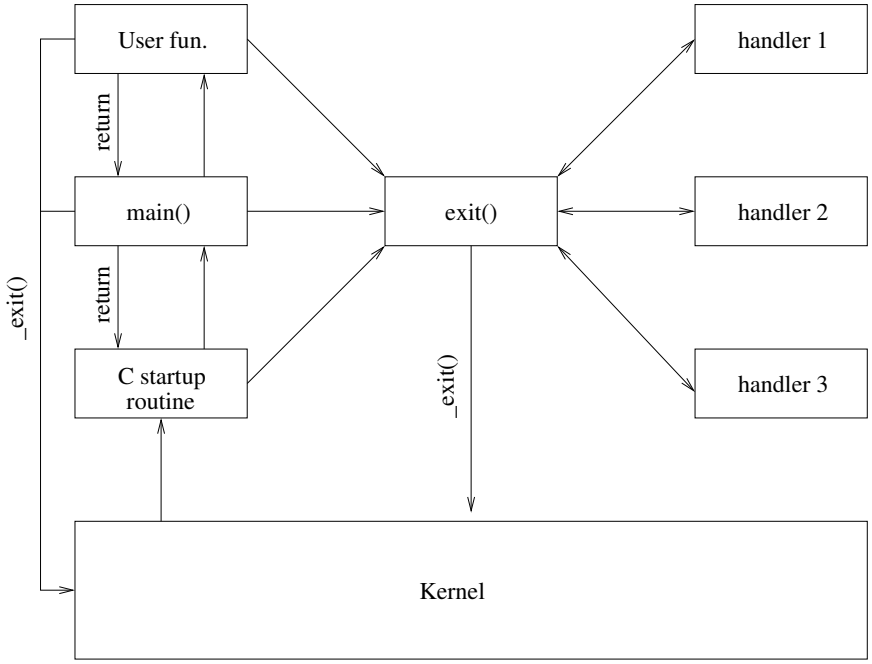


FIGURE 2.1 – Séquence d'appels effectués à l'entrée et à la sortie d'un processus.

```
export VARNAME=valeur
```

Par exemple, une variable d'environnement typique est la variable `PATH` : il s'agit d'une séquence de répertoires séparés par le caractère `:` ; chaque fois que vous tapez une commande l'interpréteur recherche le fichier exécutable dans les dossiers répertoriés dans le `PATH`. Pour lister toutes les variables définies et leur contenu, sur la shell `bash` il suffit de saisir la commande `export`.

Dans un programme, vous pouvez obtenir la liste des variables d'environnement, qui sont stockées comme un tableau de chaînes, via la fonction `getenv()` :

```
#include <stdlib.h>

char *getenv(const char *name);
```

Enfin, il est possible de modifier une variable d'environnement à l'aide des appels de fonction suivants, au sens assez intuitif :

```
int setenv(const char *name, const char *value, int rewrite);
void unsetenv(const char *name);
int putenv(const char *str);
```

2.4 Disposition de mémoire d'un processus

Chaque processus a son propre espace d'adressage privé et non visible de l'extérieur. Cela signifie que deux processus ne peuvent pas accéder à la même zone de mémoire [2]. Cette séparation totale des espaces d'adressage est obtenue en exploitant les caractéristiques matérielles du processeur (segmentation, protection de la mémoire, etc.).

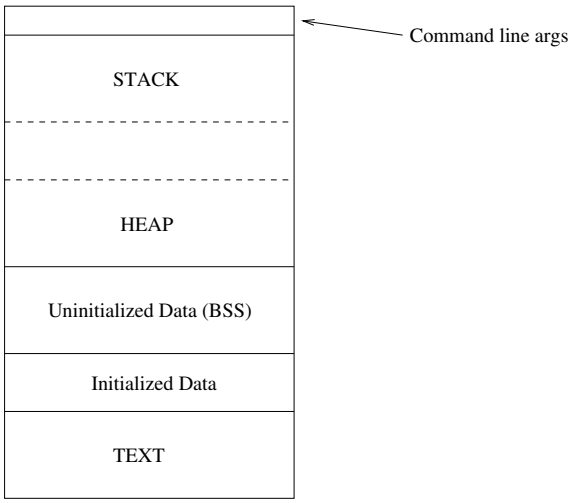


FIGURE 2.2 – Disposition de mémoire d'un processus

2. C'est en fait possible via la primitive `mmap`, qui n'est pas présentée dans ce cours pour des raisons de temps.

L'espace d'adressage d'un processus est généralement organisé logiquement comme dans la figure 2.2. Bien sûr, la mise en page exacte dépend du processeur utilisé et des conventions du système d'exploitation et du compilateur C. Par exemple, dans certains systèmes, l'index au sommet de la pile augmente vers le haut, contrairement à ce qui est montré sur la figure.

Nous retrouvons :

— une zone de mémoire (ou *segment*) TEXT qui contient le code du programme,
— un segment de données initialisées,
— un segment de données non initialisées (souvent appelé BSS) est habituellement initialisé par le chargeur à zéro ;
— un segment commun entre la pile (STACK) et le tas (HEAP).

En particulier, le tas est la zone de mémoire à longueur variable dans laquelle sont attribuées les zones de mémoire que l'utilisateur alloue dynamiquement avec les fonctions `malloc` et `free` :

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void * ptr);
```

Ces fonctions ne sont pas des appels de système d'exploitation (syscall), mais sont implémentées dans la bibliothèque C standard, et donc dans l'espace utilisateur (libcall). Elles s'appuient cependant sur des appels de système (typiquement la `sbrk()`), qui permettent d'élargir ou de restreindre la taille du tas.

Vous devez être très prudent dans l'utilisation de ces deux fonctions ! Un problème assez important est que les deux fonctions en question *ne sont pas re-entrants*, donc elles ne devraient pas être utilisées dans un gestionnaire de signal, et devraient être utilisées avec précaution en présence de signaux.

2.5 Questions et exercices

2.5.1 Variable d'environnement

Qu'est-ce que c'est une variable d'environnement ? Donnez un exemple.
(solution à la page 55).

2.5.2 Vérifier l'existence d'un fichier

Écrire un programme qui vérifie que le fichier `.bashrc` existe dans le répertoire `HOME` de l'utilisateur et il est accessible en lecture.

Pour vérifier l'accessibilité en lecture d'un fichier, utilisez la fonction `access()`. Vous pouvez utiliser la commande

```
man 2 access
```

sur le terminal pour obtenir des informations sur cette fonction.
(solution à la page 55).

Chapitre 3

Système de fichiers

3.1 Notions fondamentales

Les systèmes Unix sont des systèmes multi-utilisateurs. Chaque utilisateur activé dans le système dispose d'un certain nombre de ressources (un certain espace disque, la possibilité d'exécuter certains programmes, etc.), appelées génériquement **comptes**. Chaque utilisateur du système est identifié par :

User ID est un entier unique et non négatif. L'utilisateur avec un ID égal à 0 est le *superuser*, c'est-à-dire l'administrateur système.

Login Name est une chaîne également unique dans le système. Le nom d'utilisateur du super utilisateur est `root`, donc nous utiliserons soit `root` soit superutilisateur pour indiquer l'administrateur système.

Mot de passe chaque utilisateur dispose d'un mot de passe qui lui est demandé lors de la connexion au système. Ce n'est pas nécessairement unique, dans le sens où deux utilisateurs peuvent avoir le même mot de passe.

Group ID identifie le groupe auquel l'utilisateur appartient. Appartenir à un certain groupe plutôt qu'à un autre permet certaines autorisations, comme nous le verrons ci-dessous.

Home Directory est le répertoire où sont conservés les fichiers privés de l'utilisateur.

Shell Initial est le *shell* (ou interpréteur de commandes) qui est lancé après la procédure de connexion.

Un utilisateur peut accéder au système via la *procédure de connexion*. Par exemple, lorsque l'utilisateur se connecte à un terminal connecté au système, il se présente un écran où il est invité à entrer le *login name* et plus tard le *password*. Si le nom ainsi que le mot de passe saisis correspondent à l'un des comptes enregistrés dans le système, alors le processus *shell* est exécuté et se positionne sur le répertoire initial. À ce stade, l'utilisateur se trouve avec un *prompt* qui signale la disponibilité du programme *shell* à accepter des commandes, comme obtenir la liste des fichiers du répertoire courant avec le programme `ls`, lancer un programme utilisateur, et ainsi de suite.

Les informations énumérées ci-dessus (à l'exception du mot de passe) se trouvent dans un fichier nommé `/etc/passwd`, qui ne peut être consulté en lecture et en écriture que par `root`. Un exemple de fichier `passwd` est montré ci-dessous : chaque ligne contient des informations pour chaque utilisateur séparées par le caractère ' :'. Les seuls utilisateurs "vrais" dans l'exemple sont `root`, `lipari` et `claire`.

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/adm:
lp:x:4:7:lp:/var/spool/lpd:
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:
news:x:9:13:news:/var/spool/news:
uucp:x:10:14:uucp:/var/spool/uucp:
operator:x:11:0:operator:/root:
games:x:12:100:games:/usr/games:
gopher:x:13:30:gopher:/usr/lib/gopher-data:
ftp:x:14:50:FTP User:/home/ftp:
nobody:x:99:99:Nobody:/:
xfs:x:43:43:X Font Server:/etc/X11/fs:/bin/false
lipari:x:500:500::/home/lipari:/bin/bash
elena:x:0:0:claire:/home/claire:/bin/bash
```

Notes

Dans les anciens systèmes, les mots de passe des utilisateurs étaient également stockés dans le fichier `/etc/passwd` cryptés par un algorithme unidirectionnel. Ainsi, lorsque l'utilisateur saisissait le mot de passe, la procédure de connexion appliquait l'algorithme en obtenant une chaîne qui comparait avec le contenu le champ approprié du fichier `/etc/passwd`, alors qu'il était impossible de remonter au mot de passe original.

Cependant, un pirate, dans le but de découvrir le mot de passe de certains utilisateurs, pouvait toujours essayer d'appliquer l'algorithme pour crypter (qui est public) sur un certain nombre de mots de passe possibles et comparer les résultats avec le contenu du fichier. Avec la puissance des ordinateurs modernes, ce n'était qu'une question de temps avant de pouvoir obtenir le mot de passe du `root`.

Ainsi, pour plus de sécurité, dans les systèmes modernes, les mots de passe et d'autres informations sont contenus dans le fichier `etc/shadow` (dans certains systèmes appelés `master.passwd`) qui ne peut être lu ou écrit que par `root`, et donc l'attaque décrite ci-dessus n'est pas possible.

3.2 Fichiers

Un nom de fichier est une chaîne de caractères qui peut contenir tous les caractères exclus '/' et \0. Traditionnellement, le système de fichiers est organisé en arborescence :

— il existe un répertoire racine ("/") ;

— chaque répertoire peut contenir des fichiers ou d'autres répertoires de manière récursive.

Chaque fichier est contenu dans un seul répertoire, appelé répertoire parent du fichier. Dans l'image 3.1, il y a un exemple de structure de répertoire prise à partir d'une célèbre distribution Linux.

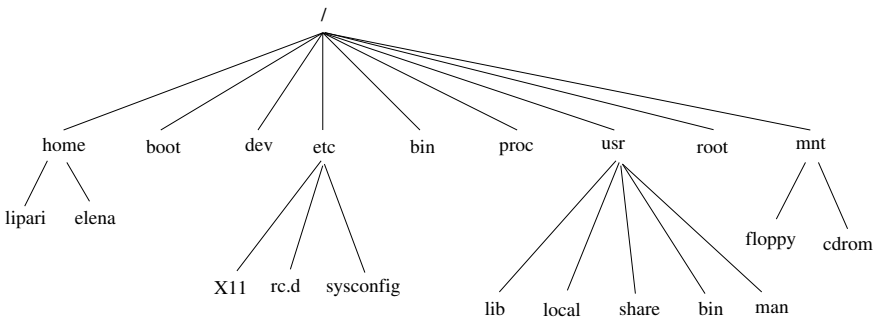


FIGURE 3.1 – Exemple de système de fichier

Sous Unix, un répertoire est un fichier spécial qui contient une liste des fichiers contenus dans le répertoire [1]. Le répertoire est lisible par tous ceux qui ont la permission en lecture sur le répertoire, mais il n'est accessible en écriture que par le noyau via les primitives appropriées, afin de garantir la cohérence du système de fichiers.

Le **full pathname** (chemin absolu) d'un fichier est la séquence des répertoires, à partir de la racine, pour arriver au fichier. Donc un chemin absolu commence toujours par '/'.

Le **current working directory** (cwd) est le répertoire courant d'un processus. Comme l'interpréteur est un processus, il possède un cwd que vous pouvez changer avec la commande shell `cd`. Lorsqu'un utilisateur entre dans le système, la procédure de connexion lance un interpréteur de commandes dont le cwd est le répertoire d'accueil de l'utilisateur. Pour obtenir le pathname de la cwd, il y a la commande `pwd`.

Le **relative pathname** (chemin relatif) d'un fichier est la séquence des repertoires, à partir du courant, pour arriver au fichier. Dans un **pathname** (*full* ou *relative*), le symbole `.` indique le repertoire courant et le symbole `..` le répertoire père.

Example 3.2.1 Si le cwd est `/home/lipari/bin`, le chemin `../rtlib/src/task.c` indique le fichier `task.c` dans le répertoire `/home/lipari/rtlib/src/`, alors que le chemin `./killns.ps` indique le fichier `killns.pl` dans le répertoire courant `/home/lipari/bin`. ■

3.3 La bibliothèque POSIX pour les fichiers

Dans cette section et dans les sections suivantes nous décrirons la bibliothèque standard POSIX pour les fichiers.

Un **descripteur de fichier** est un entier non négatif qui identifie un fichier ouvert par un processus. Chaque processus possède une table des descripteurs qui associe chaque descripteur ouvert par le processus à un fichier. Habituellement, le descripteur 0 correspond au fichier *standard input*, le descripteur 1 au fichier *standard output* et le descripteur 2 au fichier *standard error*. Pour des besoins de standardisation, POSIX définit les macros `STDIN_FILENO`, `STDOUT_FILENO` et `STDERR_FILENO` pour ces 3 descripteurs, de

1. Ce que ce fichier spécial contient dépend de la mise en œuvre et il n'y a aucune norme à cet égard.

sorte que, si ces 3 identificateurs devaient prendre d'autres valeurs dans le futur, les anciens programmes peuvent encore être utilisés sans modification, après une recompilation.

La fonction `open()` ouvre un fichier :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fnctl.h>

int open(const char *pathname, int oflag, ...);
```

Le paramètre `oflag` détermine le mode d'ouverture du fichier et est l'OR parmi les valeurs répertoriées dans le tableau 3.1. Quelques considérations méritent d'être soulignées :

— si le fichier est ouvert en append, une opération d'écriture implique toujours le déplacement à la fin du fichier avant l'écriture.

— si le fichier est ouvert avec le drapeau `O_SYNC`, chaque opération de write se bloque jusqu'à ce que les données aient été réellement écrites sur le disque. Ceci est particulièrement utile lors de la programmation de bases de données, et vous voulez être sûr que l'opération d'écriture est réussie.

TABLE 3.1 – Options pour la `open()`

Macro	**Mode**
O_RDONLY	lecture seule
O_WRONLY	écriture seule
O_RDWR	lecture et écriture
O_APPEND	en append
O_CREAT	crée le fichier s'il n'existe pas
O_EXCL	le fichier est fermé dans le cas d'une exec
O_TRUNC	la taille du fichier est mis à 0
O_NONBLOCK	toutes les opérations sont non-bloquantes
O_SYNC	la `write()` attend que les données soient écrites sur le support avant de retourner

Pour lire et écrire à partir d'un fichier, utilisez les primitives `read()` et `write()` :

```
#include <unistd.h>

ssize_t read(int filedes, void *buff, size_t nbytes);
ssize_t write(int filedes, const void *buff, size_t nbytes);
```

Les deux renvoient le nombre réel d'octets lus/écrits. Enfin, la fonction `lseek()` vous permet de déplacer le pointeur actuel du fichier vers n'importe quel emplacement :

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

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

Le paramètre `offset` peut prendre comme valeurs qui sont interprétées en fonction du paramètre `whence`, comme indiqué ci-dessous :

SEEK_SET deplace le pointer à offset octets à partir du debut ;

SEEK_CUR deplace les pointeurs à offset octets à partir de la position courante ;

SEEK_END deplace les pointeurs à offset octets à partir de la position finale.

La fonction retourne la position absolue atteinte après l'achèvement de l'opération.

Example 3.3.1 L'instruction suivante :

```
currpos = lseek(fd,0,SEEK_END);
```

déplace le pointer au fichier identifié par le descripteur `fd` à la fin du fichier. La variable `currpos` contient maintenant la taille courante du fichier.

— Pour obtenir la position courante :

```
    currpos = lseek(fd,0,SEEK_CUR);
```

— Pour se deplacer au début du fichier :

```
    currpos = lseek(fd,0,SEEK_SET);
```

— Pour avancer de 10 octets :

```
    currpos = lseek(fd,10,SEEK_CUR);
```

— Pour reculer de 10 octets :

```
    currpos = lseek(fd,-10,SEEK_CUR);
```

Il faut noter que, si on avance et on écrit après la fin du fichier, un *trou* (*hole* en anglais) sera créé, qui correspond à des 0 dans le fichier.

Par exemple, dans l'exemple suivant on crée un trou de 100 octets dans le fichier.

```
lseek(fd, 0, SEEK_END);   /* position à la fin du fichier         */
lseek(fd, 100, SEEK_CUR); /* encore 100 octets en avant           */
write(fd, "Hello", 6);    /* l'écriture crée un trou de 100 zéros */
```

Par contre, il n'est pas possible de reculer avant la position 0, la position courant doit être toujours un entier non-negative. Un appel à `lseek()` essayant de remonter la position avant 0 retourne une erreur. ■

3.4 Structures de données impliquées

Pour mieux comprendre ce qui se passe dans le système d'exploitation lorsqu'une opération est effectuée sur un fichier, il faut analyser les structures de données impliquées.

Tout d'abord, chaque processus a son propre **tableau de descripteurs de fichiers** : un exemple est montré dans la figure 3.2, où le processus A a ouvert 2 fichiers *montexte.txt* et *data.txt* (avec numéro de descripteur 3 et 4, respectivement) et le processus B a ouvert seulement fichier *data.txt* (avec numéro de descripteur 3). Commençons donc par noter que le numéro 3 a un significat différent dans chaque processus : dans le processus A, il représente le fichier *montexte.txt* pendant que dans le processus B il représente le fichier *data.txt*.

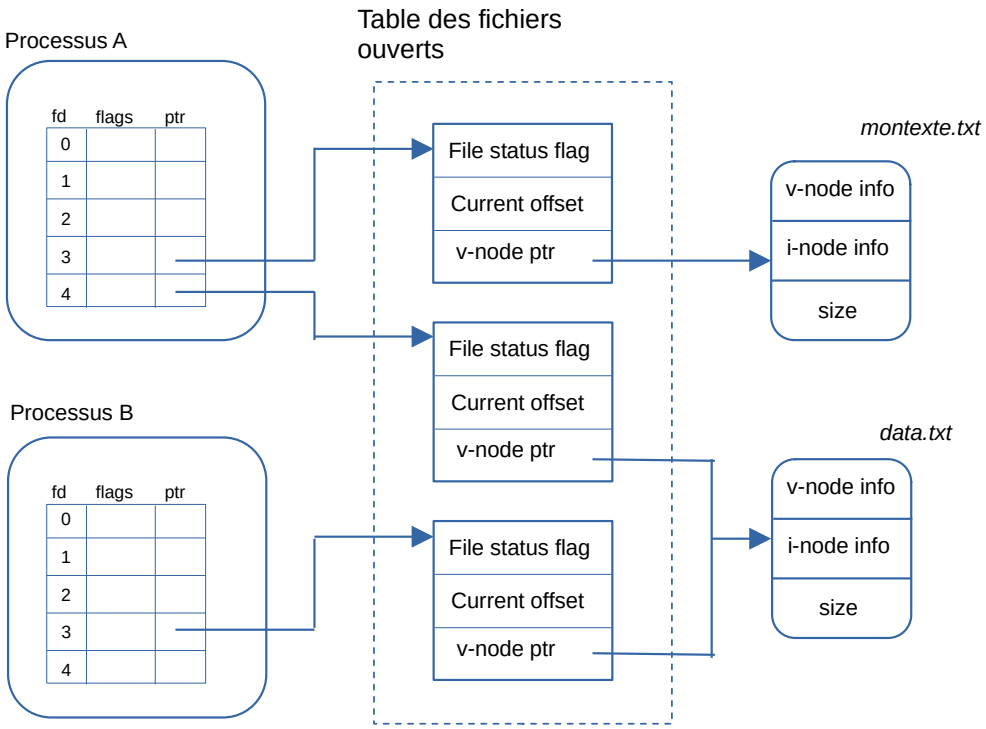


FIGURE 3.2 – Structures de données dans le noyau

Dans le noyau, il y a un **tableau des fichiers ouverts** qui contient des structures des données contenant des informations sur le fichier ouvert. Chaque fois qu'un processus exécute un `open()`, une entrée dans le tableau des fichiers ouverts et une entrée dans le tableau des descripteurs de processus est créée. Notez que des informations telles que le mode d'ouverture (read/write, append, etc.) et la position actuelle sont stockées dans l'entrée correspondante du tableau des fichiers ouverts et non dans le tableau des descripteurs.

Il y a une troisième structure de données, indiqué comme *v-node* qui contient des informations sur le fichier, comme sa taille. Noter que le fichier *data.txt* a un seul *v-node* mais deux entrées dans le tableau des fichiers ouverts, une pour chaque opération `open()` sur le fichier.

Example 3.4.1 Considerez les deux opérations suivantes faites par le processus A dans la figure 3.2 :

```
fd = open("data.txt", O_WRONLY | O_APPEND);
write(fd, "Hello", 6);
```

Comme le fichier "data.txt" est ouvert en modalité `O_APPEND`, l'opération `write()` sur le descripteur `fd` se traduit dans les étapes suivantes :

— la position `current offset` dans l'entrée correspondante dans le tableau des fichiers ouverts devient égal à `size` ;

— la chaine de caractères "Hello" est écrite dans le fichier à partir de la position courante.

Le deux étapes (déplacement du pointer et écriture) sont faites de manière *atomique*, c'est-à-dire qu'une deuxième opération sur le même fichier ne peut pas se faire entre les deux étapes. Dans ce cas, nous sommes sûrs que l'écriture sera toujours faite à partir de la fin du fichier.

Supposons maintenant que le processus `A` fait les opérations suivantes à la place :

```
fd = open("data.txt", O_WRONLY);
lseek(fd, 0, SEEK_END);
write(fd, "Hello", 6);
```

Dans ce cas, il n'y a aucune garantie que la chaine "Hello" sera écrite en fin de fichier, parce que les opérations de déplacement et écritures sont faites séparément dans deux appels différents.

Pour comprendre le problème, considérez la situation suivante :

— le processus A fait la `open()` et la `lseek()` ; le champ `current offset` dans le deuxième fichier ouvert devient égale au champ `size` dans le v-node ;
— le processus B ouvre le même fichier *data.txt* et écrit quelque chose en fin de fichier ; par conséquent, le champ `size` est augmenté ;
— le processus A maintenant fait la `write()` en écrivant à partir sa `current position` qui n'est plus à la fin du fichier. Il écrase donc les données écrites par le processus B !

■

3.5 Directory et liens symboliques

Un **répertoire** (*directory* en anglais) est un fichier spécial qui contient la liste des fichier contenus. Un programme peut lire le contenu d'un répertoire avec les fonctions `opendir` et `readdir` :

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

DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
```

Le prémier syscall retourne un pointer vers une structure opaque `DIR` qui est utilisé dans la readdir pour lire le contenu du répertoire de manière sequentiel.

Example 3.5.1 Voici un morceau de code utile pour lire un répertoire.

```
    DIR *d = opendir(argv[1]);
    assert(d != NULL);

    struct dirent *s = readdir(d);
    while (s != NULL) {
        printf("%s\n", s->d_name);
        s = readdir(d);
    }
    closedir(d);
```

Veuillez noter que, parmi les éléments listés par `readdir` on trouve `.` et `..`, c'est à dire des *liens* vers le répertoire courant et le répertoire parent, respectivement. ■

Un **lien symbolique** est un fichier spécial qui contient une référence vers un autre fichier. La commande de l'interprète `shell` pour créer un lien symbolique est le suivant :

```
ln -s <target> <lien>
```

Il correspond à appeler la syscall `link` :

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
```

Si on effectue une opération de `open` sur un lien symbolique, le fichier original est ouvert, et les opérations de lecture et écriture se feront sur le fichier référencé.

Pour effacer un lien ou un fichier du système de fichiers, la syscall à utiliser est `unlink` :

```
#include <unistd.h>

int unlink(const char *pathname);
```

Regarder la page du manual pour plus d'informations.

3.6 Statistiques

Pour avoir des informations sur un fichier, on utilise les fonctions de la famille `stat` :

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

int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
```

La syscall `stat` retourne les informations sur le fichier `pathname` (s'il existe) dans une structure `struct stat` qui contient les champs suivants :

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    gid_t     st_gid;         /* Group ID of owner */
    dev_t     st_rdev;        /* Device ID (if special file) */
    off_t     st_size;        /* Total size, in bytes */
    blksize_t st_blksize;     /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */
};
```

Le champ `st_mode` contient des informations sur le type de fichier. Pour obtenir ces informations, nous pouvons utiliser des macros comme dans le morceau de code suivant :

```
struct stat sb;
int r = stat(name, &sb);
if (r != -1) {
    if (S_ISDIR(sb.st_mode))
        printf("directory\n");
    else if (S_ISLNK(sb.st_mode))
        printf("symlink\n");
    else if (S_ISREG(st.st_mode))
        printf("regular file\n");
    else
        printf("other type of file\n");
 }
```

La structure contient aussi des informations sur la taille du fichier (taille en octets, nombre de blocs utilisés sur le disque, etc.).

La fonction `fstat` fait la même que `stat`, sauf qu'elle agit sur un fichier ouvert, et donc elle demande en premier paramètre un descripteur de fichier.

La fonction `lstat` fait la même chose que `stat`, sauf que, dans le cas d'un lien symbolique, elle retourne les informations sur le lien et non sur le fichier référencé.

3.7 La librairie standard libc

Le comité de standardisation ANSI a formalisé une norme pour faire de l'entrée/sortie tamponné qui va sous le nom d'I/O Standard Libary `stdio`. Vous pourrez retrouver la même bibliothèque sur les différents versions de UNIX et sur les systèmes Windows. Cette bibliothèque fournit l'accès à des structures qui tentent de faciliter et d'optimiser les entrées/sorties.

Les fonctions de la bibliothèque standard utilisent toutes une structure appelée `FILE`. L'accès à cette structure se fait toujours par pointeur et en utilisant les fonctions appropriées : l'utilisateur ne devrait jamais accéder directement à ses champs. Un tel type de données est souvent appelé "type opaque".

Pour ouvrir un fichier, on utilise la fonction `fopen()` :

```
#include <stdio.h>

FILE *fopen(const char *pathname, const char *type);
```

Le paramètre `type` peut être `r` (si nous voulons les ouvrir en lecture), `w` (si nous voulons ouvrir en écriture) ou `a` (si nous voulons ouvrir en append). S'il y a un `+` dans la chaîne, cela signifie que le fichier est ouvert en read/write. Donc `r+` est équivalent à `w+`. Nous pouvons également spécifier un `b` pour signifier que le fichier en question est binaire.

Notes

Sur UNIX, il n'y a pas de différence entre les fichiers texte et binaires. Sur DOS/Win, il y a une grande différence ! En effet, dans les fichiers texte, le caractère de retour panier (CR) sur UNIX est remplacé par deux caractères sur DOS/Win (CR + LF). Ainsi, lorsque vous transférez un fichier d'un système DOS/Win vers un système Unix et vice versa, vous devez faire attention au type de fichier. Le drapeau `b` dans les systèmes UNIX ne sert à rien, mais il est utilisé pour la compatibilité avec les systèmes DOS.

La fonction `fopen()` renvoie un pointeur vers une structure FILE qui peut être utilisé dans les appels suivants.

Toutes les fonctions de `stdio` génèrent un tampon d'entrée/sortie : dans la structure `FILE` un tampon d'une certaine taille y est alloué (`BUFSIZ`). La première fois que le programme lit un caractère à l'aide de l'une des fonctions de `stdio`, il lit en réalité un nombre de caractères égal à la taille de la mémoire tampon : les lectures ultérieures des caractères se traduiront par des lectures de la mémoire tampon. Il en va de même lorsqu'un programme écrit un caractère : d'abord il est écrit dans le tampon : quand le tampon est effectivement plein (ou après une opération de `flush`), son contenu est effectivement écrit sur le disque.

Ce mécanisme vise à minimiser le nombre d'accès au support physique qui sont généralement lents et coûteux, en essayant de lire/écrire les données "à blocs".

Un fichier peut être mis en mémoire tampon de 3 façons :

FULLY BUFFERED le transfert de données se fait généralement par blocs de la taille de la mémoire tampon (à moins que des opérations de vidange explicites ne soient effectuées).

LINE BUFFERED le transfert de données se fait à "lignes", c'est-à-dire qu'une ligne de texte est lue/écrite à la fois (la ligne se termine par CR).

UNBUFFERED Chaque transfert de données de/vers le tampon entraîne un transfert de données de/vers le disque.

Les 3 fichiers `stdin`, `stdout` et `stderr` sont respectivement line-buffered, line-buffered et unbuffered, tandis qu'un fichier normal ouvert en lecture ou en écriture est généralement fully buffered.

Comme anticipé, il est possible de forcer la "vidange" d'un tampon d'un fichier en écriture via la fonction `fflush` :

```
#include <stdio.h>

int fflush(FILE *fp);
```

Les fonctions suivantes permettent de lire un caractère à la fois :

```
#include <stdio.h>

int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void);
```

La première et la dernière sont en fait des macros (pour des raisons d'efficacité). Si la fin du fichier est atteinte, la constante EOF est renvoyée. Dans de nombreux systèmes EOF est défini comme -1 qui est également le code d'erreur. Pour distinguer les deux situations, on utilise les deux fonctions `feof()` et `ferror()`, au sens assez évident.

```
#include <stdio.h>

int ferror(FILE *fp);
int feof(FILE *fp);
```

Les fonctions suivantes font la sortie de caractère.

```
#include <stdio.h>

int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
```

Bien sûr, il est possible de faire des E/S par lignes : #Ovviamente, è possibile fare I/O a linee : #include <stdio.h>

```
char * fgets(char *buf, int n, FILE *fp);
char *gets(char *buf);
int fputs(const char *str, FILE *fp);
int puts(const char *str);
```

Et enfin, vous pouvez faire des E/S par blocs :

```
#include <stdio.h>

size_t fread(void *buf, size_t size, size_t nobj, FILE *fp);
size_t fwrite(void *buf, size_t size, size_t nobj, FILE *fp);
```

Notez que l'utilisation de l'une de ces fonctions n'a rien à voir avec le type de mise en mémoire tampon du fichier. Par exemple, vous pouvez utiliser la fonction `fgetc()` avec un fichier *line-buffered* et la fonction `fwrite()` avec un fichier *unbuffered* : la bibliothèque s'occupera de traduire les appels de fonctions dans les transferts de/vers le disque, en fonction du type de mise en mémoire tampon du fichier.

Enfin, les fonctions d'entrée/sortie formatées des chaînes :

```
#include <stdio.h>

int printf(const char * format, ...);
int fprintf(FILE *fp, const char * format, ...);
int sprintf(char *str, const char * format, ...);

int scanf(const char * format, ...);
int fscanf(FILE *fp, const char * format, ...);
int sscanf(char *str, const char * format, ...);
```

3.7.1 Relation entre interface POSIX et librairie ANSI

Avec un pointeur `FILE` vous pouvez obtenir le descripteur de fichier correspondant, et vice versa :

```
int fileno(FILE *fp);
FILE *fdopen(int fd);
```

Il faut dire que, alors que les fonctions standard POSIX pour l'accès aux fichiers sont primitives du système d'exploitation, et donc en tant que telles atomiques, les fonctions de bibliothèque ANSI sont de simples fonctions de bibliothèque mises en œuvre dans l'espace mémoire du processus appelant, Elles peuvent être coupées en deux. En outre, la plupart d'entre eux sont /non rentrants /, c'est-à-dire qu'ils utilisent des structures globales (par exemple, le tampon contenu dans la structure `FILE`).

Cela peut poser quelques problèmes : par exemple, il est bon de ne pas utiliser ces fonctions dans le contrôleur d'un signal ! Par exemple, supposons que vous utilisiez un `printf()` dans le gestionnaire d'un signal et dans le programme : si le signal interrompt l'exécution du `printf()`, le résultat est imprécis ! En effet, certaines des structures globales utilisées par `printf()` peuvent être dans un état inconsistant lorsque la `printf()` est appelée depuis l'intérieur du signal handler. Pour plus de détails, voir le chapitre sur la synchronisation des processus.

3.8 Questions et exercices

3.8.1 Calcul de la taille d'un fichier

Il y a plusieurs manières d'obtenir la taille d'un fichier. Décrire les méthodes que vous connaissez.
(Solution à la page 56).

3.8.2 Question : fgetc

La librairie standard ANSI C fourni la fonction `int fgetc(FILE *)` pour lire un caractère d'un fichier. Expliquez la différence entre `fgetc()` et `read()`, et quel est l'intérêt d'utiliser l'une ou l'autre ?
(Solution à la page 56)

3.8.3 Existence d'un fichier

Écrivez le code pour verifier si le fichier `/home/lipari/mydata.txt` existe ou pas.
(Solution à la page 56)

3.8.4 Exercice : trouvez l'erreur

Considerez le programme suivant :

```
char * read_string(int fd)
{
    char buf[BUF_SIZE];
    int n = read(fd, buf, BUF_SIZE - 1);
    assert(n >= 0);
    buf[n] = 0;
    return buf;
}
```

Trouvez l'erreur et proposez un possible correction.
(Solution à la page 57)

3.8.5 Exercice : lecture du dernier octet

Écrire le code d'une fonction pour lire le dernier character d'un fichier :

```
char read_last(int fd) {
    // écrire la fonction
}
```

(Solution à la page 57)

Chapitre 4

Programmation Multithread

Un processus peut contenir un ou plusieurs *thread* d'exécution. Un thread est un flux d'exécution qui n'est pas lié à un espace mémoire privé (et qui diffère d'un processus). Bien sûr, un thread a besoin d'un espace mémoire pour fonctionner, mais cet espace n'est pas nécessairement privé. En ce sens, on peut dire que chaque processus contiendra un ou plusieurs threads : un processus contenant un seul thread sera un processus Unix classique, tandis qu'un processus composé de plusieurs threads (qui peuvent donc communiquer entre eux via la mémoire commune) sera un processus *multithreaded*.

Les threads créés par un processus tournent dans le même espace d'adressage (celui du processus qui les a créés) et peuvent ainsi bénéficier, sans aucune charge du noyau, d'une zone de mémoire partagée par laquelle l'échange de données peut avoir lieu. Bien entendu, la gestion de la concurrence entre threads reste à la charge de l'application.

4.1 Généralités

De manière informelle, un thread n'est rien d'autre qu'une fonction (en ce sens que ce terme a dans le langage C) qui est exécutée de manière concurrente à d'autres fonctions, dans le cadre d'un processus.

Tous les threads créés dans une tâche partagent leur espace d'adressage. En plus de cela, chaque thread *hérite* du processus qui le crée les données suivantes :

— Descripteurs de fichiers,
— Handler du Signal,
— Répertoire courant,
— ID d'utilisateur et de groupe.

Mais chaque thread possède son propre :

— ID du thread,
— Contexte du processeur (Stack Pointer, Log, Program Counter),
— Stack,
— variable *errno*,
— Masque des Signaux,
— Priorités.

Comme toujours dans la programmation, la prise en charge de certaines caractéristiques au sein d'un système d'exploitation se fait essentiellement par deux mécanismes : *types de données* et *fonctions du système d'exploitation*. Dans les paragraphes suivants, nous allons donc introduire graduellement

quelques-uns des nouveaux types de puisque certaines des nouvelles fonctions impliquées dans la gestion des threads.

4.2 Création des threads

Pour prendre en charge le mécanisme des threads, la norme POSIX définit d'abord un nouveau type de données qui est le `pthread_t`. Ce nouveau type de données sert à contenir l'identifiant unique attribué par Unix à chaque thread.

Chaque processus, avant de commencer la phase de création d'un ou plusieurs threads, peut communiquer au noyau le nombre de threads à créer appelé *concurrency level*. Ceci est fait en invoquant la routine :

```
#include <pthread.h>

int pthread_setconcurrency (int nThread);
```

— Paramètres :

nThread Nombre maximal de threads à créer.

— Valeur restitué :

0 En cas de succès.

$\neq$ **0** En cas d'échec.

Cette fonction n'est pas obligatoire et peut être ignorée dans la plupart des applications. Dans certains systèmes embarqués dont la mémoire est limitée, elle devient nécessaire pour optimiser les structures de données du système d'exploitation et pour anticiper la quantité de ressources nécessaires au processus.

Un thread est créé en invoquant la primitive suivante :

```
#include <pthread.h>

int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   (void *(*func)(void *), void *arg);
```

— Paramètres :

tid Lorsque vous revenez de l'appel, contient l'ID du fil

attr Valeurs passées de l'utilisateur pour les attributs du thread. Si vous voulez utiliser les valeurs par défaut, vous devez passer la valeur `NULL`.

func C'est la fonction qui contient le code du thread.

arg Paramètre facultatif à passer au thread. C'est un pointeur à la zone de mémoire qui contient les arguments à passer au thread nouvellement créé.

— Valeur restitué :

0 En cas de succès.

$\neq$ **0** En cas d'échec.

Une fois que vous avez créé un thread, il peut être nécessaire de se synchroniser avec son moment de fin, c'est-à-dire de s'arrêter jusqu'à ce que le thread soit terminé. Ceci est fait par la primitive :

```
#include <pthread.h>

int pthread_join (pthread_t tid, void **status);
```

— Paramètres :

tid Identifiant du thread sur lequel l'événement de fin doit être synchronisé.

status Pointez vers la zone de mémoire où les données renvoyées par le thread sont enregistrées.

— Valeur restitué :

0 En cas de succès.

$\neq$ **0** En cas d'échec.

Cette fonction ne retourne à l'appelant qu'une fois que le thread dont l'ID est passé dans le paramètre `tid` a terminé son exécution. Le noyau conserve les informations sur un thread qui a terminé son exécution jusqu'à ce qu'une `pthread_join` soit exécutée sur ce dernier. Cela permet, par exemple, d'empêcher un thread de se bloquer indéfiniment en attendant un autre thread qui a déjà terminé son exécution. Par contre, si nous ne sommes pas intéressés à exécuter le `pthread_join`, alors vous pourriez avoir le phénomène *thread zombie* (tout à fait analogue à ce qui se produit dans les processus). Pour éviter les zombies, vous pouvez invoquer la primitive suivante :

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

Habituellement, la fonction est appelée par le même thread que vous voulez détacher avec l'instruction suivante :

```
pthread_detach(pthread_self());
```

où la fonction `pthread_self()` renvoie le thread id du thread en cours d'exécution.

La fonction `pthread_kill()`, sert à envoyer un signal logiciel à un thread, et correspond à la `kill()` pour les processus.

```
#include <pthread.h>

int pthread_kill (pthread_t  tid, int signo);
```

— Paramètres :

tid Identifiant du thread auquel vous voulez envoyer le signal.

signo Identifiant du signal à envoyer.

— Valeur restituée_ :

0 En cas de succès.

$\neq$ **0** En cas d'échec.

4.2.1 Exemple de création de deux threads

Dans le programme suivant, le `main()` crée deux threads, dont le code est dans les fonctions `premier_thread` et `deuxieme_thread`, puis attend leur terminaison. Les deux threads s'exécutent de façon concurrente

sur la machine et les deux messages s'alternent sur la sortie standard. Il n'est pas possible de déterminer à priori l'exacte séquence des messages, ceci dépend de l'ordre d'exécution des deux threads qu'il n'est pas déterminé.

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

#define NITER 10
#define ATTENTE(y) for (int x=0; x<(y);x++)

void sys_err(const char *msg)
{
    printf("%s\n", msg);
    exit(-1);
}

/* code des deux threads */
void *premier_thread(void *arg)
{
    for (int i=0; i<NITER; i++) {
        ATTENTE(200000);
        printf("Premier Thread : iter = %d\n", i);
    }
    return 0;
}

void *deuxieme_thread(void *arg)
{
    for (int i=0; i<NITER; i++) {
        ATTENTE(150000);
        printf("Deuxième Thread : iter = %d\n", i);
    }
    return 0;
}

int main()
{
    pthread_t th1, th2;
    int r;

    r = pthread_create(&th1, NULL, premier_thread, NULL);
    if (r != 0)
        sys_err("Erreur dans la création du premier thread\n");

    r = pthread_create(&th2, NULL, deuxieme_thread, NULL);
    if (r != 0)
        sys_err("Erreur dans la creation du deuxième thread\n");

    pthread_join(th1, 0);
```

```
    pthread_join(th2, 0);
}
```

Listing 4.1 – Creation de deux threads.

Comme les deux threads ont presque le même code, on peut généraliser et créer une seule fonction pour les deux.

On utilise cette technique pour créer 10 threads dans le programme du listing 4.2.

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

#define NITER 10
#define ATTENTE(y) for (int x=0; x<(y);x++)

int nthreads = 10;

void sys_err(const char *msg)
{
    printf("%s\n", msg);
    exit(-1);
}

void *t_body(void *arg)
{
    int index = (int)arg;

    for (int i=0; i<NITER; i++) {
        ATTENTE(200000);
        printf("Thread %d/%d : iter = %d\n", index, nthreads, i);
    }
    return 0;
}

int main()
{
    pthread_t th[nthreads];
    int i, r;

    for (i=0; i<nthreads; i++) {
        r = pthread_create(&th[i], NULL, t_body, (void* )i);
        if (r != 0)
            sys_err("Erreur dans la création du thread\n");
    }
    for (i=0; i<nthreads; i++)
        pthread_join(th[i], 0);
}
```

Listing 4.2 – Creation de deux threads avec passage de paramètres

...	...
1	0x870478A0 (arg)
1	0x870478A4 (index)
	0x870478A8
...	...
	0x87058844
1	0x87058848 (i)
	0x8705884C
...	...

FIGURE 4.1 – Contenu de la mémoire pour la première tecnique ("casting") de passage des paramètres.

À la ligne 17 on retrouve la fonction `t_body` qui est le corps des 10 threads. La création est faite dans le `main()` à la ligne 35, la même fonction est utilisée dans la boucle pour créer les 10 threads.

Il faut souligner que chaque thread maintient sa propre copie des variables locales de la fonction. Dans ce cas, chaque thread a sa propre variable `index` privée. Par contre, toute variable globale est partagée : dans ce cas, la variable globale `nthreads` est partagée par les 10 threads et elle a la même valeur.

Notes

Rappel :
- les variables globales sont partagées par tous les threads ;
- les variables locales (déclarées sur la pile) sont privées à chaque thread.

4.3 Passage de paramètres

Pour passer des paramètres à un thread, on peut utiliser le pointer `arg` et le quatrième argument de la primitive `pthread_create()`. Il y a 3 techniques pour passer les paramètres.

4.3.1 Casting

La première technique est utilisée dans le listing 4.2. La variable `i`, de type entier, est transformé (casting) en pointer à `void` (ligne 35), et sa valeur est passée comme 4ème paramètre de la `pthread_create()`. Dans le thread, l'argument `arg`, de type pointer à `void`, est transformé en entier et affecté à la variable locale `index` de type entier. À partir de ce moment, la variable `index` peut être utilisée dans le code du thread.

La figure 4.1 montre ce qui se passe dans la mémoire lors du passage du paramètre pour le programme du listing 4.2. La valeur de la variable `i` est copié dans l'argument `arg`, (même si les types sont différents !), et après dans la variable `index`.

Cette technique est valide lorsque la taille du type de départ (ici un entier) est égale ou inférieure à la taille d'un pointer (ici `void *`). Dans le cas d'une architecture Intel x86, c'est normalement le cas, mais

...	...
0x87058848	0x870478A0 (arg)
1	0x870478A4 (index)
	0x870478A8
...	...
	0x87058844
1	0x87058848 (i)
	0x8705884C
...	...

FIGURE 4.2 – Contenu de la mémoire pour la deuxième technique (dite "pointer") de passage des paramètres.

ce n'est pas vrais pour toutes les architectures matérielles. En général, il faut vérifier que {sizeof(int) <= sizeof(void *)}.

4.3.2 Passage par pointer

La deuxième technique consiste à passer le paramètre par pointer. Un exemple est montré ci-dessous :

```
void *thread(void *arg)
{
    int index = *((int *) arg);
    printf("Thread n. %d\n", index);
    ...
}
// dans le main
int i = 1;
pthread_create(&tid, NULL, thread, (void *) &i);
```

Dans ce cas, l'adresse de la variable `i` (de type `int *`) est transformée en `void *` avant d'être recopié dans `arg`. On peut ensuite utiliser cette valeur pour affecter la valeur pointée à la variable `index`.

La figure 4.2 montre ce qui se passe dans la mémoire lors du passage du paramètre dans le cas du passage par pointer.

Attention, cette technique doit être utilisé avec caution ! Le problème peut arriver lorsqu'on crée plusieurs threads en utilisant une même variable `i`, comme dans l'exemple suivant :

```
pthread_t th[nthreads];
int i;

for (i=0; i<nthreads; i++)
    pthread_create(&th[i], NULL, t_body, (void* )&i);
```

Dans ce cas, il peut arriver que le thread qui vient d'être créé ne démarre pas immédiatement à cause du non-déterminisme de l'ordonnancement et donc la variable `i` change de valeur avant qu'elle soit recopié dans index. Par conséquent, on peut avoir plusieurs threads ayant la même valeur de `index`.

Pour être sûr que la variable `index` soit affectée la bonne valeur, il faut créer un tableau de paramètres comme dans le code suivant :

```
pthread_t th[nthreads];
int param[nthreads];
int i;

for (i=0; i<nthreads; i++) {
    param[i] = i;
    pthread_create(&th[i], NULL, t_body, (void* )&param[i]);
}
```

Dans ce cas, la valeur de `param[i]` ne change pas pendant l'exécution ; le thread est sûr d'y retrouver la bonne valeur même s'il commence à exécuter plus tard.

4.3.3 Généralisation

Pour passer plusieurs arguments à un thread, il est utile de le regrouper dans une structure, et passer l'adresse de la structure en paramètre à la fonction.

Supposons de vouloir lancer la fonction `int fun(int a, char *s, int n)` dans un thread. La fonction prends en argument trois parametres, et elle renvoie un entier. Nous allons donc préparer une structure de donnée comme dans le code ci-dessous :

```
struct f_params {
    int a;
    char *s;
    int n;
    int ret;
};
```

Les premières 3 champs de la structure representent la liste des arguments de la fonction `f()`, et le quatrième champ represent la valeur de retour.

Ensuite, on prépare un fonction pour *enrober* l'appel à la fonction `f()` :

```
void * f_wrapper(void *arg)
{
    struct f_param *p = (struct f_param *)arg;

    p->ret = f(p->a, p->f, p->n);

    return NULL;
}
```

Enfin, nous allons créer et lancer les threads :

```
int main()
{
    pthread_t tid[N];           /* N threads      */
    struct f_params params[N]; /* les parametres */

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

```
        /* initialise les parametres */
        params[i].a = ...;
        params[i].s = ...;
        params[i].n = ...;
        /* créer et lancer le i-eme thread */
        pthread_create(&tid[i], 0, f_wrapper, &params[i]);
    }

    /* ... */
    /* attendre leur terminaison */
    for (int i=0; i<N; i++)
        pthread_join(tid[i], 0);
    /* les résultats sont dans params[i].ret */

    /* ... */
}
```

Voir les exércices vu en TD et TP pour des exemples d'utilisation de cette techniques.

4.4 Synchronisation et exclusion mutuelle

4.4.1 Problèmes de synchronisation

Les threads exécutent en concurrence (en parallèle si plusieurs processeurs sont disponibles sur la plateforme matérielle). L'ordonnanceur du système d'exploitation décide quel thread s'exécute sur quel processeur. L'ordre et la durée d'exécution des threads ne sont pas faciles à contrôler. Pourtant, le programmer a besoin de gérer l'ordre d'exécution de manière précise pour garantir l'exécution correcte du programme.

Supposons qu'un calcul compliqué soit divisé en 2 étapes, représentée par les fonctions `f1()` et `f2()` : la fonction `f1()` fait une première partie du calcul et produit des résultats intermédiaires qui sont ensuite utilisé par la fonction `f2()` pour calculer le résultat final. Pour paralléliser le programme, le développeur décide d'appeler les deux fonctions dans deux threads différents. Cependant, il faut imposer que la fonction `f2()` soit toujours exécutée après la fonction `f1()`, car elle a besoin des résultats intermédiaires produits par `f1()`.

Example 4.4.1 Comme premier exemple, considérez un programme qui doit lire un ensemble de fichiers texte pour y chercher des mots. La fonction `f1()` lit un fichier et met son contenu dans un tampon ; la fonction `f2()` cherche le mot dans le tampon.

On peut utiliser une structure dite *pipeline* pour accélérer notre programme : pendant que le premier thread lit les fichiers en mémoire, le deuxième thread analyse les tampons déjà chargés. Il faut attendre que le tampon soit complétement chargé en mémoire avant d'analyser son contenu.

Malheureusement, il n'est pas facile de synchroniser les deux activités `f1()` et `f2()`. Jusqu'ici, la seule manière qu'on connaît pour synchroniser deux threads est la primitive `pthread_join()` pour attendre la terminaison d'un thread. On est donc obligé de créer des threads différents pour chaque fichier !

Par exemple, on pourrait :

— créer un premier thread qui lit le premier fichier en appelant la fonction `f1()` ;

— attendre sa terminaison avant de créer un deuxième thread qui va lancer la fonction `f2()` pour élaborer le premier tampon ;

— créer un troisième thread pour lire le deuxième fichier pendant l'exécution de `f2()` ;
— attendre sa terminaison avant de créer un quatrième thread qui va lancer la fonction `f2()` pour élaborer le deuxième tampon ;
— etc.

Même si cette stratégie est correcte, elle n'est guère efficace, parce qu'elle demande la création d'un nombre élevé des threads. La création d'un thread étant une opération coûteuse, le gagne en performance deviendra faible, voir négatif. ■

Un premier problème à resoudre est donc de synchroniser les différents étapes d'exécution des threads pour établir de *contraintes de précédence* : une activité doit s'exécuter après une autre activité pour garantir que le résultat soit correct.

4.4.2 Problèmes d'exclusion mutuelle

Un deuxième problème est l'accès aux ressources partagées.

Les threads ont accès à la même zone de mémoire du processus auxquels ils appartiennent. En particulier, un thread peut lire et écrire toutes les variables présentes dans la mémoire de son processus. Les opérations de lecture et écriture doivent être synchronisées et exécutés dans le bon ordre.

Example 4.4.2 Comme premier exemple, on considère une structure de donnée avec deux champs `a` et `b`. Sur cette structure, on défini deux opérations, la fonction `pair_add()` qui incrémente les deux champs et la fonction `pair_double()` qui double leurs valeurs.

```
struct pair {
    int a;
    int b;
};

void pair_init(struct pair *s)
{
    s->a = 0;
    s->b = 0;
}

void pair_add(struct pair *s)
{
    s->a++;
    s->b++;
}

void pair_double(struct pair *s)
{
    s->a = s->a * 2;
    s->b = s->b * 2;
}
```

Comme les deux champs `a` et `b` sont initialisé à `1`, si on utilise seulement les deux fonctions `pair_add()` et `pair_double()`, après chaque opération les deux champs doivent avoir la même valeur. Cette propriété est une *invariante* de notre programme.

Supposons qu'une instance de cette structure de données est accédé par deux threads différents. Voici le code :

```
void * t_a(void *arg)
{
    struct pair *s = (struct pair *)arg;

    pair_add(s);

    return 0;
}

void * t_b(void *arg)
{
    struct pair *s = (struct pair *)arg;

    pair_double(s);

    return 0;
}

int main()
{
    pthread_t tid_a, tid_b;
    struct pair p;
    pair_init(&p);

    pthread_create(&tid_a, 0, t_a, &p);
    pthread_create(&tid_b, 0, t_b, &p);

    pthread_join(tid_a, 0);
    pthread_join(tid_b, 0);
}
```

Supposons de lancer ce programme. Comme l'ordre d'exécution des instructions de deux threads n'est pas garanti, il y a plusieurs entrelacements possibles. Une possibilité est que les fonctions `pair_add` et `pair_double` soient appelée l'une après l'autre. Dans ce cas, la propriété d'invariance est respectée.

Thread `t_a`	Thread `t_b`	`p.a`	`p.b`
`pair_add(s) {`		1	1
`. s->a++;`		2	1
`. s->b++;`		2	2
`}`		2	2
	`pair_double(s) {`	2	2
	`. s->a*=2;`	4	2
	`. s->b*=2;`	4	4
	`}`	4	4

Une autre possibilité est que les instructions s'entremêlent :

Thread `t_a`	Thread `t_b`	`p.a`	`p.b`
`pair_add(s) {`		1	1
`. s->a++;`		2	1
	`pair_double(s) {`	2	1
	`. s->a*=2;`	4	1
	`. s->b*=2;`	4	2
	`}`	4	2
`. s->b++;`		4	3
`}`		**4**	**3**

Dans ce cas, la propriété d'invariance n'est pas respéctée.

Notez que le même programme produit un résultat différent selon l'entrelacement des instructions, ce qui n'est pas sous le contrôle du programmeur. Dans certains exécution, et de manière aléatoire, le programme pourra produire des résultats incorrects.

Notez aussi que c'est très difficile de découvrir ce genre d'erreurs par testing :

1. l'erreur n'est pas reproductible : deux exécutions avec le même donnée en entrée pourront donner des résultats différents ;
2. pour trouver l'erreur, il faudrait tester le programme plusieurs fois avec les mêmes entrées en espérant de reproduire la condition défavorable.

Ce type d'erreur est aussi appelé **course critique**. ■

Notes

Il faut noter que même des instructions apparemment simples, comme l'incrémentation d'une variable, sont en réalité composé de plusieurs étapes élémentaires. Par exemple, dans les architectures modernes l'instruction `x++` consiste concrètement de 3 étapes :

— le contenu de la variable `x` est transféré de la mémoire vers un des registres du processeur ;
— le registre est incrémenté ;
— la valeur dans le registre est transférée à nouveau vers l'adresse en mémoire de la variable `x`.

Par conséquent, si deux threads incrémentent la même variable `x` en mémoire, le résultat peut-être différent selon la manière dont les trois opérations s'entrelacent.

Une **opération atomique** est une opération qui ne peut pas être divisé en plusieurs étapes élémentaires. Par exemple, la lecture d'une variable est une opération atomique ; l'incrémentation d'une variable ne l'est pas.

C'est utile d'introduire quelques définitions.

1. Une **ressource partagée** est un élément logiciel utilisé par plusieurs threads. Un exemple de ressource partagée est une variable ou un ensemble de variables utilisées en lecture et en écriture par plusieurs threads.
2. Une **section critique de code** est une séquence d'instructions opérant sur une ressource partagée.
3. Deux sections critiques sur la même ressource doivent s'exécuter en **exclusion mutuelle**, c'est-à-dire les instructions de l'une ne doivent pas s'entrelacer avec les instructions de l'autre.
4. Si l'exclusion mutuelle n'est pas respectée, on parle de **course critique** (**race condition** en anglais) dans le code.

4.5 Les sémaphores

Pour resoudre les problèmes de synchronisation et exclusion mutuelle décrits dans la section précedente, le système d'exploitation met à disposition du développeur des mécanismes spécifiques et des appels système.

Un de ces mécanismes est le **sémaphore**. Un sémaphore est une structure de donnée qui se trouve dans le noyau et que le développeut peut utiliser en déclarant une variable de type `sem_t` :

```
sem_t s;
```

Les champs de cette structure de donnée ne sont pas directement accessibles par l'utilisateur ; il est cependant utile de la décrire ici pour bien comprendre le mécanisme. Le sémaphore est une structure qui consiste d'un compteur et d'une file d'attente de threads.

```
/* Pseudo-code de la structure sémaphore dans le noyau du
   système d'exploitation. Ce code n'est normalement pas
   visible aux utilisateurs. */
struct __semaphore {
    int counter;
    queue_t waiting_queue;
};
```

La primitive `sem_init()` sert à initialiser le sémaphore :

```
#include <semaphore.h>

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

sem est l'adresse du sémaphore qu'on veut initialiser

pshared si egale à 1, il indique que le sémaphore est partagé avec autre processus ; dans ce cours, ce paramètre sera toujours à 0 ;

value valeur initiale du compteur `counter`.

La fonction retourne `0` en cas de succès, `-1` en cas d'erreur.
Après avois initialisé la variable `sem_t`, on peut faire 2 opérations.

```
#include <semaphore.h>

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

Le pseudo-code de la primitive `sem_wait` est montré ci-dessous

```
/* Pseudo code de sem_wait */
int sem_wait(sem_t *sem)
{
    if (sem->counter <= 0) {
        /* - bloque le thread courant,                          */
        /* - et il le met dans la file d'attente sem->waiting_queue */
        /* - appelle l'ordonnanceur pour exécuter un autre thread   */
```

```
        }
        else sem->counter--;
    }
```

Le comportement de la primitive dépend de la valeur du compteur :

— s'il est négatif ou null, le thread qui vient d'appeler la `sem_wait()` est suspendu, il ne peut pas continuer son exécution. Il est enregistré dans la file d'attente du sémaphore et l'ordonnanceur du système d'exploitation est appelé pour mettre en exécution un autre thread.

— S'il est positif, le compteur est décrémenté puis le thread continue son exécution.

Le pseudo-code de la primitive `sem_post()` est montré ci-dessous :

```
    /* Pseudo code de sem_post */
    int sem_post(sem_t *sem)
    {
        if (/* s'il existe un thread dans sem->waiting_queue */) {
            /* - extrait le premier thread dans la file d'attente sem->
    waiting_queue */
            /* - et il le réveille (il est à disposition de l'ordonnanceur) ;
         */
            /* - appelle l'ordonnanceur pour décider si exécuter (ou pas) le
         */
            /*   thread réveillé
         */
        }
        else sem->counter++;
    }
```

La `sem_post()` est la fonction *inverse* de la `sem_wait()` : si nécessaire, il reveille un thread bloqué precedemment par une `sem_wait()`, et si non il incrémente le compteur du sémaphore.

Nous allons maintenant voir des exemple d'utilisation des sémaphores.

4.5.1 Sémaphore pour synchronisation

Pour synchroniser deux threads, on utilise un sémaphore `s` dont le compteur est initialisé à `0`. S'il y a un contraint de précedente entre un activité `A` et une activité B, on ajoute un appel à `sem_post(&s)` après `A` et un appel à `sem_wait(&s)` avant `B` (voir le code ci-dessous).

```
    void *t_a(void *arg)
    {
        / * ... */
        A();
        sem_post(&s);
        /* ... */
    }

    void *t_b(void *arg)
    {
        /* ... */
        sem_wait(&s);
        B();
```

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

Nous allos regarder maintenant deux situations possible. Dans le premier cas, le thread `t_a` demarre en premier et arrive au point de synchronisation avant `t_b` :

Thread `t_a`	Thread `t_b`	s.counter	s.waiting_queue
`A();`	`/* ... */`	0	empty
`sem_post(&s);`	`/* ... */`	1	empty
`/* ... */`	`/* ... */`	1	empty
	`sem_wait(&s);`	0	empty
	`B();`	0	empty

Nous considerons maintenant le cas ou le thread `t_b` demarre en premier et arrive au point de synchronisation avant `t_a` :

Thread `t_a`	Thread `t_b`	s.counter	s.waiting_queue
`/* ... */`	`/* ... */`	0	empty
`/* ... */`	`sem_wait(&s);`	0	`[ t_b ]`
`/* ... */`	<blocked>	0	`[ t_b ]`
A();		0	`[ t_b ]`
`sem_post(&s);`	<unblocked>	0	empty
`/* ... */`	`B();`	0	empty
`/* ... */`	`/* ... */`	0	empty

Quand le thread `t_b` appelle la `sem_wait()` (deuxième ligne du tableau), la valeur du compteur est à zéro, par conséquent le thread `t_b` est bloqué, et l'ordonnanceur va exécuter un autre thread à sa place. Quand plus tard le thread `t_a` appelle la `sem_post()`, cette dernière débloques les thread `t_b` qui peut reprendre son exécution et continuer avec l'appel à `B()`.

Dans le deux cas, la fonction `B()` s'est exécuté après la fonction `A()`, comme demandé : dans le première cas de manière naturel, dans le deuxième cas en bloquant le thread `t_b` en attente que la fonction `A()` soit terminée.

4.5.2 Sémaphore pour exclusion mutuelle

Pour garantir l'exclusion mutuelle sur deux séctions critiques sur la même ressource, on utilise un sémaphore d'exclusion mutuelle `m` initialisé à `1`. Avant chaque section critique on appelle la `sem_wait(&m)`, et après chaque section critique on appelle la `sem_post(&m)`.

Example 4.5.1 Considerons à nouveau l'exemple de la section précedente. Pour garantir l'exclusion mutuelle, on ajoute un sémaphore `m` à la structure et on l'initialise à 1. Voici le code modifié.

```
struct pair {
    int a;
    int b;
    sem_t m;
};

void pair_init(struct pair *s)
```

```
{
    s->a = 0;
    s->b = 0;
    sem_init(&m, 0, 1);
}

void pair_add(struct pair *s)
{
    sem_wait(&s->m);
    s->a++;
    s->b++;
    sem_signal(&s->m);
}

void pair_double(struct pair *s)
{
    sem_wait(&s->m);
    s->a = s->a * 2;
    s->b = s->b * 2;
    sem_signal(&s->m);
}
```

Maintenant on simule son exécution, et on essaie de répliquer le deuxième cas ou les instructions des deux threads sont entrelacées.

Thread `t_a`	Thread `t_b`	`p.a`	`p.b`	`p.m.counter`	`p.m.waiting_queue`
`pair_add(s) {`		1	1	1	empty
`. sem_wait(&s->m);`		1	1	0	empty
`. s->a++;`		2	1	0	empty
	`pair_double(s) {`	2	1	0	empty
	`. sem_wait(&s->m);`	2	1	0	`[t_b]`
`. s->b++;`	<blocked>	2	2	0	`[t_b]`
`. sem_post(&s->m);`	<unblocked>	2	2	0	empty
	`. s->a*=2;`	4	2	0	empty
	`. s->b*=2;`	4	4	0	empty
	`. sem_post(&s->m);`	4	4	1	empty
	`}`	4	4	1	empty
`}`		4	4	1	empty

D'abord, on observe que quand le thread `t_a` appelle la primitive `sem_wait(&s->m)` (ligne 16 du code, ligne 2 du tableau d'exécution ci-dessus) le compteur descend à `0` pour signaler que la ressource est occupé. Si un autre thread essaie de faire un `sem_wait()` sur le même sémaphore, il sera bloqué.

En effet, quand le thread `t_b` essaie de faire la `sem_wait(&s->m)` (ligne 24 du code, ligne 5 dans le tableau d'exécution), il est bloqué : il ne peut pas continuer à exécuter les instructions de la section critique, il devra attendre que la ressource soit libérée par le thread `t_a`.

Plus tard, après avoir exécuté le code critique, le thread `t_a` libère la ressource en appelant la primitive `sem_post(&s->m)` (ligne 19 du code, ligne 7 dans le tableau), ce qui réveille le thread `t_b` qui pourra maintenant exécuter sa section critique.

Il est facile de voir qu'une situation similaire va se reproduire lorsque le thread `t_b` démarre en premier.
■

4.6 Interblocage et famine

L'utilisation des sémaphores peut conduire à des erreurs de programmations tel que l'**interblocage** (ou *deadlock* en anglais) et la famine (ou **livelock** en anglais).

4.6.1 Interblocage

Dans l'interblocage, un ensemble de threads est bloqué pour toujours, chaque thread attends d'être reveille par un des autres threads qui est déjà bloqué. Normalement, ce type de problème se manifeste lors d'une condition d'*attente circulaire*.

Analysons le code suivant :

```
sem_t m1; // initialisé à 1
sem_t m2; // initialisé à 1

void *t_a(void *arg)
{
    sem_wait(&m1);
    sem_wait(&m2);
    // code
    sem_post(&m2);
    sem_post(&m1);
}

void *t_b(void *arg)
{
    sem_wait(&m2);
    sem_wait(&m1);
    // code
    sem_post(&m1);
    sem_post(&m2);
}
```

Supposons maintenant la suite d'exécution décrite dans le tableau suivant :

Thread `t_a`	Thread `t_b`	m1.counter	m1.waiting_queue	m2.counter	m2.waiting_queue
		1	[]	1	[]
`sem_wait(&m1);`		0	[]	1	[]
	`sem_wait(&m2);`	0	[]	0	[]
`sem_wait(&m2);`		0	[]	0	[t_a]
<blocked>	`sem_wait(&m1);`	0	[t_b]	0	[t_a]
<blocked>	<blocked>	0	[t_b]	0	[t_a]

Dans ce cas, les deux threads restent bloqués, l'un attends d'être réveillé par l'autre.

L'interblocage est une condition d'erreur qui doit être évité. Dans le code de l'exemple, une solution simple est d'inverser l'ordre d'exécution des `sem_wait()` dans le thread `t_b`.

Dans de situations plus compliquées, il n'est pas toujours évident de repérer le problème et de le résoudre.

4.6.2 Famine

La famine (**livelock**) est une condition ou un thread n'arrive pas à accéder à une ressource, même s'ils ne restent jamais bloqués.

Pour expliquer la famine, on prend l'exemple de la primitive `sem_trywait()`.

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
```

La primitive fonctionne de la manière suivante :

— si le compteur est inférieur ou égale à 0, elle retourne -1 et `errno` est égale à `EAGAIN`.
— si le compteur est supérieur à zéro, il est décrémenté.

La `sem_trywait()` se comporte comme la `sem_wait()`, mais il ne bloque jamais le thread.

Analysons maintenant le code suivant :

```
sem_t m; // initialisé à 1

void *thread_body(void *arg)
{
    while (1) {
        /* ... */
        while (sem_trywait(&m) < 0);
        /* section critique */
        sem_post(&m);
    }
}
```

Supposons qu'il y a 3 threads qui exécutent le même code `thread_body`, les threads `tid_1`, `tid_2` et `tid_3`. Un seul thread à la fois parmi le trois peux exécuter le code marqué comme *section critique* ; les autres 2 vont attendre dans une boucle `while`. Selon l'ordre d'exécution des 3 threads, il peut arriver que seulement 2 de ces threads accèdent à la ressource, le troisième reste toujours en attente parce qu'il n'a jamais la chance d'être en exécution quand le compteur du sémaphore est à 1. Notez qu'aucun thread n'est jamais bloqué, mais il existe une possibilité (même si rare) qu'un thread n'arrive jamais à avancer.

Un autre problème très connu de famine est le problème des lecteurs/écrivains que nous verrons dans la suite de ce cours (PDS+). Voir aussi le problème du diner des philosophes.

Ce type de problème est moins embêtant que l'interblocage, et il peut normalement être résolu en utilisant de files d'attentes triées par date d'arrivée (*First In First Out*, premier arrivé, premier servi).

4.7 Questions et exercices

4.7.1 Question : Passage de paramètres

Décrivez la méthode dite "Casting" pour passer un paramètre à un thread, en donnant un exemple. Quel sont les limitation de cette méthode ?

4.7.2 Question : Valeur de retour

Décriver au moins une méthode pour obtenir la valeur de retour d'un thread, en donnant un exemple.

4.7.3 Question : Synchronization

Supposons de vouloir synchroniser un thread TA avec un thread TB :

```
void *threadA(void *arg)
{
    FA(1);
    // synchronisation
    FA(2);
}

void *threadB(void *arg)
{
    FB(1);
    // synchronisation
    FB(2);
}
```

Il faut que `FB(2)` s'execute après `FA(1)`. Déclarez les sémaphores nécessaires et écrivez le code de synchronization pour imposer la condition ci-dessous.

(Solution à la page 57)

4.7.4 Exercice : graphe de synchronization

Considerez les threads suivants :

```
void *ta(void *arg)
{
    // TODO : synchronization
    printf("TA\n");
    // TODO : syncronization
}
void *tb(void *arg)
{
    // TODO : synchronization
    printf("TB\n");
    // TODO : syncronization
}
void *tc(void *arg)
```

```
{
    // TODO : synchronization
    printf("TB\n");
    // TODO : syncronization
}
void *td(void *arg)
{
    // TODO : synchronization
    printf("TB\n");
    // TODO : syncronization
}
```

On veut imposer les constraintes de synchronization suivantes :
— `TB` doit s'afficher après `TC` ;
— `TC` doit s'afficher après `TD` ;
— `TA` doit s'afficher après `TD`.

Déclarer les semaphores et leur valeur initiale, et écriver le code de synchronization (en replaçant les commentaires `TODO`) dans le code, pour imposer les contraintes de précedence.

(Solution à la page 58)

4.7.5 Exercice : dessiner le graphe

Dessiner un graph de synchronisation [1] pour le thread suivants :

```
sem_t sa; // initialisé à 0
sem_t sd; // initialisé à 0

void *ta(void *arg)
{
    printf("TA\n");
    sem_post(&sa);
    sem_post(&sa);
}
void *tb(void *arg)
{
    sem_wait(&sa);
    printf("TB\n");
    sem_post(&sd);
}
void *tc(void *arg)
{
    sem_wait(&sa);
    printf("TC\n");
    sem_post(&sd);
}
void *td(void *arg)
{
    sem_wait(&sd);
```

1. Un graphe de synchronisation est un graphe ou les noeuds sont des threads, et ou les arêtes indique une synchronisation entre le deux thread reliés : le thread "source" de l'arête doit s'exécuter avant le thread "destination" de l'arête.

```
    sem_wait(&sd);
    printf("TD\n");
}
```

(Solution 59)

4.7.6 Exercice : ressources partagées

Plusieurs threads appellent les fonctions `get_ressource()` et `release_ressource()` décrites ci-dessous :

```
int ressource[N]; // ressource[i] = 0 : libre
                  // ressource[i] = 1 : occupé

int get_ressource()
{
    for (int i=0; i<N; i++)
        if (ressource[i] == 0) {
            ressource[i] = 1;
            return i;
        }
    return -1; // aucune ressource libre trouvé !
}

int release_ressource(int i)
{
    ressource[i] = 0;
}
```

La fonction `get_ressource()` occupe la première ressource libre, et retourne son index. La fonction `release_ressource()` libère la ressource avec index `i`.

La même ressource ne doit pas être occupé par deux threads différents.

Est-ce qu'il faut proteger le code avec des semaphores d'exclusion mutuelle ? Si oui, modifiez le code. Sinon, expliquez pourquoi.

(Solution à la page 59).

Chapitre 5

Solutions aux questions et exercices

5.1 Questions sur le processus

5.1.1 Variable d'environnement

C'est une variable qu'on peut definir dans une session de l'interpret de commande (*shell*), et on peut lui affecter une chaîne de characters. Par exemple, chaque instance de shell declare toujours la variable `HOME` qui correspond au repértoire de base de l'utilisateur courant ; la variable `USER` qui correspond au login name de l'utilisateur ; la variable `PATH` qui contient la liste de répertoire ou il faut chercher les fichiers executables ; et beaucoup d'autres.

5.1.2 Vérifier l'existence d'un fichier

La fonction `access` accepte deux paramètres : le premier est le chemin vers le fichier dont on veut obtenir des informations ; le deuxième est la modalité d'accès qu'on souhaite tester. Dans notre cas, le deuxième paramètre vaut `R_OK` parce qu'on cherche à vérifier l'accès en lecture (Read).

Le premier paramètre sera l'enchainement de la valeur de la variable d'environnement `HOME` et de la chaîne de caractères `.bashrc`. Voici la solution finale :

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

int main()
{
    char *home = getenv("HOME");
    char filename[PATH_MAX];

    strcpy(filename, home);
    strcat(filename, "/.bashrc");
    printf("Checks that %s is accessible in read mode ...\n", filename);
    int result = access(filename, R_OK);
```

```
    if (result == 0) {
        printf("YES\n");
        return EXIT_SUCCESS;
    }
    else {
        printf("No\n");
        return EXIT_FAILURE;
    }

}
```

5.2 Questions sur le systèmes de fichiers

5.2.1 Calcul de la taille d'un fichier

On peut utiliser la syscall `lstat()`

```
    struct stat s;
    assert(stat("myfile", &s) == 0);
    printf("Taille : %d bytes", s.st_size);
```

Une deuxième manière consiste à utiliser la `lseek()`

```
    int fd=open("myfile", O_RDONLY);
    int s = lseek(fd, 0, SEEK_END);
    printf("Taille : %d bytes", s);
    close(fd);
```

5.2.2 fgetc

La fonction `read()` est une *syscall* ; la fonction `fgetc()` est une fonction de la librarie standard du C. La fonction `fgetc()` utilise un tampon pour éviter des appels trop fréquents et inutiles à la fonction `read()` ; le première appel de `fget()` corresponds d'un appel à la `read()` pour lire un bloc d'octets dans le tampon, et ensuite retourner un seul caractère du tampon ; les appels suivants à la `fgetc()` retournent les caractères suivants du tampon, sans appeler la `read()`.

Comme les appels systèmes sont notamment plus coûteux que les appels de librairie, `fgetc()` permet de reduire le nombre d'appels à la `read()`, et donc reduire le côut moyenne pour lire un caractère.

5.2.3 Existence d'un fichier

```
   if (access("/home/lipari/mydata.txt", F_OK) == 0)
       printf("Le fichier existe\n");
   else printf("Le fichier n'existe pas\n");
```

5.2.4 Trouvez l'erreur

Il ne faut jamais retourner un pointer à une variable automatique. Il faut utiliser `malloc()` ou une variable globale pour la variable `buf`. Par exemple :

```
static char buf[BUF_SIZE];
char * read_string(int fd)
{
    int n = read(fd, buf, BUF_SIZE - 1);
    assert(n >= 0);
    buf[n] = 0;
    return buf;
}
```

5.2.5 Exercice : lecture du dernier octet

```
char read_last(int fd) {
    char c;
    if (lseek(fd, -1, SEEK_END) < 0) {
        printf("lseek failed\n");
        exit(-1);
    }
    int n = read(fd, &c, 1);
    if (n < 1) {
        printf("read failed\n");
        exit(-1);
    }
    return c;
}
```

5.3 Questions sur la programmation multi-thread

5.3.1 Synchronization

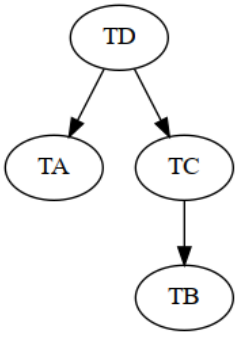
Il suffit de déclarer un seul semaphore `s`.

```
sem_t s; // initialisé à 0

void *threadA(void *arg)
{
    FA(1);
    sem_post(&s);
    FA(2);
}

void *threadB(void *arg)
{
    FB(1);
```

```
    sem_wait(&s);
    FB(2);
}
```

5.3.2 Graphe de synchronization

On peut visualiser les contraintes comme un graphe :

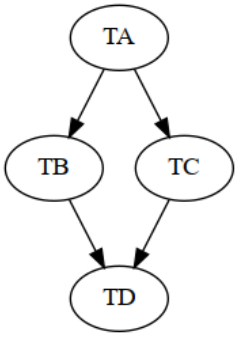


Par conséquent, il suffit de definire 2 semaphores, un pour le thread `TD` et un pour le thread `TB`.

```
sem_t sd; // initialisé à 0
sem_t sc; // initialisé à 0

void *ta(void *arg)
{
    sem_wait(&sd);
    printf("TA\n");
}
void *tb(void *arg)
{
    sem_wait(&sc);
    printf("TB\n");
}
void *tc(void *arg)
{
    sem_wait(&sd);
    printf("TB\n");
    sem_post(&sc);
}
void *td(void *arg)
{
    printf("TB\n");
    sem_post(&sd);
    sem_post(&sd);
}
```

5.3.3 Dessiner le graph

Voici le graphe :



5.3.4 Ressources partagées

Oui, il faut proteger le code, sinon on risque que 2 threads s'empare de la même ressource. Ce qu'il peut se passer :

- le première thread demarre et trouve la `ressource[0]` égale à 0 ; mais avant qu'il puisse mettre la ressource à 1 (pour signaler qu'elle est occupée) ...
- ...un deuxième thread exécute la même fonction, trouve la même `ressource[0]` à zéro, la met à 1, et retourne l'index 0 ;
- quand le premier thread reprends son exécution, il retourne le même index 0.

C'est une erreur !

Voici la correction :

```
int ressource[N]; // ressource[i] = 0 : libre
                  // ressource[i] = 1 : occupé
sem_t m; // initialisé à 1

int get_ressource()
{
    int i;
    sem_wait(&m);
    for (i=0; i<N; i++)
        if (ressource[i] == 0) {
            ressource[i] = 1;
            break;
        }
    if (i == N) i = -1; // aucune ressource libre trouvé !
    sem_post(&m);
    return i;
}

int release_ressource(int i)
{
    sem_wait(&m);
```

```
    ressource[i] = 0;
    sem_post(&m);
}
```

Notez qu'il faut aussi mettre le code de `release_ressource()` dans une section critique.