ENSICAEN
ÉCOLE NATIONALE SUPÉRIEURE D'INGÉNIEURS DE CAEN
& CENTRE DE RECHERCHE

Spécialité Informatique
2ème année
Année 2013–2014

Systèmes d'exploitation

Cahier de Travaux Pratiques

Gilles Lebrun (gilles.lebrun@ensicaen.fr)
Document original de : Sébastien Fourey (sebastien.fourey@ensicaen.fr)

⋱

Table des matières

Présentation

Ce cahier de TP comporte des explications sur des notions que ne seront pas détaillées en cours ainsi que des exercices mettant en pratique ces notions. Les exercices marqués du symbole

??/20

sont à rendre à votre enseignant de TP.

Attention : Il est fortement recommandé de répondre à *tous* les exercices qui sont proposés et pas seulement à ceux qui sont à rendre. En effet, ils correspondent à un niveau de progression et de difficulté croissants, ce qui permet d'aborder les exercices à rendre, qui sont plus « conséquents », de manière progressive. Une grande partie de ce que vous aurez écrit pour répondre aux exercices intermédiaires vous sera utile pour réaliser ceux que vous devrez rendre.

Comment être efficace en TP ?

- Apprendre à utiliser le manuel en ligne d'Unix (commande `man`). Celui-ci permet en effet un accès rapide à la documentation des commandes Unix, des appels système, des fonctions de la bibliothèque standard du C, etc.
  Rappelons que le manuel `man` est divisé en sections :
  - Section 1 : Commande de base shell et Unix ;
  - Section 2 : Appels système ;
  - Section 3 : Bibliothèques de programmation ;
  - etc.

  Par exemple, sous Linux, la commande "`man printf`" affiche la documentation de la commande `printf` du shell alors que "`man 3 printf`" affiche celle de la fonction `printf()` du langage C (Section 3). Quand on fait référence à une page de manuel, on juxtapose le nom de la page et la section pour lever toute ambiguïté : `printf(3)`.

Chapitre 1

Processus

1.1 Notion de processus

Ici, on appelle *programme* une suite d'instructions exécutables par l'ordinateur et qui se présente sous la forme d'un fichier. Ce peut être du code objet ($\simeq$ code machine) mais on désignera aussi par ce terme un fichier compréhensible par un interpréteur (p. ex. un shell Unix comme `sh` ou `bash`, un interpréteur `perl`, une machine virtuelle Java, etc.).

Définition 1 (Processus) *Un* processus *est un programme informatique en cours d'exécution sur une machine donnée. C'est une entité dynamique qui évolue dans le temps, par opposition à un programme qui est une entité statique.*

L'espace occupé en mémoire par un processus est divisé en 3 segments :
- Un segment de *code* contenant la partie TEXT du fichier exécutable ;
- Un segment de *données* dans lequel sont placées les variables globales ou statiques (partie DATA) ainsi que les zones allouées dynamiquement pendant l'exécution ;
- Un segment de *pile* (STACK) contenant les données liées aux appels de fonctions (adresses de retour, résultats, paramètres et variable locales).

Quand la pile grossit, l'adresse de son sommet diminue (cf. figure 1.1).

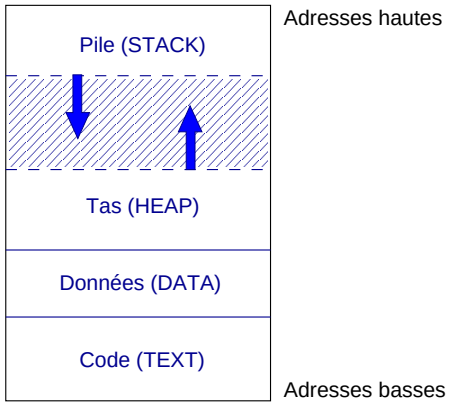


FIGURE 1.1 – Segments de mémoire d'un processus.

1.2 Multitâches et ordonnancement de processus

Le terme *multitâches* s'applique à un système sur lequel peuvent s'exécuter *simultanément* plusieurs processus. Sur un système équipé d'une seule unité de traitement, c.-?-d. un seul processeur simple-cœur, cette simultanéité n'est qu'apparente. En réalité, à un instant donné seulement un processus s'exécute réellement et tous les autres sont en attente.

Une partie du système d'exploitation, l'*ordonnanceur*, se charge d'attribuer le processeur [1] à un processus choisi parmi N, et ce pour une temps donné. Cette durée maximum pendant laquelle le processus *élu* pourra s'exécuter est appelé *quantum* (généralement de l'ordre de quelques millièmes ou centièmes de seconde).

En première approximation, on peut dire qu'un processus peut être dans l'un des trois états suivants (voir aussi la figure 1.2) :

Élu Le processus est en cours d'exécution, le processeur lui est attribué.

Prêt Le processus est prêt à être exécuté.

En attente Le processus est en attente d'une opération bloquante. (Il serait donc inutile de l'élire.)

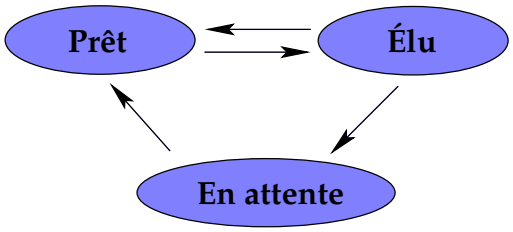


FIGURE 1.2 – Trois états d'un processus.

Un processus peut ne pas utiliser la totalité du quantum qui lui est attribué. Par exemple, s'il déclenche une demande de lecture dans un fichier par un appel système et que l'opération sera longue, alors il sera interrompu et mis en attente.

1.3 Création de processus

Il n'existe pas de génération spontanée chez les processus. Un processus est toujours créé à la demande d'un autre processus. Celui qui est à l'initiative de la création est alors appelé processus *père*, celui qui est créé est le *fils*. On peut donc parler de hiérarchie de processus (Figure 1.3).

Sous Unix, la création d'un nouveau processus n'est possible que par clonage à l'aide de l'appel système `fork()`. Sous Windows, elle se fait grâce à la fonction `CreateProcess()` qui provoque le chargement d'un programme exécutable.

1.3.1 Création de processus sous Unix

Sous Unix, un processus est crée grâce à l'appel système `fork()`.

1. Ou l'un des processeurs s'il s'agit d'un système multi-processeurs. Dans la suite, on supposera que le système est mono-processeur.

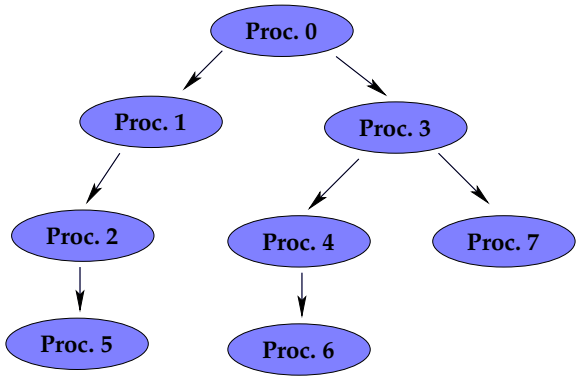


FIGURE 1.3 – Arbre de parenté entre processus.

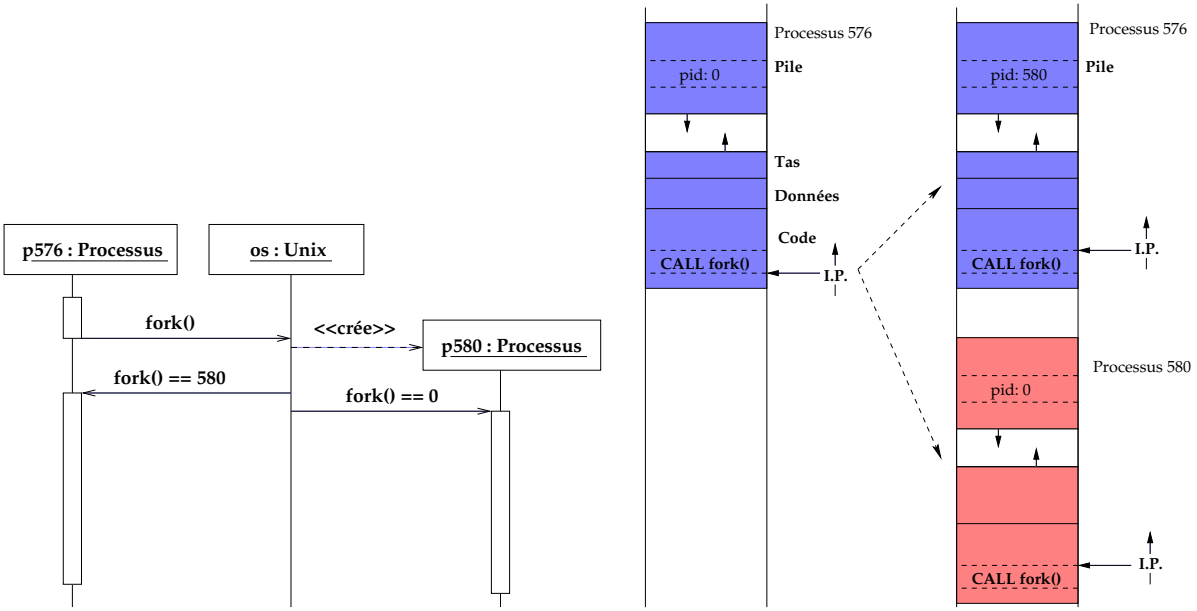
```
pid_t fork(void);    // pid_t est un type entier
```

Appel système `fork()`

L'appel de la fonction `fork()` provoque la création d'un nouveau processus qui est la copie conforme du processus qui a réalisé l'appel. Les contenus des segments de code, de donnée et de pile sont recopiés à un détail près : la valeur de retour de la fonction `fork()` qui se trouve dans la pile (cf. figure 1.4(b)).

Si la création échoue, la valeur renvoyée par `fork()` est -1. Si elle a réussi, l'exécution de chacun des deux processus se poursuit, précisément au retour du `fork()`. Seule la valeur de retour de la fonction diffère. Dans le père, la valeur renvoyée est le numéro du processus fils nouvellement créé : son PID (*Process Identification*) qui est un entier strictement positif. Dans le fils, la valeur renvoyée est simplement 0. (Voir figure 1.4.)

Pour le reste, le processus fils hérite notamment de tous les descripteurs de fichiers qui étaient ouverts dans le père.



(a) Séquence (b) Clonage

FIGURE 1.4 – Appel système `fork()`.

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

int main()
{
  pid_t pid=0;
  pid = fork();
  if ( pid == -1 ) {
    perror("fork()");
    exit(0);
  }
  if ( pid > 0 ) {
    printf("J'ai un fils de PID : %d\n", pid );
  } else {                                    // pid == 0
    printf("Je suis un nouveau n");
  }
  return 0;
}
```

Listing 1.1 – Appel système `fork()`

Un exemple classique d'utilisation de `fork()` est présenté dans le listing 1.1.

Mise en garde : Ne jamais utiliser `fork()` à l'intérieur d'une boucle avant d'y avoir réfléchi à deux fois. [2]

1.3.2 Exécution d'un programme sous Unix

Il était question dans la section précédente de créer un nouveau processus, mais le processus créé exécute exactement le même programme que son père puisqu'il en est une copie conforme. Comment alors lance-t-on l'exécution d'un programme quelconque pour en faire un processus sachant que le seul moyen de créer un nouveau processus est d'en cloner un via l'appel `fork()` ?

Unix prévoit pour cela un appel système qui demande le chargement d'un programme qui viendra remplacer (recouvrir) en mémoire le processus qui en fera la demande. C'est en fait une famille d'appels systèmes, de type `exec()`.

Il s'agit encore d'un appel système particulier car à la différence de `fork()` qui avait normalement *deux* retours, `exec()`, lui, ne retourne *jamais* si tout se passe bien ! En effet, si l'appel réussit c'est le programme chargé qui commence son exécution et le code du processus appelant a disparu de la mémoire ; il a été en quelque sorte écrasé.

Finalement, le seule moyen pour lancer l'exécution d'un programme sous Unix passe par l'utilisation du couple `fork/exec`. Le principe d'un tel lancement est résumé par la figure 1.5. Un exemple de code C réalisant le lancement d'un programme est donné dans le listing 1.2.

Toute les fonctions `exec*()` permettent d'exécuter le programme donné par leur premier ar-

2. Disons plutôt à $2^{n+1} - 1$ fois, où n est le nombre d'itérations de la boucle.

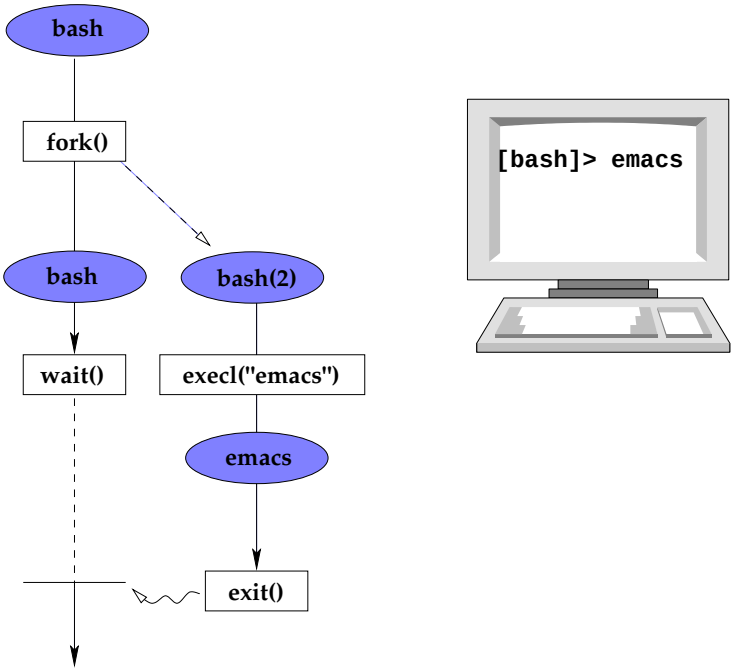


FIGURE 1.5 – Éxécution d'un programme sous Unix : Exemple du lancement d'emacs par un shell.

```c
#include <unistd.h>

int main()
{
  pid_t pid;
  pid = fork();
  if ( pid == -1 ) {
    perror("fork()");
  }
  if ( pid == 0 ) {
    printf("Prt  disparaitre\n");
    execl("/bin/ls","ls","-lF",NULL);
    perror("exec()");                    /* Pas de "if" ? */
  } else {
    printf("Je suis le pre\n");
  }
}
```

Listing 1.2 – Appel système `execl()`

gument. Celui-ci doit être le nom d'un fichier objet exécutable ou bien un script commençant par la ligne

```
#!/chemin/vers/interprete [args]
```

où `interprète` est un programme (et non un script) qui sera effectivement lancé avec le nom du script ajouté comme dernier argument.

Les différents prototypes des fonctions `exec()` sont donnés dans le listing ci-dessous et leurs différences sont précisées plus bas.

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ..., (char*)NULL);

int execv(const char *path, char *const argv[]);

int execle(const char *path, const char *arg0, ..., (char*)NULL,
           char *const envp[]);

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

int execlp(const char *file, const char *arg0, ..., (char*)NULL);

int execvp(const char *file, char *const argv[]);
```

Listing 1.3 – Les différentes fonctions de la famille `exec()`

Les fonctions `execl()` et `execv()` permettent d'exécuter un programme en précisant les arguments de l'appel sous la forme d'un liste de paramètres de la fonction (`execl()`) ou bien sous la forme d'un tableau de chaînes (`execv()`).

Les fonctions `execle()` et `execve()` permettent en plus de spécifier les variables du nouvel environnement souhaité sous la forme d'un tableau de chaînes (le dernier argument « envp »).

Les fonctions `execlp()` et `execvp()` se comportent comme `execl()` et `execv()` à ceci près que si le nom du programme donné par le premier argument ne commence pas par '/', alors le nom du programme est recherché dans tous les chemins de la variable d'environnement PATH.

1.3.3 Fin d'un processus sous Unix

Un processus se termine en faisant appel à la fonction `exit(3)`.

```
void exit(int status);
```

La valeur de *status* est renvoyée au processus père qui a créé ce processus. Le père peut récupérer cette valeur grâce à l'appel `wait(2)` qui le place en attente de la mort d'un de ses fils. Par défaut, la valeur de retour d'un processus *doit* être récupérée. Plusieurs situations sont possibles lors de l'appel `exit()` :

– Le processus père a précisé qu'il n'était pas intéressé par cette valeur de retour en s'immunisant contre le signal SIGCHLD par l'appel `signal(SIGCHLD,SIG_IGN)`. Dans ce cas, le processus fils disparaît et toutes ses ressources sont libérées.
– Le processus père est bloqué sur l'appel système `wait()` en attente de la mort d'un fils.

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Dans ce cas, le père reçoit le pid de son fils comme retour de la fonction `wait()` et la variable `status` est modifiée. Le processus fils se termine et ses ressources sont libérées.
– Si aucun des points précédents ne s'applique, alors les ressources sont libérées mais le processus ne disparaît pas complètement. Il passe dans l'état de « zombie » et occupe toujours une entrée dans la liste de processus du système. Cette vie de zombie ne dure heureusement pas éternellement car lorsque le processus père meurt à son tour, le zombie est adopté par l'ancêtre de tous les processus : `init`. Ce dernier, après avoir fait sont travail lors du lancement du système, passe l'essentiel de son temps à attendre la mort de sa descendance.

Exercice 1.1

Écrivez un programme « thriller » qui crée un zombie. Lancez ce programme qui devra s'exécuter jusqu'à être interrompu par un `Ctrl-C` mais sans consommer de ressource CPU (voir la fonction `pause(2)`). Depuis un autre terminal, vérifiez l'existence du zombie à l'aide de la commande `ps(1)`. Finalement, interrompez le programme et vérifiez la disparition du zombie.

1.4 Les entrées/sorties

1.4.1 Sous Unix

Notion de descripteurs de fichier

Les fonctions d'entrée/sortie de la bibliothèque standard du C utilisent une structure de type FILE manipulée par adresse pour désigner un fichier ouvert. Sous Unix, les fonctions comme `fprintf`, `fread`, `fwrite` et autres sont traduites en appels systèmes de manipulation de fichiers. Ces appels systèmes utilisent en fait comme descripteurs de fichiers de simples entiers.

Les cinq appels de base sont :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);

#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

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

Listing 1.4 – Appels systèmes de manipulation de fichiers.

Les entiers renvoyés par `open(2)` et donnés en arguments à `read(2)`, `write(2)` et `close(2)` sont donc appelés simplement « descripteurs de fichiers » (*file descriptors*, fd). Les valeurs possibles pour les drapeaux d'ouverture sont données dans la page de manuel `open(2)`.

Entrées et sorties standard

Le descripteur 0 est un fichier ouvert en lecture seule qui correspond à l'entrée standard du processus, les fichiers 1 et 2 (en écriture seule) correspondent respectivement à sa sortie standard et sa sortie d'erreur. Ainsi, les deux lignes ci-dessous sont strictement équivalentes :

```
1  fprintf(stderr,"Je vous souhaite bien le bonjour\n");
2  write(2,"Je vous souhaite bien le bonjour\n",33);
```

Duplication des descripteurs de fichiers

Les descripteurs de fichier peuvent être dupliqués.

```
#include <unistd.h>

int dup(int fd);
int dup2(int fd, int newfd);
```

dup() Crée un synonyme pour le descripteur `fd` qui sera le plus petit numéro libre.

dup2() Fait de `newfd` un synonyme pour `fd`. Si `newfd` était un descripteur de ficher déjà ouvert, il est d'abord fermé.

La fonction `dup2()` sera utilisée dans un futur exercice. Elle permet notamment de faire en sorte qu'un descripteur de fichier ouvert deviennent la sortie ou l'entrée standard d'un processus.

Processus parents et descripteurs de fichiers

Les descripteurs de fichiers ouverts sont préservés lors d'un `fork()` mais aussi (sauf contre ordre, cf. `fcntl(2)`) lors d'un appel de type `exec*()`.

Exercice 1.2 *Affichage régulier de la date avec l'heure* ??/20

Écrivez un programme qui affiche 10 fois la date du jour toutes les 3 secondes à partir d'un processus fils exécutant le binaire `date` dans le répertoire /bin avec l'option `u` (`date -u`). Utilisez pour cela la fonction `execl`. Un second processus fils affichera 30 fois toutes les deux secondes le message "Attendre !". Le processus père attendra la fin de ces deux fils avant d'afficher "C'est terminé !" et de s'arrêter.
Modifiez ensuite votre programme pour que les messages vers la sortie standard des processus fils soient dirigés vers un fichiers : `message.log` ouvert par le processus père. Exploitez pour

obtenir ce résultat l'une des deux fonctions `dup` à partir du processus père.
Explicitez la différence de placer la fonction `dup` dans les deux processus fils au lieu du processus père.

Chapitre 2

Communication inter-processus (IPC)

2.1 Signaux

Un signal permet de dérouter à tout moment l'exécution d'un processus, en réponse à une sollicitation externe (du système ou bien d'un autre processus). Cette sollicitation prend simplement la forme d'un signal qui porte un numéro.

A la réception d'un signal un processus peut alors :
– se terminer (comportement par défaut) ;
– ignorer le signal (SIG_IGN) ;
– exécuter une fonction particulière.
Un signal peut être envoyé par une commande shell :

```
[bash]> kill -numéro_du_signal PID
```

Par exemple :

```
[bash]> kill -9 2367
```

Un processus peut envoyer un signal à un autre à l'aide de l'appel système `kill(2)`.

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

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

Des exemples de valeurs de signaux sont donnés dans la tableau 2.1.

Signal	Valeur	Description
SIGUP	1	Émis lors d'une déconnexion
SIGINT	2	Interruption clavier par ^C
SIGQUIT	3	Interruption clavier par ^\
SIGKILL	9	Mort certaine !
SIGSEGV	11	Accès mémoire invalide (Seg. Fault !)
SIGCHLD	20	Reçu à la mort d'un fils

TABLE 2.1 – Quelques signaux Unix.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>

void repondeur(int s) {
  if ( s == SIGINT ) {
    printf( "Recu SIGINT : je reste.\n" );
    signal( SIGINT, repondeur );
  }
  if ( s == SIGQUIT ) {
    printf( "Recu SIGQUIT : je quitte.\n" );
    exit( 0 );
  }
}

int main() {
  signal( SIGINT, repondeur );
  signal( SIGQUIT, repondeur );
  while ( 1 ) {
    sleep( 60 );
  }
}
```

Listing 2.1 – Gestion de signaux

Pour qu'une fonction particulière s'exécute à la réception d'un signal, le processus doit installer une fonction de réponse à ce signal à l'aide de la fonction `signal(2)`.

```c
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Cet appel installe la fonction `handler` comme réponse au signal donné par le premier argument `signum`. La fonction de réponse précédemment installée est renvoyée. Lors de la réception du signal, la fonction `handler` sera appelée ; le processus reprendra son exécution là où elle s'est interrompue lorsque la fonction se terminera. L'argument `handler` peut prendre deux valeurs spéciales :

– SIG_IGN Le signal sera ignoré.
– SIG_DFL Le comportement par défaut sera rétabli.

Un exemple de gestion de signaux est donné dans le listing 2.1.

Exercice 2.1 *Signaux*

1. Écrire un programme C qui affiche en boucle le message "Toc-toc N" toutes les 2 secondes où N est le nombre de fois où le processus a reçu le signal SIGTERM (= 15) depuis son lancement. (Fonction utile : `sleep(3)`)
2. Lancez le programme précédent et envoyez lui des signaux SIGTERM depuis un autre terminal à l'aide de la commande shell `kill`. (Commande utile : `ps(1)`)

2.2 Les tubes sous Unix

Deux processus peuvent échanger des données via des tubes (*pipes* en anglais). Un tube se compose de deux extrémités qui se comportent chacunes comme des fichiers, l'un ouvert en écriture seule, l'autre en lecture seule. Conceptuellement, un tube est une structure FIFO (First In First Out).

FIGURE 2.1 – Un tube.

2.2.1 Tubes anonymes

La création d'un tube sous Unix se fait à l'aide de la fonction `pipe()`.

```
#include <unistd.h>
int pipe(int filedesc[2]);
```

La lecture comme l'écriture dans un tube se fait en mode caractère : il n'est pas possible d'utiliser la fonction lseek() pour se « déplacer » dans un tube. Les seules opérations autorisées sont :
– La lecture d'un bloc de données par la fonction `read(2)` en utilisant le premier descripteur de fichier (d'indice 0) ;
– L'écriture d'un bloc de données par la fonction `write(2)` en utilisant le second descripteur (d'indice 1) ;
– La fermeture de l'une ou l'autre des extrémités par `close(2)`.

Les prototypes des fonctions `read()` et `write()` sont rappelés ci-dessous :

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

int main()
{
  int tube[2];
  char buffer[1024];
  size_t n;
  if ( pipe( tube ) == -1 ) {
    perror("pipe()");
  }
  write( tube[1], "Hello", 5 );
  n = read( tube[0], buffer, 1024 );
  buffer[ n ] = '\0';
  printf( "Lu: [%s]\n", buffer );
  close(tube[0]);
  close(tube[1]);
  exit(0);
}
```

Listing 2.2 – Création et utilisation d'un tube au sein d'un unique processus.

```c
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Remarques importantes :
- La fonction `read()` utilisée avec un *pipe* est bloquante tant qu'il n'y a aucune donnée à lire dans le tube.
- `read()` renvoie immédiatement la valeur 0 (nombre de caractères lus) lorsque l'extrémité en écriture du tube a été fermée.
- Rien ne garantit que le nombre de caractères lu soit égal à la taille précisée lors de l'appel à `read()`, taille qui n'est en fait qu'un maximum demandé. De plus, si 1024 octets ont été écrits dans un tube, un appel à read peut très bien n'en lire que 512 ! Dans ce cas, plusieurs appels sont nécessaires pour récupérer la totalité de ce qui a été écrit.

Un exemple d'utilisation (inutile !) d'un tube est donné dans le listing 2.2. Mais l'intérêt d'un tube réside plutôt dans l'échange de données entre processus ayant un lien de parenté avec celui qui a créé le tube. En effet, lors du clonage de processus les descripteurs de fichiers (créés par `pipe()`, `open()`, `socket()`, etc.) sont hérités par le processus fils. Il est alors possible de faire communiquer le fils avec son père, ou même avec un de ses frères.

Le principe est le suivant (Figure 2.2) : Un processus crée un tube puis appelle `fork()`. Le fils hérite alors des deux descripteurs de fichiers associés au tube et ferme (`close()`) celui dont il ne se servira pas. Le père fait de même avec l'extrémité dont lui ne se servira pas. A ce stade, l'un des processus dispose d'un descripteur ouvert en écriture (`fd[1]`), l'autre possède le descripteur en lecture (`fd[0]`). L'échange de données, dans une seule direction, peut alors

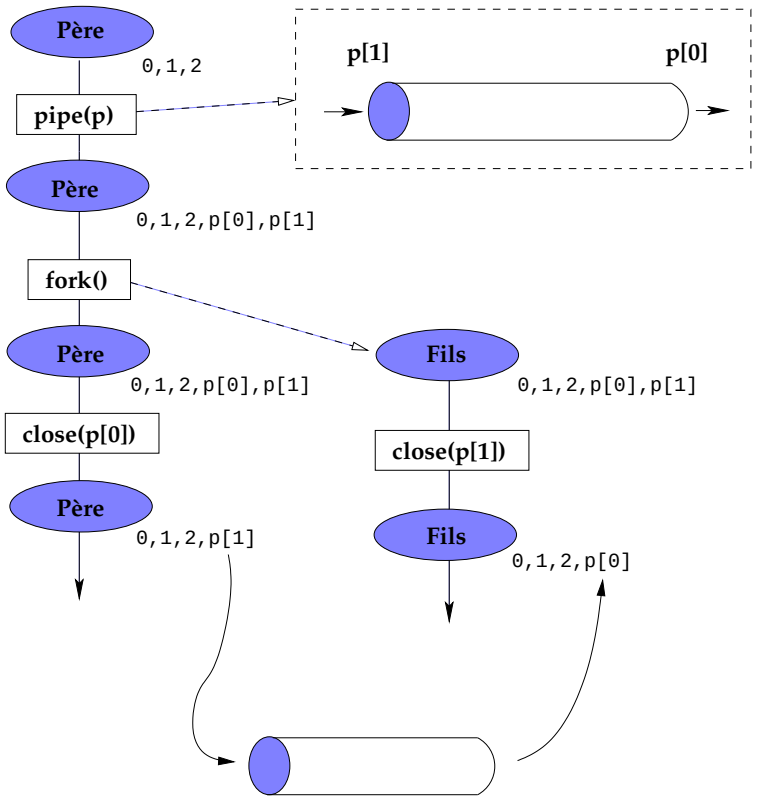


FIGURE 2.2 – Communication père/fils à sens unique à l'aide d'un tube.

commencer.

Quelques remarques :
- Lorsque deux processus partagent des descripteurs d'un même tube, il est souhaitable qu'ils ferment (`close()`) l'extrémité qu'ils n'utilisent pas.
- Un tube disparaît quand ses deux extrémités sont fermées.
- Écrire dans un tube dont l'extrémité en lecture est fermée provoque une erreur et la réception d'un signal SIGPIPE.

2.2.2 Tubes nommés

Si deux processus sans liens de parenté veulent communiquer, ils le peuvent à l'aide de tubes nommés. La création d'un tube nommé peut se faire à l'aide de la commande `mkfifo` qui crée un fichier spécial dans le système de fichier.

```
[bash]> mkfifo nom_tube
```

Elle peut aussi se faire par l'appel `mkfifo()` :

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

int mkfifo(const char *pathname, mode_t mode);
```

Chaque processus peut alors ouvrir le tube comme un fichier classique avec la fonction `open()` et ainsi récupérer un descripteur de fichier qu'il utilisera soit en lecture, soit en écriture. Par

défaut, l'ouverture d'un tube nommé en lecture seule est bloquant tant qu'il n'a pas été ouvert en écriture par un autre processus (et réciproquement). Ensuite, l'utilisation se fait via les fonctions `read()`/`write()` comme pour les tubes anonymes.

2.3 Exercices

2.3.1 Tube nommé

Exercice 2.2 *Tube nommé*

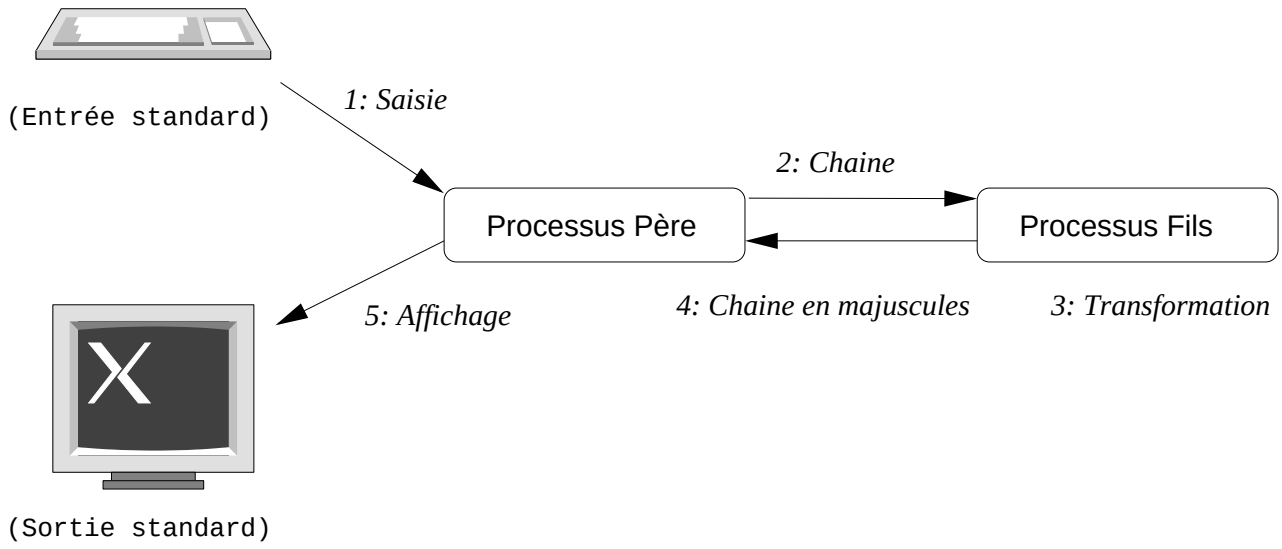
1. Écrire un premier programme appelé `producteur` qui prend comme argument sur sa ligne de commande le nom d'un tube nommé existant. Ce programme envoie dans le tube un caractère aléatoire de l'ensemble $\{'A', \ldots, 'Z'\}$ toutes les secondes.
2. Écrire un second programme appelé `consommateur` qui prend le même type d'argument et extrait 1 à 1 des caractères du tube jusqu'à ce que la même lettre ait été lue deux fois.

Fonctions utiles : `rand(3)`, `srand(3)`, `time(2)` et `sleep(3)`.

2.3.2 Échange père/fils via un tube

Exercice 2.3 *Échange père/fils via des pipes* ??/20

Écrire un programme composé de 2 processus : un processus (le père) lit une chaîne de caractères au clavier et la transmet au second processus (le fils) qui la transforme en majuscule ; le processus père affiche ensuite le résultat. Les deux processus s'arrêtent lorsque l'utilisateur saisit la chaîne « quit ».



Remarques

– Les échanges doivent se faire au plus court. On n'envoie pas 1024 caractères pour une chaîne qui n'en contient que 10.
– N'oubliez pas les remarques importantes de la section 2.2.1.

2.3.3 Échange avec un programme existant via ses entrées/sorties standard

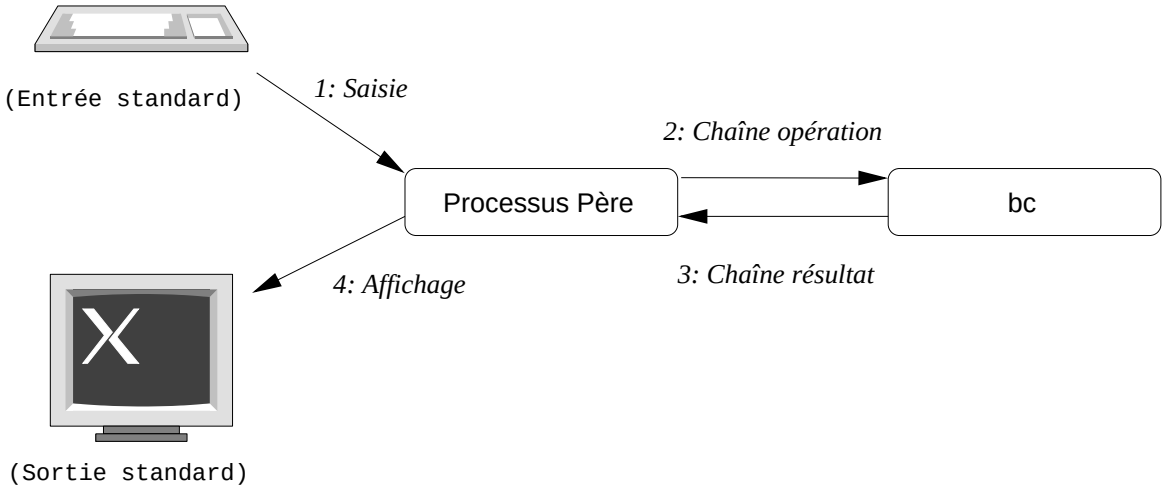
Exercice 2.4 *Échange avec un programme existant* **??/20**

Les systèmes UNIX implémentent une calculatrice en mode ligne à travers la commande `bc`. Par défaut, le programme `bc` lit une chaîne de caractères représentant une opération à effectuer sur l'entrée standard et affiche le résultat sur la sortie standard. La chaîne de caractères « quit » permet de quitter l'application.

```
e102pc01:/home/eleves/vous> bc
2+3
5
quit
e102pc01:/home/eleves/vous>
```

Listing 2.3 – Exemple d'utilisation de la commande `bc`.

Écrire un interpréteur permettant de saisir une opération au clavier et de transmettre celle-ci à un processus correspondant au programme `bc` qui renverra le résultat (cf. figure ci-dessous). La communication entre les deux processus sera réalisée avec des tubes, et l'interpréteur se charge du lancement du processus `bc`. Le processus père communiquera avec `bc` via l'entrée et la sortie standard de ce processus, qui auront été redirigées vers des *pipes* avant que le processus fils ne soit recouvert via l'appel système `execl()`.



Exercice 2.5 *Inteprète de commandes pour un programme de dessin* ??/20

Le programme `board` (`/home/public/SE_et_Reseaux/board`) ouvre une fenêtre graphique et peut lire sur son entrée standard des commandes de dessin rudimentaires (dessiner une ligne ou un cercle, changer la couleur de trait ou la couleur de remplissage). Ces commandes sont précisées dans un format binaire correspondant au type `DrawingCommand` décrit dans le fichier d'entête `/home/public/SE_et_Reseaux/DrawingCommand.h`.

Le fichier `/home/public/SE_et_Reseaux/body.bin` est un exemple de fichier contenant plusieurs *commandes*. Vous pouvez ouvrir ce fichier avec emacs en mode "hexl-mode". Pour visualiser le résultat de ces commandes utilisez :

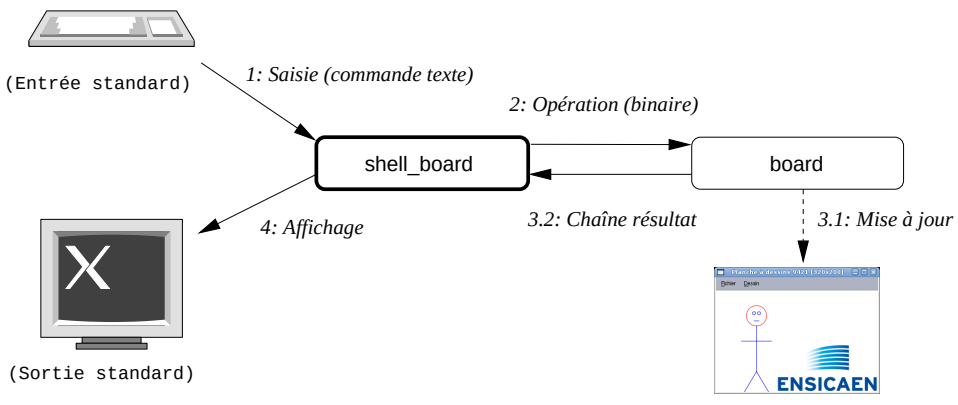
```
e102pc01> cat man.bin | board -
```

1. En regardant la sortie standard de la commande précédente ainsi que la taille du fichier `"body.bin"`, donnez la taille de la structure `DrawingCommand`. Interprétez.

On aimerait pouvoir communiquer avec le programme `board` via un interprète de commandes en mode texte. Pour ce faire, un programme père se chargera de lancer le programme `board` puis de lire des commandes qu'il analysera avant de les envoyer en format binaire par un tube au programme board (voir schéma ci-dessous). L'interaction avec l'interpète pourra prendre la forme suivante :

```
e102pc01> shell_board
shell_board> line 10 10 200 10
Board says: "OK, I drew a line"
shell_board> pen 255 0 0
Board says: "OK, I changed the pen color"
shell_board> fill 0 255 0
Board says: "OK, I changed the fill color"
shell_board> circle 200 100 30
Board says: "OK, I drew a circle"
shell_board> quit
Board says: "OK, I quit!"
e102pc01>
```

2. Écrivez le programme interpréteur `shell_board`. La communication entre ce programme et le programme `board` sera réalisée avec des tubes, et l'interpréteur se charge du lancement du processus `board`. Le processus père communiquera avec `board` via l'entrée et la sortie standard de ce dernier, qui auront été redirigées vers des *pipes* avant que le processus fils ne soit recouvert via l'appel système `execl()`.



2.4 Mémoire partagée sous Unix

La mémoire partagée est un moyen supplémentaire pour échanger des informations entre deux processus. Les processus impliqués peuvent avoir ou non un lien de parenté. En effet, une zone de mémoire partagée possède un identifiant, de type entier, au niveau du système (à la façon d'un fichier qui possède un chemin d'accès et un nom). Ces identifiants peuvent être visualisés par la commande `ipcs`.

Appels systèmes liés à la mémoire partagée

Tous les prototypes de fonctions de gestion de la mémoire partagée sont présents dans les deux fichiers :

```
#include <sys/ipc.h>
#include <sys/shm.h>
```

Création d'un segment de mémoire partagée

```
int shmget(key_t cle, size_t size, int shmflg) ;
```

Attachement d'un segment de mémoire partagée à un processus

```
 char * shmat(int shmid, char *shmaddr, int shmflg);
```

Détachement d'un segment de mémoire partagée du processus

```
int shmdt(char *shmaddr);
```

Manipulation d'un segment de mémoire partagée

```
int shmctl(int shmid, int op, struct shmid_ds *buf);
```

La structure `shmid_ds` est composée de nombreux champs comme par exemple le PID du créateur, la date du dernier attachement, la taille du segment, les permissions associées au segment, etc.

- si op = `IPC_STAT`, lecture de la valeur courante de la structure associée à shmid et copie dans buf.
- si op = `IPC_SET`, copie de buf dans la structure associée à shmid.
- si op = `IPC_RMID`, destruction du segment de mémoire partagée.

2.5 Les sémaphores

2.5.1 Présentation

Plusieurs processus peuvent entrer en conflit pour l'accès à une ressource (fichier, mémoire partagée). Il est aussi parfois souhaitable de limiter le nombre d'accès simultanés à une ressource. Pour ce faire, on peut avoir recours aux *sémaphores*, concept introduit par E.W. Djikstra.

Définition 2 *Un sémaphore est une valeur entière, initialisée avec le nombre d'accès simultanés autorisés. Deux fonctions atomiques sont associées à un sémaphore : P(sem) pour acquérir un accès et V(sem) pour rendre un accès ; ainsi qu'une file d'attente.*
- *Acquérir un accès revient à décrémenter la valeur du sémaphore s'il est* > 0 *; si le sémaphore vaut 0, le processus ayant tenté l'acquisition est stoppé et placé en file d'attente.*
- *Rendre un accès revient à incrémenter la valeur du sémaphore si la file d'attente est vide, sinon la valeur du sémaphore reste nulle et un processus de la file d'attente est réveillé.*

Unix gère les sémaphores à la norme POSIX (proche de la définition des sémaphores de Djikstra) et des sémaphores Unix qui sont en fait des tableaux de sémaphores.

2.5.2 Les sémaphores POSIX

Les sémaphores POSIX sont déclarés dans le fichier d'en-tête `<semaphore.h>`. Deux opérations sont définies sur un sémaphore : l'incrémentation (par la fonction `sem_post(3)`) et la décrémentation (par la fontion `sem_wait(3)`).

Sémaphores nommés Un sémaphore nommé (de la forme "*/un_nom*") est créé ou bien ouvert via la fonction `sem_open(3)`, il peut être fermé grâce à la fonction `sem_close(3)` et détruit par `sem_unlink(3)`.

Sémaphores sans nom Un sémaphores sans nom, de type `sem_t`, est créé à l'aide de la fonction `sem_init(3)` et il est détruit grâce à la fonction `sem_destroy(3)`.

2.5.3 Les sémaphores Unix

Les types et prototypes nécessaires pour manipuler les sémaphores « Unix » sont déclarés dans les deux fichiers :

```
#include <sys/ipc.h>
#include <sys/sem.h>
```

L'appel système `semget()` permet de créer un tableau de sémaphores :

```
int semget(key_t clé, int NombreSéma, int flag)
```

Par exemple :

```
key_t clef;
int id_sem;
clef = ftok("nom_fichier_existant", 'A');
id_sem = semget(clef,
                5,    /* Cration d'un tableau de 5 smaphores */
                IPC_CREAT | 0600); /* Droits d'accs exclusifs */
```

La fonction `ftok()` permet de générer une clef `key_t`, un entier, à partir d'un nom de fichier existant et d'un entier. La clef générée est la même pour un même couple (*nom de fichier*,*entier*) et devrait être différente pour des couples différents.

```
key_t  clef = ftok("nom_fichier_existant", 'A');
```

Cette fonction permet à deux processus de convenir d'une même clef à partir d'un fichier existant et donc d'un objet externe. Par exemple, utiliser le nom du programme exécutable permet de garantir dans une certaine mesure que la clef produite n'est pas déjà utilisée.

La manipulation (initialisation, consultation, destruction) d'un sémaphore est possible grâce à l'appel `semctl()`.

```
int semctl(int idSema, int NumSema, int commande, [union semun param]);
```

Par exemple, l'appel :

```
semctl(id_sem,0,SETVAL,1);
```

demande l'initialisation/affectation avec la valeur 1 du premier sémaphore (position 0) du tableau identifié par `id_sem`.

L'aquisition ou la libération d'un sémaphore (opérations P et V) est possible grâce à l'appel `semop()`.

```
int semop(int id_sem, struct sembuf *sops, size_t nb_ops);
```

Par exemple :

```
struct sembuf op;
op.sem_num = 0 ;    /* Indice du smaphore au sein du tableau */
op.sem_op = -1 ;    /* -1 : Acquisition, 1 : Libration       */
op.sem_flg = 0 ;    /*  0 : Acquisition bloquante             */

semop(semid,&op,1) ;/* Si la valeur est <= 0,                */
                    /* la fonction sera bloquante            */
```

2.5.4 Exercices

Garantie de l'alternance stricte

Exercice 2.6 *Garantie de l'alternance stricte*

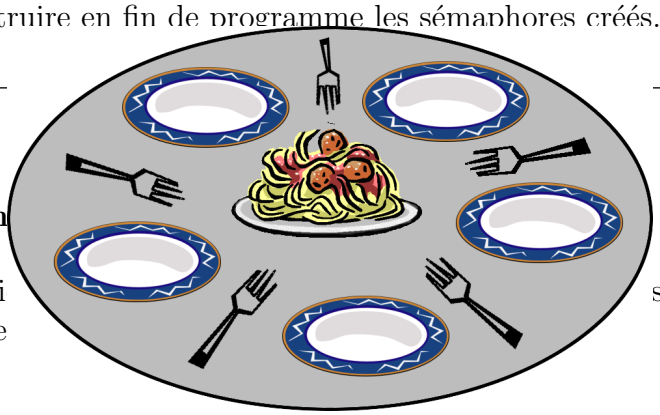
A l'aide des sémaphores Unix, écrire les fonctions `P()` et `V()` (protocole d'acquisition et de libération) telles que définies par Djisktra et tester ces fonctions en lançant deux processus qui affichent chacun 10 fois un message de manière alternée selon le modèle ci-dessous :

```
Je suis le processus père de PID 10050.
Je suis le processus fils de PID 10051.
Je suis le processus père de PID 10050.
Je suis le processus fils de PID 10051.
Je suis le processus père de PID 10050.
... (20 lignes)
```

Conseils : Deux sémaphores sont nécessaires pour assurer l'alternance stricte.

Pensez à détruire en fin de programme les sémaphores créés.

Le dîner des philosoph



Ce problème, grand classi st connu sous le nom du « dîner des philosophe

FIGURE 2.3 – La table du dîner. Pour manger confortablement des spaghettis, il faut 2 fourchettes.

Cinq philosophes sont réunis pour mener à bien deux activités majeures : penser et manger. Chaque philosophe pense pendant un temps aléatoire, mange (si possible) pendant un temps aléatoire puis se remet à penser. Lorsqu'un philosophe demande à manger, une assiette de spaghettis l'attend ; les cinq assiettes sont disposées autour d'une table ronde comme le montre la figure 2.3.

Pour qu'un philosophe puisse manger ses spaghettis, 2 fourchettes sont nécessaires : la sienne (celle de droite) et celle de son voisin de gauche. Naturellement si l'un de ses voisins ou les deux voisins sont déjà en train de manger, le philosophe ne peut pas manger et doit attendre que l'une ou les deux fourchettes occupées se libèrent.

La solution de ce problème doit être optimale : un philosophe voulant mais ne pouvant pas manger doit attendre son tour sans consommer de CPU inutilement. Par ailleurs on considèrera que si un philosophe ne dispose pas des 2 fourchettes, il n'en prendra aucune et attendra que les 2 fourchettes soient libres.

Chaque philosophe effectue plusieurs fois le cycle Pense → Demande à manger → Mange.

On doit naturellement constater que deux philosophes voisins ne mangent jamais en même temps.

Exercice 2.7 *Simulation du dîner des philosophes*

??/20

Afin de résoudre le problème décrit, chaque philosophe sera représenté par un processus. On associera un *sémaphore à chaque philosophe* afin de profiter de sa file d'attente (un philosophe désirant manger mais ne le pouvant pas se placera dans la file d'attente de son sémaphore). L'état des philosophes sera stocké dans un *segment de mémoire partagée* (donc visible par les 5 processus) ; les primitives d'utilisation de la mémoire partagée ont étés données précédemment. Ce segment de mémoire partagée devra être considéré comme une ressource critique (et donc protégé par un sémaphore).

- Un philosophe doit utiliser les deux fourchettes situées de chaque côté de lui pour pouvoir manger.
- A chaque changement d'état d'un philosophe, on devra afficher l'état des cinq philosophes. On utilisera pour cela le segment de mémoire partagée.

Le résultat à obtenir pourra être de la forme :

```
[PPPPP] Philosophe 0 : Pense
[PPPPP] Philosophe 1 : Pense
[PPPPP] Philosophe 2 : Pense
[PPPPP] Philosophe 3 : Pense
[PPPPP] Philosophe 4 : Pense
[PPPDP] Philosophe 3 : Demande a manger
[PPPMP] Philosophe 3 : Mange
[PDPMP] Philosophe 1 : Demande a manger
[PMPMP] Philosophe 1 : Mange
[DMPMP] Philosophe 0 : Demande a manger
[DMPMP] Philosophe 0 : Ne peux pas manger
[DMPMD] Philosophe 4 : Demande a manger
...
```

La première colonne représente l'état de chacun des 5 philosophes et permet de vérifier simplement que deux philosophes voisin ne mangent jamais en même temps ; avec la convention suivante : **P**ense, **M**ange, **D**emande à manger (c.-à-d. en attente d'au moins une fourchette).

Exercice 2.8 *Dîner des philosophes version 2*

??/20

Programmez la simulation du dîner des philosophes en associant cette fois les sémaphores aux fourchettes.

– L'acquisition des sémaphores sera notamment faite de manière atomique sur deux sémaphores à la fois. (Un philosophe essaie de saisir les deux fourchettes à la fois.)
– Un segment de mémoire partagée, bien que devenu inutile, sera conservé pour les besoins d'affichage des états des philosophes.

Chapitre 3

Processus légers

3.1 Notion de *thread*

Dans de nombreuses applications, il est souvent nécessaire de réaliser plusieurs tâches de manière simultanée. C'est par exemple le cas dans une interface graphique qui ne doit pas se figer lorsqu'un travail qui prend du temps est en cours. Souvent, il est aussi possible de décomposer un algorithme en tâches qui peuvent s'exécuter en parallèle, ce qui permet de réduire le temps d'exécution de l'algorithme pour peu que chaque tâche s'exécute sur un cœur ou processeur différent.

Un thread, ou processus léger [1], est un fil d'exécution au sein d'un processus. Ainsi, deux threads au sein d'un même processus partagent le même espace mémoire et les mêmes ressources. Pour ce qui est de la mémoire occupée, seule la pile est propre à chaque thread (cf. cours).

Le minimum d'information concernant l'état d'un thread est la valeur des registres et en particulier celle du compteur ordinal (*IP : Instruction Pointer*). C'est aussi l'adresse du haut de la pile qui contient les valeurs des variables temporaires ou locales aux fonctions ainsi que les adresses de retour de ces fonctions (notion de fenêtre ou *frame*).

Sous Unix, la bibliothèque pthread (POSIX *threads*) permet d'utiliser les threads. Il faut ajouter l'option `-lpthread` lors de la compilation.

La création d'un thread passe par l'utilisation de la fonction `pthread_create()`.

```
#include <pthread.h>

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

où :
- `thread` est l'identifiant de thread renvoyé par la fonction ;
- `attr` définit les attributs à appliquer au thread (NULL pour un comportement par défaut) ;
- `start_routine` est la fonction qui sera exécutée par le processus léger ;
- `arg` est le pointeur qui sera donné en argument à la fonction `start_routine`.

1. *light-weight process*

La fonction `pthread_create()` renvoie 0 si le thread a bien été créé, et un code différent de 0 en cas d'erreur.

Un thread se termine lorsque la fonction `start_routine` retourne ou bien lorsqu'il appelle la fonction `pthread_exit` :

```
void pthread_exit(void *return_value);
```

Enfin, un thread peut attendre la fin de l'exécution d'un autre thread en faisant appel à la fonction `pthread_join`.

```
int pthread_join(pthread_t th, void **thread_return);
```

La valeur de retour du thread, de type `void*`, est stockée à l'emplacement donné par le paramètre `thread_return`.

A noter que la mort du thread principal d'un processus entraîne la mort de tous les autres threads.

3.2 Exemple

Le listing 3.1 montre un exemple de code utilisant un thread.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  char message[] = "Bonjour";
5
6  void * thread_function( void *arg )
7  {
8     printf( "thread demarre avec %s\n", (char *) arg );
9     sleep( 3 );
10    strcpy( message, "Bye" );
11    pthread_exit( "au revoir" );
12 }
13
14 int main()
15 {
16    pthread_t a_thread;
17    void *thread_result;
18    if ( pthread_create( &a_thread, NULL,
19                         thread_function,
20                         (void *) message ) < 0 ) {
21       perror( "erreur creation du thread" );
22       exit( EXIT_FAILURE );
23    }
24    printf( "On attend la fin du thread\n" );
25    if ( pthread_join( a_thread, &thread_result ) < 0 ) {
26       perror( "erreur join" );
27       exit( EXIT_FAILURE );
28    }
29    printf( "Fin du join, le thread a retourne %s\n",
```

```
30              (char*) thread_result );
31      printf( "message contient maintenant : %s\n", message );
32      exit( EXIT_SUCCESS );
33  }
```

Listing 3.1 – Exemple de code utilisant un thread.

3.3 Synchronisation de threads

Puisque les threads d'un même processus partagent le même espace mémoire, il est très fréquent qu'ils essaient d'accéder « simultanément » aux mêmes zones mémoire. Sans précaution, le résultat est imprévisible. Un exemple de code erroné est donné dans le listing 3.2.

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  int count = 0;
5
6  void * counting( void * arg )
7  {
8    int i, a;
9    for ( i = 0; i < 1000000; i++ ) {
10     a = count;
11     a *= 3;
12     a = ( a / 3 ) + 1;
13     count = a;
14   }
15   return 0;
16 }
17
18 int main()
19 {
20   int res1, res2;
21   pthread_t a_thread1, a_thread2;
22   void *thread_result1, *thread_result2;
23   res1 = pthread_create( &a_thread1, NULL, counting, NULL );
24   res2 = pthread_create( &a_thread2, NULL, counting, NULL );
25   printf( "On attend la fin des threads\n" );
26   pthread_join( a_thread1, &thread_result1 );
27   pthread_join( a_thread2, &thread_result2 );
28   printf( "count = %d\n", count );
29   return 0;
30 }
```

Listing 3.2 – Exemple de code erroné.

3.3.1 Exclusion mutuelle à l'aide d'un mutex

Un mutex est fonctionnellement équivalent à un sémaphore binaire initialisé à la valeur 1, ce qui revient en effet à autoriser l'accès à une ressource en exclusion mutuelle.

Pratiquement, un thread peut essayer d'obtenir un verrou sur le mutex. Si aucun autre thread ne possède ce verrou, alors il l'obtient sinon il est simplement mis en attente dans une liste. En effet, un mutex ne peut être verrouillé que par un thread à la fois.

Un mutex est une variable de type `pthread_mutex_t`. L'initialisation d'un mutex est possible grâce à la constante prédéfinie `PTHREAD_MUTEX_INITIALIZER` ou bien grâce à la fonction `pthread_mutex_init`. La destruction d'un mutex est réalisée à l'aide de la fonction dédiée `pthread_mutex_destroy`.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init( pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *mutexattr );

int pthread_mutex_destroy( pthread_mutex_t *mutex );
```

Le verrouillage et déverrouillage d'un mutex sont réalisés par les deux fonctions :

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Exercice 3.1 *Exclusion mutuelle à l'aide d'un mutex*

1. Recopiez le code du listing 3.2, exécutez le et constatez l'erreur produite.
2. Corrigez le code du listing 3.2 à l'aide d'un mutex.

3.3.2 Exclusion mutuelle à l'aide d'un sémaphore POSIX

Le listing 3.3 montre comment il est possible de corriger le problème du listing 3.2 à l'aide des sémaphores POSIX.

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <semaphore.h>
4  int compteur = 0;
5  sem_t sem;
6  void * counting( void *arg )
7  {
8     int i, a;
9     for ( i = 0; i < 1000000; i++ ) {
10      sem_wait( &sem );
11      a = compteur;
12      a *= 3;
13      a = ( a / 3 ) + 1;
14      compteur = a;
15      sem_post( &sem );
16    }
17 }
18 int main()
19 {
20   int res1, res2;
21   pthread_t a_thread1, a_thread2;
22   void *thread_result1, *thread_result2;
23   sem_init( &sem, 0, 1 );
24   res1 = pthread_create( &a_thread1, NULL, counting, NULL );
25   res2 = pthread_create( &a_thread2, NULL, counting, NULL );
26   printf( "On attend la fin des threads\n" );
27   pthread_join( a_thread1, &thread_result1 );
28   pthread_join( a_thread2, &thread_result2 );
29   sem_destroy( &sem );
30   printf( "compteur = %d\n", compteur );
31 }
```

Listing 3.3 – Exemple de code correct à l'aide d'un sémaphore.

3.4 Exercices

Exercice 3.2 *Compteurs*

??/20

Un « compteur » est un thread auquel on associe un nom (Toto par exemple) et qui compte de 1 à N (nombre entier positif quelconque). Il marque une pause aléatoire entre chaque nombre (de 0 à 2000 millisecondes par exemple). Le nom du thread et le nombre N sont précisés lors de la création du thread.

Un compteur affiche chaque nombre (Toto affichera par exemple, « Toto : 3 ») et il affiche un message du type « Toto a fini de compter jusqu'à 10 » quand il a fini.

1. Écrivez un programme qui lance 3 compteurs.
2. Modifiez le programme précédent pour que chaque compteur affiche son ordre d'arrivée. Par exemple en affichant un message du type : « Toto a fini de compter jusqu'à 10 en position 4 ». (Conseil : La variable « ordre d'arrivée » devra être manipulée en exclusion mutuelle par les threads.)

Exercice 3.3 *Produit matriciel* multi-threadé

??/20

On donne dans le listing 3.4 le code en langage C permettant d'allouer une matrice de *rows* lignes et *columns* colonnes. (L'organisation mémoire obtenue est schématisée dans la figure 3.1.)

1. Écrire une fonction `random_matrice()` qui affecte aléatoirement les coefficients d'une matrice de taille donnée.
2. Écrire une fonction `product()` qui effectue le produit de deux matrices carrées de taille donnée.
3. Écrire une fonction `product_threaded()` qui effectue le produit de deux matrices carrées de taille donnée à l'aide de 4 threads. Chaque thread se chargera du calcul d'un quart de la matrice résultat.
4. A l'aide du programme `time`, comparez les temps d'exécution des deux fonctions précédentes (avec les même données). Dans quelles conditions peut on espérer observer un gain de temps d'exécution ?

```
1  double ** alloc_matrix(unsigned int rows, unsigned int columns)
2  {
3    double **matrix;
4    unsigned int row;
5    matrix = (double**) calloc( rows, sizeof(double*) );
6    if ( !matrix ) return NULL;
7    matrix[0] = (double*) calloc( rows*columns, sizeof(double) );
8    if ( !matrix[0] ) {
9      free( matrix );
10     return NULL;
11   }
12   for (row = 1; row < rows; ++row )
13     matrix[row] = matrix[row-1]+columns;
14   return matrix;
15 }
16
17 void free_matrix( double **matrix )
18 {
19   free( matrix[0] );
20   free( matrix );
21 }
```

Listing 3.4 – Fonctions d'allocation et de libération d'une matrice.

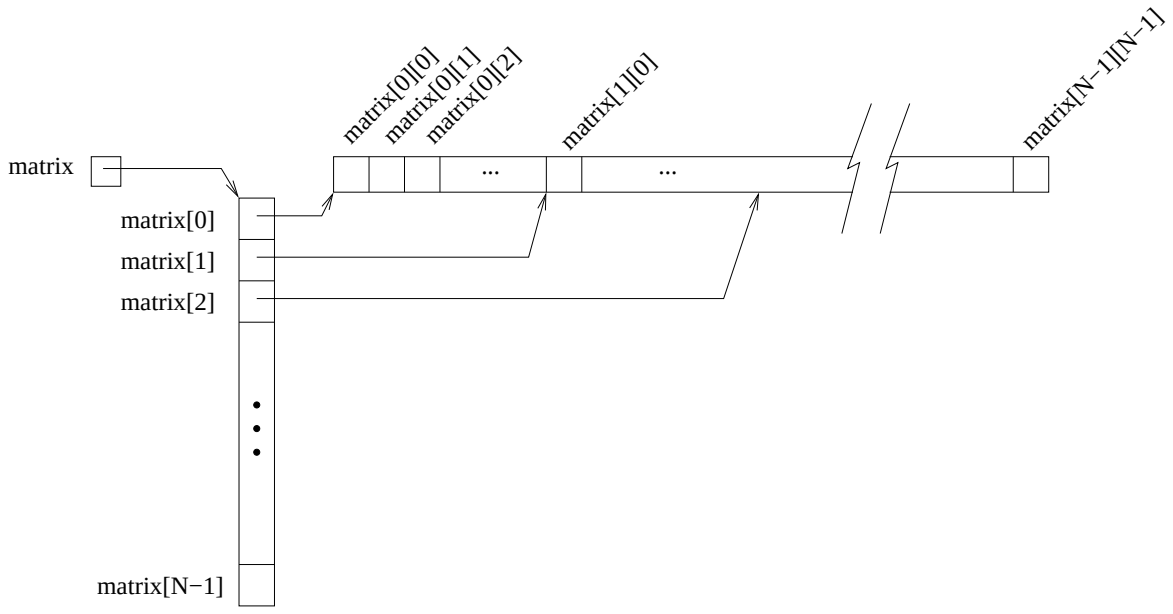


FIGURE 3.1 – Organisation en mémoire d'un matrice de doubles.