

Programmation système

Christine Solnon

Table des matières

1 Les processus	2
1.1 Etats d'un processus	2
1.2 Descripteur d'un processus (PCB)	3
1.3 Contrôle des processus	4
1.3.1 Création et identification de processus	4
1.3.2 Terminaison de processus	5
1.3.3 Exécution d'un nouveau programme	6
1.4 Communication entre processus	7
2 Les signaux	7
2.1 Envoyer un signal avec <code>kill</code>	7
2.2 Interceptor des signaux avec la primitive <code>signal</code>	8
2.3 Attendre un signal	9
3 Les tubes	9
3.1 Retour sur le système de gestion de fichiers	9
3.2 Création d'un tube	10
3.3 Fermeture d'une extrémité d'un tube	10
3.4 Lecture et Ecriture dans un tube	10
4 Les IPC	12
4.1 Identification des IPC par clés	13
4.2 Listage et destruction d'IPC	13
4.3 Les segments de mémoire partagée	13
4.4 Les sémaphores	17
5 Les processus légers ou "Threads"	19
5.1 Contrôle de threads	20
5.2 Verrous d'exclusion mutuelle (mutex)	22
5.3 Les variables de condition	24

1 Les processus

Un processus peut être défini comme “une instance de programme en cours d’exécution” :

- le programme est un code statique unique (une suite d’instructions) qui est chargé en mémoire centrale à partir du moment où il doit être exécuté ;
- l’exécution d’un programme correspond à l’exécution séquentielle de ses instructions ; on utilise un compteur ordinal (instruction pointer) pour mémoriser, lors de l’exécution du programme, l’adresse de la prochaine instruction à exécuter ;
- une instance d’un programme en cours d’exécution désigne une exécution particulière d’un programme : le même programme peut être exécuté par plusieurs processus “en même temps” ; chaque exécution correspond à une instance différente et donc à un processus différent.

Chaque processus est créé par un et un seul processus, appelé processus père. Un seul processus n’a pas de père : le processus “init”, qui est l’ancêtre commun à tous les autres processus, et qui est créé à l’initialisation du système par le super-utilisateur (root).

Le système d’exploitation maintient une table des processus qui contient une entrée pour chaque processus existant. Quand un nouveau processus est créé, le système d’exploitation

1. charge en mémoire le code à exécuter (si cela n’a pas déjà été fait pour un autre processus exécutant le même programme) et alloue un nouvel emplacement mémoire pour les données et la pile du processus,
2. crée le “descripteur” ou “bloc de contexte du processus” (PCB = Process Control Bloc) qui contient toutes les informations relatives au processus,
3. ajoute dans la table des processus une nouvelle entrée qui pointe sur ce PCB,
4. insère le nouveau processus dans la liste de ses processus prêts à être exécutés.

On va voir tout d’abord approfondir les notions d’état et de descripteur d’un processus. On étudiera ensuite les primitives système permettant de contrôler (création, destruction, ...) des processus.

1.1 Etats d’un processus

L’ordinateur a des ressources limitées (en particulier le temps CPU, puisqu’en général on travaille sur une machine mono-processeur), de sorte que les processus se retrouvent en compétition pour ces ressources : il s’agit de partager équitablement les ressources entre les processus qui en ont besoin. Cette gestion est effectuée notamment par le “scheduler” (ordonnanceur de processus), qui est un processus qui gère les autres processus en fonction de leurs priorités et de leurs besoins. Pour cela, le scheduler distingue différents états de processus, et fait passer les processus d’un état à un autre. Pour chaque état, il gère la liste des processus étant dans cet état (avec une file de priorité).

Les différents états possibles pour un processus sont :

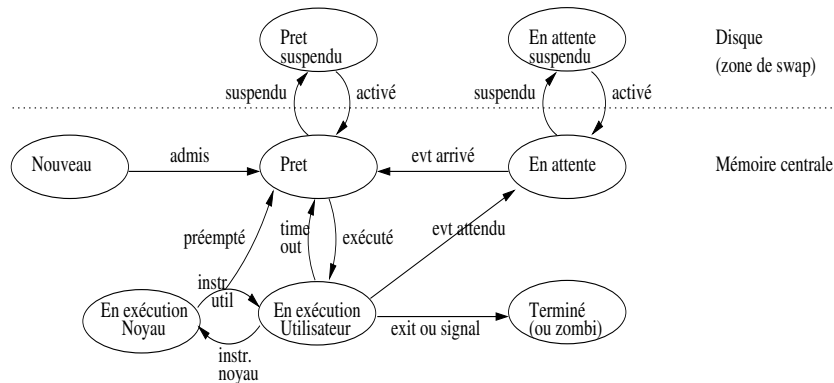
- Nouveau, quand on demande l’exécution d’un programme (naissance d’un processus)
- En exécution (élu), quand l’unité centrale est en train d’exécuter le processus (incrémenter le compteur ordinal)
- Prêt (elligible), quand le processus ne s’exécute pas, mais qu’il n’a besoin que de CPU pour s’exécuter
- En attente, quand le processus ne s’exécute pas car il attend une ressource autre que du CPU (E/S, attente d’un message d’un autre processus, ...)
- Terminé, quand le processus a fini de s’exécuter

Par ailleurs, les processus prêts et en attente sont stockés en mémoire centrale, et lorsqu’il y a trop de processus (de sorte qu’ils ne peuvent pas tous être stockés en mémoire centrale), certains processus prêts ou en attente sont mis dans la zone de swap du disque. On dit alors qu’ils sont dans l’état “suspendu prêt” ou “suspendu en attente”.

Enfin, le scheduler distingue également deux états différents pour les processus en exécution, en fonction du mode de l’instruction courante. En effet, on distingue deux types d’instructions : les instructions “privilégiées” (qui sont généralement utilisées pour contrôler d’autres processus) sont dites en mode noyau ; les autres instructions sont dites en mode utilisateur. Ainsi, le scheduler

distingue l'état "en exécution utilisateur" de l'état "en exécution noyau", selon le mode de l'instruction courante du processus en train de s'exécuter. Notons finalement que les processus qui exécutent des instructions en mode utilisateur sont interrompus lorsque leur quantum de temps est écoulé (ils passent alors dans l'état "prêt"). En revanche, les processus qui exécutent des instructions système, en mode noyau, n'ont pas de quantum de temps et ne sont donc jamais interrompus. Lorsque l'instruction suivant une instruction en mode noyau se trouve être une instruction en mode utilisateur, on a alors deux possibilités : en général, le processus est basculé dans l'état "exécution en mode utilisateur" ; cependant, le processus peut être préempté (notamment si un processus dans l'état prêt a une priorité plus forte que celle du processus en exécution) ; dans ce cas, le processus passe dans l'état préempté, puis dans l'état prêt.

Les transitions possibles d'un état à l'autre sont décrites par le graphe suivant :



La commande `nice` permet d'exécuter un programme avec une priorité d'ordonnancement modifiée : la priorité par défaut est 0 ; elle peut être ajustée dans l'intervalle -20 (le plus prioritaire) à 19 (le moins prioritaire). La priorité d'un processus peut aussi être modifiée au cours de la vie du processus à l'aide de la commande `renice`.

1.2 Descripteur d'un processus (PCB)

Le système d'exploitation maintient une table des processus qui contient une entrée pour chaque processus existant. Les informations contenues dans cette table sont appelées "descripteur du processus" ou bloc de contexte (PCB = Process Control Bloc). Ainsi, pour chaque processus, le bloc de contexte associé contient :

- l'image mémoire du processus : il s'agit de pointeurs sur les segments de code (le programme), de données (le tas) et de pile du processus
- l'état du processus (prêt, en attente, en exécution, ...)
- les numéros identifiants qui servent à le repérer parmi les processus :
 - le numéro du processus (`pid`)
 - le numéro du processus père qui a créé le processus (`ppid`)
 - le numéro du groupe de processus auquel appartient le processus (`pgid`)

Un nouveau groupe de processus est créé lorsqu'on lance une suite de commandes reliées par des tubes/pipes, ou une suite de commandes entre parenthèses ; le premier processus d'un groupe est dit "processus leader" ; tous les processus du groupe ont pour numéro de `gid` le numéro de `pid` du processus leader du groupe.
- le numéro de la session à laquelle appartient le processus (`sid`)
- Une nouvelle session est créée à la connexion de l'utilisateur ou lorsqu'on ouvre un nouveau terminal (`tty`) ; le premier processus d'une session est dit "processus leader" ; tous les processus lancés depuis une session ont pour numéro de `sid` le numéro de `pid` du processus leader de la session.
- les numéros identifiant les droits associés au processus
 - le numéro de l'utilisateur qui a lancé le processus (`uid`),

Par défaut un processus s'exécute avec les droits de l'utilisateur qui l'a lancé.

 - le numéro du groupe de l'utilisateur qui a lancé le processus (`gid`),

- le numéro de l'utilisateur effectif du processus (`euid`),
Un processus peut temporairement s'exécuter avec les droits d'un autre utilisateur (appelé utilisateur effectif) que celui qui l'a lancé. Pour cela, le programme doit faire un appel à la primitive système `setuid(new-uid)` (sous réserve que le propriétaire du programme soit `root` ou `new-uid`).
- le numéro du groupe effectif de l'utilisateur du processus (`egid`),
Idem `euid` mais au niveau du groupe.
- le contenu des registres de l'unité centrale, y compris le compteur ordinal (qui donne l'adresse de la prochaine instruction à exécuter),
- la table des descripteurs de fichiers associés au processus,
Cette table contient une entrée pour chaque fichier ouvert par le processus. A la création du processus, elle contient par défaut trois entrées :
 - le descripteur de fichiers numéro 0 désigne l'entrée standard du processus, c'est-à-dire le flux des données communiquées par l'utilisateur au processus via le clavier ; ce descripteur est associé au fichier `/dev/stdin`,
 - le descripteur de fichiers numéro 1 désigne la sortie standard du processus, c'est-à-dire le flux de données communiquées par le processus à l'utilisateur via l'écran ; ce descripteur est associé au fichier `/dev/stdout`,
 - le descripteur de fichiers numéro 2 désigne la sortie des erreurs du processus, c'est-à-dire le flux de messages d'erreurs communiqués par le processus à l'utilisateur via l'écran ; ce descripteur est associé au fichier `/dev/stderr`.
- des informations pour le scheduler : priorité du processus, temps CPU récemment consommé, ...
- des informations pour le gestionnaire de fichiers : masque permettant de définir les droits d'accès des fichiers créés, répertoire courant, racine courante, ...
- les signaux envoyés au processus (par d'autres processus), en attente de traitement.

La commande `ps` affiche à l'écran la liste des processus en cours, avec leur numéro de `pid`, leur état, Elle permet notamment de voir si un processus est prêt à être exécuté (R pour running), en attente (S pour sleeping) ou à l'état de zombie (Z) ; s'il est en zone de swap, l'état sera suivi d'un W.

1.3 Contrôle des processus

Unix fournit un certain nombre de primitives système (des instructions en langage C) permettant de contrôler les processus, à savoir : `fork`, `getpid`, `getppid`, `exit`, `wait*` et `exec*`. Ces primitives sont définies dans les fichiers `unistd.h` et `sys/wait.h`, et les types qu'elles utilisent sont définis dans le fichier `sys/types.h`. Il faudra donc faire un `include` de ces trois fichiers avant d'utiliser ces primitives.

1.3.1 Création et identification de processus

La primitive

```
pid_t fork(void)
```

crée un nouveau processus. Le processus appelant le `fork` est appelé "processus père" ; le processus créé est appelé "processus fils".

Au moment de la création d'un nouveau processus par la primitive `fork`, le système crée une nouvelle entrée dans la table des processus. Cette nouvelle entrée pointe sur le PCB du nouveau processus. Ce nouveau PCB est créé par copie de celui du père, sauf pour

- le numéro de `pid`, qui prend pour valeur un nouvel entier,
- le numéro de `ppid` qui prend pour valeur le numéro de `pid` du père,
- l'image mémoire : le système alloue de la mémoire pour les données (le tas) et la pile du fils et y recopie les informations contenues dans la pile et le tas du père ; les pointeurs de pile et de tas du fils prennent donc pour valeur les adresses de ces emplacements nouvellement alloués.

Notons bien qu'après le `fork` les deux processus exécutent le même code, et que le fils hérite du compteur ordinal du père. Par conséquent, la première instruction qui sera exécutée par le fils est

celle qui suit l'appel au `fork` qui l'a créé (et la prochaine instruction du père sera la même). En général, cette instruction consiste à tester la valeur retournée par le `fork...` pour savoir si on est père ou fils : la fonction `fork()` renvoie la valeur 0 chez le fils, et le numéro de `pid` du fils chez le père. S'il n'y a plus assez de mémoire pour allouer le nouveau PCB, ou plus d'entrée disponible dans la table des processus, `fork()` renvoie -1.

Ainsi, on aura généralement la séquence d'instructions suivante :

```
pid_t pid_fils;
pid_fils=fork();
switch(pid_fils){
  case -1:
    perror("Erreur à l'appel de fork");
    exit(1);
  case 0:
    // code à exécuter par le fils
  default:
    // code à exécuter par le père
}
```

Les primitives

```
pid_t getpid(void)
pid_t getppid(void)
```

retournent respectivement le numéro de `pid` du processus et le numéro de `pid` du père du processus.

1.3.2 Terminaison de processus

Les primitives

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

permettent de gérer la fin des processus :

- Pour le fils, la primitive `exit` permet de terminer l'exécution en renvoyant au père la valeur passée en paramètre. Par convention, la valeur zéro indique que le processus s'est terminé normalement, tandis qu'une valeur différente de zéro indique une erreur. Si le père attend cette valeur de sortie (à l'aide des primitives `wait` ou `waitpid`), alors le fils passe dans l'état "terminé", son PCB est détruit et l'entrée correspondante dans la table des processus est supprimée; sinon il passe dans l'état "zombie" et son PCB reste dans la table des processus (jusqu'à ce que son père se décide à l'attendre, ou qu'il soit adopté par le processus `init` qui effectuera le `wait` afin qu'il puisse mourir en paix...).
- Pour le père, les primitives `wait` et `waitpid` suspendent l'exécution jusqu'à ce qu'un fils se termine (ou jusqu'à ce qu'un signal à intercepter arrive; dans ce cas, il traite le signal si c'est possible puis se remet en attente). Si un processus fils s'est déjà terminé au moment de l'appel (il est devenu "zombie"), la fonction `wait` s'exécute immédiatement, et le zombie disparaît (son PCB est enfin détruit).

Les deux primitives `wait` et `waitpid` retournent le `pid` du fils qui s'est terminé; dans les deux cas `*status` contient la valeur du paramètre passé par le fils dans la fonction `exit`.

La différence entre `wait` et `waitpid` est que `wait` attend la fin de n'importe lequel de ses fils (le premier qui termine), alors que `waitpid` attend la fin du fils dont le `pid` est passé en premier paramètre. Par ailleurs, on peut passer des options en paramètre à `waitpid`, comme par exemple l'option `WNOHANG` qui permet au père de ne pas suspendre son exécution : si le fils est déjà zombie au moment de l'exécution de `waitpid` avec l'option `WNOHANG`, alors le fils passe dans l'état terminé, et le père continue son exécution; sinon le père continue son exécution.

Exemple

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

int main(void){
    int etat;
    pid_t pid_fils;
    pid_fils=fork();
    switch(pid_fils){
        case -1:
            perror("Erreur à l'appel de fork");
            exit(1);
        case 0:
            printf("Mon PID est %d, et je suis le fils de %d\n",getpid(),getppid());
            exit(0);
        default:
            printf("Mon PID est %d, et j'ai créé un processus fils dont le PID est %d\n",getpid(),pid_fils);
            fils = wait(&etat);
            printf("Mon fils %d a fini de s'exécuter et est sorti avec %d\n",fils,etat);
            exit(0);
    }
}
```

1.3.3 Exécution d'un nouveau programme

Les primitives

```
int execl (const char *path, const char *arg, ...)
int execlp (const char *file, const char *arg, ...)
int execl_e (const char *path, const char *arg , ..., char * const envp[])
int execv (const char *path, char *const argv[])
int execvp (const char *file, char *const argv[])
```

permettent de changer le programme exécuté par un processus : elles remplacent l'image du programme en cours par un nouveau programme. Notons que le numéro de `pid` du processus n'est pas modifié par la fonction `exec`. Pour toutes ces primitives, le premier argument `path` est le chemin d'accès au fichier à exécuter (chemin absolu pour `execl`, `execl_e` et `execv`, et chemin relatif pour les autres). Pour les primitives `execl`, `execlp` et `exec`, les arguments suivant le `path` sont, dans l'ordre :

- le nom de la commande (correspondant à `argv[0]` en C et `$0` en shell)
- les arguments de la commande (correspondant à `argv[i]` en C et `$i` en shell)
- NULL

Pour les primitives `execv` et `execvp`, le nom et les arguments de la commande à exécuter sont passés en paramètre dans le tableau `argv`.

Enfin, la primitive `execl_e` permet de changer les variables d'environnement du processus.

Par exemple, pour exécuter `grep -c extern /usr/include/stdio.h`, on appellera

```
execl("/bin/grep","grep","-c","extern","/usr/include/stdio.h",NULL)
```

ou

```
execlp("grep","grep","-c","extern","/usr/include/stdio.h",NULL)
```

ou

```
execv("/bin/grep",argv)
```

où `argv` est un tableau de chaînes de caractères dont le premier élément contient `"grep"`, le deuxième `"-c"`, le troisième `"extern"` le quatrième `"/usr/include/stdio.h"` et le cinquième `NULL`.

En cas de réussite, une primitive `exec*()` ne revient pas dans le code du programme appelant car les segments de code, de données, ainsi que la pile d'exécution présents dans le PCB du processus appelant `exec` sont remplacés par ceux du programme chargé. Par conséquent, en cas de réussite, `exec*()` ne retourne pas de valeur, et les lignes de code suivant l'appel à `exec*()` ne sont pas exécutées. En cas d'erreur d'un appel à `exec*()` (par exemple si la commande appelée n'existe pas), la fonction retourne `-1`.

1.4 Communication entre processus

Lorsqu'un processus veut échanger des informations avec un autre processus, une première solution consisterait à utiliser un fichier texte, ouvert en écriture par le processus qui veut donner des informations, et en lecture par celui qui veut recevoir des informations... si cette solution est techniquement possible, elle pose des problèmes de synchronisation, car le processus lecteur ne sait pas quand il peut lire les informations.

On va maintenant voir trois solutions pour échanger correctement des données entre des processus : les signaux, les tubes et les IPC.

2 Les signaux

Un signal est un "message très court" qu'un processus peut envoyer à un autre, sous réserve que le processus destinataire accepte les signaux de ce processus. Un processus accepte les signaux venant de processus ayant le même numéro d'`uid` que lui (autrement dit, appartenant au même utilisateur) ; il accepte également les signaux venant de processus appartenant au super utilisateur (root).

Par défaut, le processus recevant un signal termine brutalement son exécution... sauf s'il est un zombie (car il est déjà mort!). Cependant, le processus receveur peut, s'il le souhaite, intercepter le signal et mettre en œuvre une réponse prédéfinie. On appelle handler la fonction définie par l'utilisateur qui sera exécutée à la réception d'un signal pour y répondre. Ce mécanisme de prise en compte des signaux est implémenté par un moniteur, au niveau du système, qui scrute en permanence l'apparition de signaux.

Notons que ce mécanisme permet par exemple de faire de la programmation événementielle, à la différence qu'ici les événements peuvent être provoqués par des erreurs internes (violation de pages mémoire, erreurs d'entrée/sortie, ...), et un nombre limité d'actions utilisateur (interruption via les touches `[ctrl C]`, `[ctrl \]`, ...), tandis qu'en programmation événementielle, les événements proviennent toujours d'actions de l'utilisateur et il existe un grand nombre d'actions possibles...

2.1 Envoyer un signal avec kill

On utilise la commande `kill` du shell pour envoyer un signal à un processus de façon interactive (cf le man du Shell). On peut aussi envoyer un signal depuis un programme C en utilisant la primitive

```
int kill(pid_t pid,int signal)
```

Cette fonction envoie l'évènement numéro `signal` au processus de numéro `pid` ; si `pid=0`, alors tous les processus appartenant au même utilisateur que le processus émetteur recevront cet évènement.

Enfin, notons que certains signaux sont envoyés par l'unité centrale ou le système d'exploitation à l'occasion d'évènements divers (problèmes d'entrée/sortie, division par zéro, frappe de certains caractères de contrôle, ...).

La liste complète des signaux définis par le système est donnée par la commande Shell `kill -l`; ces signaux sont définis dans `signal.h`. Les principaux signaux à connaître sont :

- **SIGHUP** (numéro 1) : envoyé par le système en cas de mort du processus “leader” de la session. Rappelons qu’une nouvelle session est créée à la connexion de l’utilisateur ou lorsque l’on ouvre un nouveau terminal. Ainsi, lorsque l’on ferme un terminal, tous les processus envoyés à partir de ce terminal (et tous leurs descendants) reçoivent le signal **SIGHUP**.
- **SIGINT** (numéro 2) : envoyé par le système à tous les processus de la session courante lors de la frappe du caractère de contrôle associé à `intr`. En général, ce caractère est `[ctrl C]` (cf la commande Shell `stty` pour changer ce caractère).
- **SIGQUIT** (numéro 3) : idem **SIGINT**, mais ce signal est envoyé lors de la frappe du caractère de contrôle associé à `quit` qui est, en général, `[ctrl \]`.
- **SIGFPE** (numéro 8) : envoyé par l’unité centrale à un processus en cas d’erreur arithmétique (division par zéro...)
- **SIGKILL** (numéro 9) : envoyé par la commande `kill`, ou la primitive système `kill`. On verra bientôt que ce signal est le seul qui ne peut être intercepté. Il cause donc toujours la mort du processus qui le reçoit.
- **SIGUSR1** et **SIGUSR2** (numéros 10 et 12) : laissés disponibles pour les utilisateurs, afin qu’ils puissent créer leurs propres signaux sans surcharger ceux existants.
- **SIGALRM** (numéro 14) : envoyé par le système à un processus lorsque son horloge arrive à zéro ; cette horloge peut être initialisée à l’aide de la fonction système `alarm`

```
unsigned int alarm(unsigned int s)
```

Si `s=0`, alors le signal ne sera pas envoyé ; sinon, le signal sera envoyé au bout de `s` secondes.

- **SIGTERM** (numéro 15) : envoyé par le système à tous les processus existants lors de l’appel de la commande `shutdown`.

2.2 Interceptor des signaux avec la primitive `signal`

Par défaut, un processus recevant un message termine brutalement son exécution. Cependant, le processus receveur peut mettre en œuvre une réponse prédéfinie, sauf pour le signal **SIGKILL** qui ne pourra jamais être intercepté, en utilisant la fonction système `signal` (définie dans `signal.h`) :

```
typedef void (*sig_handler_t)(int);
// sig_handler_t est le type ‘‘pointeur sur une fonction ayant
// un argument de type entier et retournant void’’.
sig_handler_t signal(int signum, sig_handler_t handler);
```

Cette fonction installe le nouveau gestionnaire `handler` pour le signal numéro `signum`, et retourne l’ancien gestionnaire. Le gestionnaire de signal `handler` peut être

- soit une fonction spécifique de l’utilisateur, qui sera exécutée à la réception du signal `signum` ; à la fin de l’exécution de cette fonction, le processus revient à l’instruction où il en était juste avant de recevoir le signal,
- soit la constante `SIG_IGN` ; dans ce cas, le signal `signum` sera ignoré,
- soit la constante `SIG_DFL` ; dans ce cas, le comportement par défaut (terminaison du processus) sera appliqué.

Exemple d’interception de signal :

```
#include <signal.h>

void traite_signal(int signum){
    printf("Je viens de recevoir le signal %d et je vais l'ignorer\n",signum);
}

int main(){
    signal(SIGINT,traite_signal);
    while(1) sleep(5);
    return 0;
}
```

2.3 Attendre un signal

Par principe, un signal est inattendu... si toutefois on souhaite suspendre l'exécution d'un processus jusqu'à ce qu'il reçoive un signal, on peut utiliser la fonction système `pause()`.

3 Les tubes

Les tubes permettent à des processus d'échanger des flux de caractères (en fait, des octets puisqu'un caractère est codé sur un octet). Il s'agit là d'un échange unidirectionnel selon le principe d'une file FIFO : il y a un processus qui écrit, et l'autre qui lit ; les caractères sont lus séquentiellement dans l'ordre où ils ont été écrits ; chaque caractère est lu une seule fois puis il est supprimé du tube. Enfin, la capacité d'un tube est limitée, et un processus écrivant dans un tube plein sera bloqué jusqu'à ce qu'un autre processus lise dans le tube et libère ainsi de la place.

Ce mécanisme de tubage est implémenté à l'aide de fichiers. Notons que les fichiers utilisés pour les tubes sont en mémoire centrale et non sur le disque.

3.1 Retour sur le système de gestion de fichiers

Pour bien comprendre le mécanisme du tubage, il est nécessaire de revenir sur la gestion des fichiers par UNIX/Linux. Pour gérer l'accès aux fichiers, le système maintient une "table des fichiers ouverts", une "table des i-nœuds" et, pour chaque processus, une "table des descripteurs de fichiers du processus" :

1. La table des descripteurs de fichiers d'un processus contient une entrée pour chaque fichier ouvert par ce processus. A la création du processus, cette table contient généralement trois entrées : l'entrée 0 pour le fichier correspondant aux données lues sur le clavier (`/dev/stdin`), l'entrée 1 pour le fichier correspondant aux données écrites sur le terminal (`/dev/stdout`) et l'entrée 2 pour le fichier correspondant aux messages d'erreur écrits sur le terminal (`/dev/stderr`).
2. La table des fichiers ouverts contient une entrée pour chaque fichier ouvert (notons que si un même fichier physique est ouvert par plusieurs processus différents en même temps, alors la table des fichiers ouverts contiendra une entrée différente pour chacune de ces ouvertures). Cette table des fichiers ouverts contient notamment le mode d'ouverture du fichier (lecture, écriture ou lecture/écriture), la position courante du pointeur de position dans le fichier, et le numéro identifiant du fichier physique (appelé `inode`).
3. La table des i-nœuds (`i-nodes`) contient une entrée pour chaque fichier physique différent.

A chaque fois qu'un processus ouvre un nouveau fichier, le système crée une nouvelle entrée dans la table des descripteurs du processus, ainsi qu'une nouvelle entrée dans la table des fichiers ouverts, et fait pointer la nouvelle entrée de la table des descripteurs sur la nouvelle entrée de la table des fichiers ouverts, qui pointe elle-même sur la table des i-nodes... Cette organisation à trois niveaux permet au système de gérer le fait que plusieurs processus différents peuvent ouvrir un même fichier physique.

En général, chaque ligne de la table des fichiers ouverts est pointée par un descripteur de fichier d'un seul processus (celui qui a ouvert le fichier). Cependant, lorsque l'on crée un nouveau processus avec la fonction `fork`, le processus fils hérite du PCB de son père, et donc de sa table des descripteurs de fichiers. De façon plus générale, tous les descendants d'un processus ayant ouvert un fichier héritent (potentiellement) du descripteur de fichier correspondant. Ainsi, il peut y avoir plusieurs descripteurs de fichiers, appartenant à des processus différents, qui pointent sur une même entrée de la table des fichiers ouverts. La table des fichiers ouverts contient, pour chacune de ses entrées, le nombre de descripteurs de fichiers pointant sur elle.

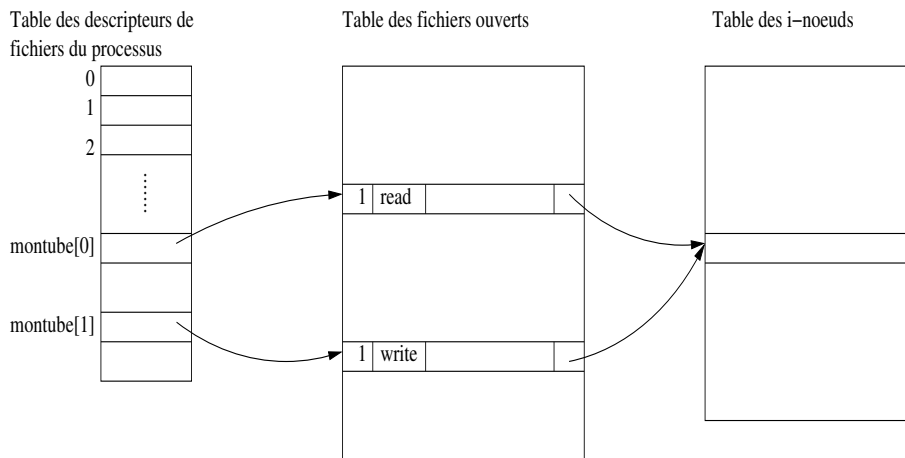
3.2 Création d'un tube

Un tube est un fichier (stocké en mémoire centrale) et est identifié par son *i-node* comme tous les autres fichiers. Pour créer un tube, on utilise la primitive système

```
int pipe(int montube[2]);
```

Cette primitive retourne -1 en cas d'erreur (par exemple, s'il y a trop de descripteurs de fichiers ouverts...). Sinon, elle crée un nouveau fichier (stocké en mémoire centrale) et ajoute une nouvelle entrée pour ce fichier dans la table des *i-nœuds*. Elle crée également deux nouvelles entrées dans la table des descripteurs de fichiers du processus, et deux nouvelles entrées dans la table des fichiers ouverts : `montube[0]` contient le descripteur de fichiers utilisé pour lire des données dans le tube ; `montube[1]` contient celui utilisé pour écrire des données dans le tube.

Ainsi, après l'exécution de `pipe(montube)`, on se retrouve dans la configuration suivante :



3.3 Fermeture d'une extrémité d'un tube

Pour fermer l'extrémité d'un tube, on utilise la même primitive système que pour fermer n'importe quel fichier, *i.e.*, `int close(int fd)` qui ferme le descripteur `fd`, de manière à ce qu'il ne référence plus aucun fichier, et ne puisse être réutilisé dans ce processus. Si `fd` est la dernière copie d'un descripteur de fichier donné, la ligne correspondante dans la table des fichiers ouverts est détruite.

La fonction `close` retourne -1 en cas d'erreur (par exemple, si `fd` n'est pas un descripteur de fichier...).

3.4 Lecture et Ecriture dans un tube

On écrit et on lit des caractères dans un tube en utilisant les mêmes primitives que pour écrire et lire dans des fichiers. La fonction `ssize_t read(int fd, void *buf, size_t count)` lit jusqu'à `count` caractères depuis le descripteur de fichier `fd` dans le buffer pointé par `buf`. Cette fonction renvoie -1 si elle échoue. Sinon, elle renvoie le nombre de caractères lus. Ce nombre peut-être inférieur à `count` si le tube contient moins de `count` caractères. Si le tube est vide, alors s'il n'y a pas d'écrivain dans ce tube, `read` retourne 0 (la fin de fichier est détectée); sinon le processus est bloqué sur le `read` jusqu'à ce qu'un processus écrive dans le tube.

La fonction `ssize_t write(int fd, const void *buf, size_t count)` écrit jusqu'à `count` caractères dans le fichier associé au descripteur de fichier `fd` depuis le buffer pointé par `buf`. S'il n'y a plus de lecteur pour ce tube, alors le signal `SIGPIPE` est envoyé au processus ayant appelé la fonction `write`, ce qui entraîne sa mort (sauf s'il s'est protégé contre ce signal...). S'il y a des lecteurs pour ce tube, mais s'il n'y a plus assez de place dans le tube pour écrire `count` caractères, alors le processus s'endort jusqu'à ce que le tube se vide.

Un premier exemple vraiment simple :

```
int main(void){
    int fin;
    int tube[2];
    char c;
    if (pipe(tube)==-1){ perror("pipe"); exit(1); }
    write(tube[1],"123456",6);
    close(tube[1]);
    fin = read(tube[0],&c,1);
    while (fin > 0){
        printf("J'ai lu : %c\n",c);
        fin = read(tube[0],&c,1);
    }
    exit(0);
}
```

En général, ce sont deux processus différents qui vont lire et écrire. Ces deux processus devront nécessairement être des descendants du processus qui a appelé `pipe`. On aura alors quelque chose qui ressemble à ça :

```
int main(void){
    pid_t pid_fils1, pid_fils2;
    int tube[2];
    int etat,i,nbchar;

    if (pipe(tube)==-1){ perror("pipe"); exit(1); }
    pid_fils1=fork();
    if (pid_fils1==0){ // fils écrivain
        close(tube[0]);
        write(tube[1],"123456",6);
        //close(tube[1]); pas indispensable car fermé par le exit...
        exit(0);
    }
    else {
        pid_fils2=fork();
        if (pid_fils2 == 0){ // fils lecteur
            char c;
            int fin;
            close(tube[1]);
            fin = read(tube[0],&c,1);
            while (fin > 0){
                printf("J'ai lu : %c\n",c);
                fin = read(tube[0],&c,1);
            }
            exit(0);
        }
        else { // père
            close(tube[0]);
            close(tube[1]); // INDISPENSABLE sinon l'écriture ne se termine jamais...
            pid_fils1 = waitpid(pid_fils1,&etat,0);
            pid_fils2 = waitpid(pid_fils2,&etat,0);
            exit(0);
        }
    }
}
```

Duplication d'un descripteur de fichier. La primitive système

```
int dup2(int oldfd, int newfd);
```

crée une copie du descripteur de fichier `oldfd` dans le descripteur de fichier `newfd`. Si `newfd` correspondait auparavant à un descripteur de fichier ouvert, alors ce descripteur de fichier sera fermé avant la copie.

Cette primitive sera utilisée pour rediriger les entrées/sorties :

```
int main(void){
    pid_t pid_fils1, pid_fils2;
    int tube[2];
    int etat,i,nbchar;

    if (pipe(tube)==-1){ perror("pipe"); exit(1); }
    pid_fils1=fork();
    if (pid_fils1==0){ // fils écrivain
        close(tube[0]);
        dup2(tube[1],1);
        close(tube[1]);
        printf("123456");
        exit(0);
    }
    else {
        pid_fils2=fork();
        if (pid_fils2 == 0){ // fils lecteur
            char c;
            close(tube[1]);
            dup2(tube[0],0);
            close(tube[0]);
            c = getchar();
            while (c != EOF){
                printf("J'ai lu : %c\n",c);
                c = getchar();
            }
            exit(0);
        }
        else { // père
            close(tube[0]);
            close(tube[1]);
            pid_fils1 = waitpid(pid_fils1,&etat,0);
            pid_fils2 = waitpid(pid_fils2,&etat,0);
            exit(0);
        }
    }
}
```

4 Les IPC

Les IPC (Inter Process Communication) sont un ensemble de primitives système permettant à différents processus de communiquer. On distingue trois différents types de communication inter-processus avec les IPC :

- les sémaphores, qui permettent à plusieurs processus de se synchroniser et de communiquer des informations sur l'utilisation de ressources,
- les segments de mémoire partagée, qui permettent à plusieurs processus de partager un segment de mémoire, dans lequel il peuvent lire ou écrire des données,
- les messages, qui permettent à des processus de s'envoyer des messages, avec une gestion des files d'attente de messages.

4.1 Identification des IPC par clés

Pour tous les IPC, il existe un mécanisme d'adressage par clé qui permet à différents processus de récupérer l'identifiant d'un IPC partagé sans avoir à communiquer. Il suffit que tous les processus souhaitant partager un même IPC choisissent un fichier et une lettre ; à partir de ce fichier et de cette lettre, le système crée une clé unique qui permettra de retrouver le numéro identifiant de l'IPC.

Création d'une clé. On utilise la primitive `ftok()` qui se trouve dans le fichier `sys/ipc.h`

```
key_t ftok (char * pathname, int proj) ;
```

Cette fonction utilise le numéro identifiant (i-node) du fichier (ou répertoire) indiqué par `pathname` (qui doit exister et être accessible), et les huit bits de poids faible de `proj` (qui doit donc être un caractère non nul) pour créer la clé d'un IPC. Cette clé, de type `key_t` (type défini dans `sys/types.h`) sera ensuite utilisable dans les primitives `msgget`, `semget`, ou `shmget` pour récupérer le numéro identifiant de l'IPC.

4.2 Listage et destruction d'IPC

La commande Shell `ipcs` permet de lister l'ensemble des IPC en cours ; la commande `ipcrm` permet de détruire un IPC (sous réserve que plus aucun processus ne l'utilise ; s'il y a encore des processus qui utilisent l'IPC, il sera marqué comme étant "à détruire", et le système attendra que tous les processus aient fini de l'utiliser pour le détruire effectivement).

4.3 Les segments de mémoire partagée

Toutes les primitives de gestion des segments de mémoire partagée sont définies dans le fichier `sys/shm.h` (`shm` pour "shared memory") ; certains types sont définis dans les fichiers `sys/types.h` et `sys/ipc.h`. Il faudra donc faire un `include` de ces trois fichiers.

Création et récupération d'un segment de mémoire partagée. La création d'un segment de mémoire partagée se fait par un seul processus ; les autres processus récupéreront alors l'identificateur du segment créé. Dans les deux cas (création ou récupération), on utilise la primitive système `shmget` :

```
int shmget(key_t clé, int size, int shmflg) ;
```

Le premier paramètre est la clé attachée au segment de mémoire partagé (ou bien `IPC_PRIVATE` si l'on ne veut pas associer une clé particulière au segment). Le deuxième paramètre est le nombre d'octets du segment (ce nombre peut-être connu à l'aide de la fonction `sizeof`). Le troisième paramètre est une liste de "flags" séparés par des '|'. De façon pratique, `shmflg` sera égal à 0 lors de la récupération d'un segment existant, tandis qu'il sera égal à `IPC_CREAT|IPC_EXCL|mode` lors de la création d'un nouveau segment (`IPC_EXCL` est utilisé avec `IPC_CREAT` pour garantir l'échec si le segment existe déjà). Dans ce cas, `mode` sera un nombre octal composé de 3 chiffres compris entre 0 et 7 (les 9 bits de poids faibles de ce nombre sont significatifs) indiquant les permissions pour le propriétaire, le groupe et les autres. Pour indiquer que le nombre est en octal, il faudra le faire précéder de 0 (et on aura donc 4 chiffres). Actuellement la permission d'exécution n'est pas utilisée par le système, et donc 7 équivaut à 6.

Le comportement de `shmget` peut être résumé de la façon suivante :

- Si la clé n'est pas valide, alors `shmget()` retourne -1.
- Sinon
 - Si `IPC_CREAT` et `IPC_EXCL` et `mode` sont présents dans `shmflg` alors
 - si aucun IPC n'est déjà associé à `clé`, alors un nouveau segment mémoire, de taille `size` (arrondie au multiple supérieur de `PAGE_SIZE`), est créé et la primitive renvoie l'identificateur du segment de mémoire partagée associé à la valeur de l'argument `clé`.
 - Sinon la primitive retourne -1.

- Sinon (si `shmflg==0`)
 - Si le processus appelant a les bons droits par rapport à ceux qui ont été donnés lors de la création du segment, et si le segment existe, alors la primitive renvoie l'identificateur du segment de mémoire partagée associé à la valeur de l'argument clé.
 - Sinon la primitive retourne -1.

Attachement d'un segment à un processus. La création alloue l'emplacement mémoire nécessaire, et retourne un numéro identifiant de cet emplacement mémoire. Ce numéro peut être récupéré par les autres processus. Pour que les processus puissent effectivement utiliser le segment de mémoire partagée, il faut qu'ils se l'attachent, c'est-à-dire qu'ils mettent dans leur zone de mémoire (dans leur tas) un pointeur sur le segment de mémoire partagée. On utilise pour cela la primitive

```
void *shmat ( int shmId, const void * shmaddr, int shmflg );
```

qui attache le segment de mémoire partagée identifié par `shmId` au segment de données du processus appelant. `shmaddr` permet de préciser l'adresse physique à laquelle on veut réaliser l'attachement... en pratique, on n'utilisera pas cette possibilité, et on mettra `shmaddr` à `NULL` pour indiquer au système qu'il doit se charger de trouver cette adresse.

`shmflg` précise ce que le processus va faire dans ce segment de mémoire : il peut prendre pour valeur `SHM_RDONLY` si le processus ne fait que lire ; `SHM_W` s'il ne fait qu'écrire et `SHM_R|SHM_W` s'il lit et écrit.

La primitive `shmat` retourne l'adresse du premier octet du segment partagé. Cette adresse doit être stockée dans un objet du même type que celui utilisé lors de la création du segment... Pour faire les choses proprement, il faudra convertir la valeur retournée par `shmat` (qui est un `void*`), en un objet de type "pointeur sur T", où T est le type du segment de mémoire partagée.

Le processus pourra alors utiliser le segment partagé, en passant par ce pointeur, comme s'il avait créé dans sa propre mémoire un objet de ce type.

Détachement d'un segment de mémoire partagée pour un processus. Quand un processus a fini de travailler sur un segment de mémoire partagée, il l'indique au système en utilisant la primitive

```
int shmdt (const void * shmaddr);
```

Cette fonction détache le segment de mémoire partagée situé à l'adresse indiquée par `shmaddr`. Le segment doit être effectivement attaché, et l'adresse `shmaddr` doit être celle renvoyée précédemment par `shmat`. Concrètement, cela revient à supprimer le lien entre le tas du processus et la zone de mémoire partagée.

Notons que si votre processus "oublie" de détacher ses segments de mémoire partagée, le détachement sera fait automatiquement par le système à la mort du processus.

Contrôle d'un segment de mémoire partagée. Pour obtenir des informations sur un segment de mémoire partagée, ou pour le détruire, on utilise la primitive

```
int shmctl(int shmId, int cmd, struct shmId_ds *info);
```

Cette primitive permet à l'utilisateur d'obtenir des informations concernant un segment de mémoire partagée, ainsi que de changer le propriétaire, ou le groupe, ou les permissions d'accès à ce segment. Cette fonction permet également de détruire un segment.

Le paramètre `cmd` peut prendre une des trois valeurs suivantes : `IPC_STAT`, `IPC_SET` ou `IPC_RMID`.

- Si `cmd = IPC_STAT`, alors on récupère dans `*info` les informations concernant le segment identifié par `shmId`, où `*info` doit être une structure du type `shmId_ds` suivant :

```
struct shmId_ds {
    struct ipc_perm shm_perm; /* Permissions d'accès      */
    int shm_segsz;          /* Taille segment en octets */
    time_t shm_atime;      /* Heure dernier attachement */
};
```

```

    time_t shm_dtime;          /* Heure dernier détachement */
    time_t shm_ctime;         /* Heure dernier changement */
    unsigned short shm_cpid;  /* PID du créateur */
    unsigned short shm_lpid;  /* PID du dernier opérateur */
    short shm_nattch;        /* Nombre d'attachements */
}

```

Le champ `shm_perm` a la forme suivante :

```

struct ipc_perm {
    key_t key;
    ushort uid; /* UID et GID effectifs du propriétaire */
    ushort gid;
    ushort cuid; /* UID et GID effectif du créateur */
    ushort cgid;
    ushort mode; /* Mode d'accès sur 9 bits de poids faible */
    ushort seq; /* numéro de séquence */
};

```

- Si `cmd = IPC_SET`, alors les changements que l'utilisateur a apportés dans les champs `uid`, `gid`, ou `mode` de la structure `shm_perms` seront appliquées.
- Si `cmd = IPC_RMID`, alors le segment partagé sera marqué comme étant "prêt pour la destruction". Il sera détruit effectivement après le dernier détachement (quand le membre `shm_nattch` de la structure `shmid_ds` associée vaudra zéro). L'appelant doit être le créateur du segment, son propriétaire, ou le Super-utilisateur.

Exemple d'utilisation d'un segment de mémoire partagée Il faut définir quelque part (dans un fichier avec l'extension `.h`) le type du segment de mémoire partagée. Par exemple,

```

typedef struct {
    int nb;
    int t[256];
} Tdonnees;

```

Code pour créer le segment :

```

int main(void){
    key_t cle;
    int id;
    Tdonnees *commun;

    /* Génération de la clé d'accès */
    cle = ftok(getenv("HOME"), 'A');
    if (cle == -1) { perror("ftok"); exit(EXIT_FAILURE);}
    /* Création du segment */
    id = shmget(cle, sizeof(Tdonnees), IPC_CREAT | IPC_EXCL | 0666);
    if (id == -1) {perror("Création shm"); exit(EXIT_FAILURE);}
    /* Attachement en écriture du segment à 'commun' */
    commun = (Tdonnees *) shmat(id, NULL, SHM_W);
    if (commun == NULL) { perror("Attachement shm"); exit(EXIT_FAILURE);}
    /* Initialisation des données du segment (facultatif) */
    commun->nb = 0;
    /* Détachement du segment */
    if (shmdt(commun) == -1) {perror("Détachement shm");exit(EXIT_FAILURE);}
    exit(EXIT_SUCCESS);
}

```

Code pour accéder en lecture écriture au segment :

```

int main(void){

```

```

key_t cle;
int id;
Tdonnees *commun;

/* Génération de la clé d'accès */
cle = ftok(getenv("HOME"), 'A');
if (cle == -1) { perror("ftok"); exit(EXIT_FAILURE);}
/* Récupération de l'adresse du segment */
id = shmget(cle, sizeof(Tdonnees), 0);
if (id == -1) {perror("Récupération shm");exit(EXIT_FAILURE);}
/* Attachement en lecture et écriture du segment à 'commun' */
commun = (Tdonnees *) shmat(id, NULL, SHM_R | SHM_W);
if (commun == NULL) { perror("Attachement shm"); exit(EXIT_FAILURE);}
/* Utilisation : saisie d'entiers et ajout dans t à partir de l'indice nb */
while(1) {
    printf("+ ");
    if (scanf("%d", &(commun->t[commun->nb])) != 1) break;
    commun->nb++;
};
/* Détachement du segment */
if (shmdt(commun) == -1) {perror("Détachement shm");exit(EXIT_FAILURE);}
exit(EXIT_SUCCESS);
}

```

Code pour accéder en lecture au segment :

```

int main(void){
    key_t cle;
    int id, i, total;
    Tdonnees *commun;

    /* Génération de la clé d'accès de SHM */
    cle = ftok(getenv("HOME"), 'A');
    if (cle == -1) {perror("ftok shm");exit(EXIT_FAILURE);}
    /* Récupération de l'adresse du segment */
    id = shmget(cle, sizeof(Tdonnees), 0);
    if (id == -1) {perror("Récupération shm");exit(EXIT_FAILURE);}
    /* Attachement en lecture du segment à 'commun' */
    commun = (Tdonnees *) shmat(id, NULL, SHM_R);
    if (commun == NULL) {perror("Attachement du shm");exit(EXIT_FAILURE);}
    /* Utilisation du segment en lecture */
    printf("Il y a %d valeurs dans t : ",commun->nb);
    total = 0;
    for (i=0; i<commun->nb; i++){
        printf("%d ",commun->t[i]);
        total += commun->t[i];
    }
    printf("\nTotal = %d\n",total);
    /* Détachement du segment (pas de suppression) */
    if (shmdt((char *) commun) == -1) {perror("Détachement shm");exit(EXIT_FAILURE);}
    exit(EXIT_SUCCESS);
}

```

Code pour détruire le segment :

```

int main(void){
    key_t cle;
    int id;

```

```

/* Génération de la clé d'accès */
cle = ftok(getenv("HOME"), 'A');
if (cle == -1) { perror("ftok"); exit(EXIT_FAILURE);}
/* Récupération de l'identifiant du segment */
id = shmget(cle, sizeof(Tdonnees), 0);
if (id == -1) {perror("Récupération shm"); exit(EXIT_FAILURE);}
/* Suppression segment */
if (shmctl(id, IPC_RMID, NULL) == -1) {perror("Suppression shm");exit(EXIT_FAILURE)};
exit(EXIT_SUCCESS);
}

```

Remarque : le segment ne sera effectivement détruit que lorsque plus aucun processus n'aura ce segment attaché à sa mémoire...

4.4 Les sémaphores

Les IPC fournissent un ensemble de primitives pour manipuler des sémaphores. Ces sémaphores permettent de gérer l'accès concurrent des processus aux ressources et plus généralement de synchroniser l'exécution de processus. Un sémaphore est constitué des deux composants suivants :

- un compteur, qui donne la quantité de ressource disponible quand il est positif, et la quantité de ressource demandée quand il est négatif,
- une file d'attente, qui contient l'ensemble des processus en attente de la ressource.

Ce compteur et cette file d'attente sont stockés dans une zone mémoire partagée entre plusieurs processus. La différence avec les segments de mémoire partagée est que les primitives de gestion des sémaphores sont atomiques (autrement dit, un processus ne sera jamais interrompu lorsqu'il exécutera une de ces primitives).

Les sémaphores permettent notamment d'implémenter des exclusions mutuelles et de synchroniser des processus.

Utilisation de sémaphores pour implémenter une exclusion mutuelle. Lorsque plusieurs processus différents veulent accéder à une même ressource (par exemple un segment de mémoire partagée), il faut placer les suites d'instructions testant et modifiant cette ressource dans une "section critique", c'est-à-dire une portion de code qui ne sera pas interrompue par un autre processus voulant accéder à la même ressource : c'est le principe de "l'exclusion mutuelle".

Le principe général pour gérer une exclusion mutuelle entre plusieurs processus à l'aide de sémaphores est le suivant :

- Création d'un sémaphore partagé par tous les processus voulant accéder à la ressource critique ; initialisation du compteur du sémaphore à 1 et de sa file d'attente à vide.
- A chaque fois qu'un processus veut entrer en section critique (pour utiliser la ressource critique), il appelle la procédure `wait` ou `P` sur ce sémaphore. Cette procédure est atomique et effectue le traitement suivant :
 - si le compteur du sémaphore est inférieur ou égal à zéro, alors le compteur est décrémenté de un, le processus est ajouté dans la file d'attente, et son exécution est suspendue (sauf si le processus a fait un appel non bloquant) ;
 - sinon (le compteur est égal à 1, ce qui signifie qu'aucun processus n'utilise la ressource critique), le compteur est mis à zéro et le processus continue son exécution (et donc utilise la ressource critique).
- A chaque fois qu'un processus a fini d'utiliser la ressource critique (il sort de la section critique), il appelle la procédure `signal` ou `V` sur le sémaphore. Cette procédure est atomique et effectue le traitement suivant :
 - si le compteur du sémaphore est inférieur à zéro, alors le compteur est incrémenté de un, et un processus de la file d'attente est ré-activé.
 - sinon (le compteur est égal à zéro), le compteur est mis à un (plus personne ne demande la ressource critique).

Utilisation de sémaphores pour synchroniser des processus. Les sémaphores permettent de mettre en attente un processus jusqu'à ce qu'un autre processus incrémente le compteur du sémaphore. Ainsi, les sémaphores peuvent être utilisés pour "synchroniser" l'activité de processus. Lorsqu'un processus A doit "attendre" un processus B pour effectuer une instruction donnée I (par exemple, le processus A attend que le processus B est modifié la valeur d'une variable du segment de mémoire partagée pour effectuer une opération particulière), on utilise les sémaphores de la façon suivante :

- Création d'un sémaphore partagé par A et B ; initialisation du compteur à 0 et de la file d'attente à vide.
- Le processus A, avant d'exécuter l'instruction I, appelle la procédure `wait` ou `P` sur le sémaphore. Cette procédure va bloquer A jusqu'à ce qu'un autre processus incrémente le compteur du sémaphore...
- Le processus B appelle la procédure `signal` ou `V` sur le sémaphore au moment où il veut que le processus A exécute l'instruction I.

Primitive C pour la création/récupération d'un ensemble de sémaphores. La création d'un sémaphore se fait par un seul processus ; les autres processus récupéreront alors l'identificateur du sémaphore créé. Dans les deux cas (création ou récupération), on utilise la primitive système `semget` :

```
int semget(key_t clé, int nb_sem, int semflg) ;
```

Cette primitive crée un ensemble de `nb_sem` sémaphores, et retourne le numéro identifiant de cet ensemble. Le troisième argument `semflg` contient les options. Ces options sont les mêmes que celles de `shmget` (`IPC_CREAT|IPC_EXCL|mode` pour créer un nouveau sémaphore, et 0 pour récupérer l'identificateur d'un sémaphore existant). Comme pour les segments de mémoire partagée, si `clé = IPC_PRIVATE` alors une nouvelle clé est créée pour ce sémaphore.

Contrôle d'un sémaphore. La primitive

```
int semctl (int semid, int semno, int cmd) ;  
int semctl (int semid, int semno, int cmd, union semun arg) ;
```

initialise, détruit ou accède au sémaphore numéro `semno` (numérotation à partir de 0) dans l'ensemble de sémaphores identifié par `semid`. Le troisième argument, `cmd`, peut prendre différentes valeurs, en fonction de la commande que l'on souhaite faire ; dans certains cas, la commande est suivie d'un argument. On utilisera la commande `SETVAL` suivie d'un entier positif pour affecter cet entier au compteur du sémaphore. On utilisera la commande `GETVAL` pour récupérer cette valeur. On utilisera la commande `IPC_RMID` pour détruire le sémaphore.

En cas d'échec, l'appel-système renvoie -1. Autrement, l'appel-système renvoie une valeur non-négative (dépendant de l'argument `cmd...`).

Opérations sur les sémaphores. On utilise la primitive `semop`

```
int semop (int semid, struct sembuf *tab, unsigned nb) ;
```

Cette fonction effectue des opérations sur des sémaphores de l'ensemble de sémaphores identifié par `semid`. Le paramètre `nb` contient le nombre d'opérations que l'on souhaite effectuer. Le paramètre `tab` est un tableau comportant un élément de type `struct sembuf` pour chacune de ces opérations (si l'on souhaite faire une seule opération, alors `nb=1` et le tableau aura un seul élément). Les éléments du tableau `tab` sont du type structure `sembuf`, contenant les champs suivants :

- le champ `short sem_num` donne le numéro du sémaphore (compris entre 0 et `nb_sem-1`) sur lequel l'opération doit être effectuée,
- le champ `short sem_op` décrit l'opération à effectuer :
 - si `sem_op>0`, alors le compteur du sémaphore est incrémenté de `sem_op` ; s'il y a des processus en attente qui demandent une quantité de ressource inférieure ou égale à la valeur du compteur, alors un de ces processus est réveillé,
 - si `sem_op = 0` alors, si le compteur a une valeur non nulle, le processus est endormi jusqu'à ce que le compteur du sémaphore prenne la valeur 0,

- si `sem_op < 0` alors le compteur est décrémenté de la valeur absolue de `sem_op`; puis, si le compteur a une valeur négative, alors le processus est endormi jusqu'à ce que le compteur du sémaphore devienne supérieur ou égal à la valeur absolue de `sem_op`. Cependant, s'il y a plusieurs processus endormis en attente d'une incrémentation du compteur de ce sémaphore, alors on ne sait pas lequel sera réveillé...

De façon pratique, si l'on veut implémenter la procédure P, on positionne `sem_op` à -1, tandis que si l'on veut implémenter la procédure V, on positionne `sem_op` à 1.

- le champ `sem_flg` permet d'activer des options. Si ce champ est égal à 0, alors aucune option n'est activée. Si ce champ a pour valeur `IPC_NOWAIT`, alors l'opération est non bloquante, c'est-à-dire qu'au lieu d'endormir le processus, la valeur d'erreur `EAGAIN` est retournée.

Exemple. Exemple de code pour créer et utiliser un sémaphore d'exclusion mutuelle :

```
int main(void){
    key_t cle;
    int sem;
    struct sembuf sb[1];

    /* Génération de la clé d'accès */
    cle = ftok(getenv("HOME"), 'A');
    if (cle == -1) { perror("ftok"); exit(EXIT_FAILURE);}
    /* Crée un ensemble de _un_ sémaphore */
    sem = semget(cle, 1, IPC_CREAT | IPC_EXCL | 0666);
    if (sem < 0) {perror("creer_sem");exit(EXIT_FAILURE);}
    /* Initialisation du compteur à 1 */
    if (semctl(sem, 0, SETVAL, 1) < 0) {perror("init sem"); exit(EXIT_FAILURE)};
    ...
    ...
    /* Entrée en section critique : P */
    sb[0].sem_num = 0;
    sb[0].sem_op = -1;
    sb[0].sem_flg = 0;
    if (semop(sem, sb, 1) != 0) {perror("changer_sem");exit(EXIT_FAILURE)};
    ...
    ...
    /* Sortie de la section critique : V */
    sb[0].sem_num = 0;
    sb[0].sem_op = 1;
    sb[0].sem_flg = 0;
    if (semop(sem, sb, 1) != 0) {perror("changer_sem");exit(EXIT_FAILURE)};
    ...
    ...
    /* destruction du sémaphore */
    if (semctl(semget(cle, 1, 0), 0, IPC_RMID) != 0) {perror("detruire_sem");exit(EXIT_FAILURE)};
}

```

5 Les processus légers ou “Threads”

Sur une machine parallèle (ayant plusieurs processeurs), on peut accélérer l'exécution d'un programme en exécutant certaines parties en parallèle sur les différents processeurs. En UNIX, cela peut être fait en créant (avec un `fork`) un nouveau processus pour chaque partie à paralléliser. Dans ce cas, les processus se communiquent leurs résultats, et se synchronisent, à l'aide des IPC (segments de mémoire partagée et sémaphores). Cependant, cette façon de paralléliser les tâches s'avère assez “lourde”. En effet, chaque nouveau processus possède son propre PCB et sa propre zone de mémoire. Il s'ensuit trois problèmes :

- problème de performance à la création du processus, puisque l'allocation et la copie du PCB et de la zone mémoire sont des mécanismes coûteux,
- problème de performance à la commutation (dans l'unité centrale) d'un processus à l'autre, puisque les processus travaillent sur des zones de mémoire différentes qu'il faut recharger.
- problème de communication entre les processus, qui ont des variables séparées (... d'où la nécessité d'utiliser des segments de mémoire partagée pour mettre des données en commun).

Il existe une deuxième façon plus "légère" de paralléliser l'exécution d'un code : en utilisant les "processus légers" ou "threads". Ces processus légers partagent plus de choses que les processus traditionnels : tous les processus légers d'un même ensemble partagent le même PCB, le même code et le même tas (les mêmes variables). En revanche, chaque processus léger a son propre compteur ordinal (lui donnant l'instruction en cours d'exécution dans le code à exécuter), et sa propre pile.

5.1 Contrôle de threads

Les primitives permettant de créer, détruire et contrôler des processus légers sont définies dans le fichier `pthread.h`, qu'il faudra donc inclure avant de faire des threads. Par ailleurs, la compilation de programmes utilisant des threads se fera de la façon suivante

```
gcc -D_REENTRANT NT -o mesthreads mesthreads.c -lpthread
```

Le `-lpthread` fait une édition de lien avec la bibliothèque des threads; `-D_REENTRANT NT` est une option qui indique au compilateur que plusieurs threads différents pourront exécuter en même temps une même procédure.

Création d'un processus léger. La primitive

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

demande le lancement d'un nouveau processus léger, et retourne 0 en cas de succès. Ce processus léger s'exécutera de façon concurrente avec le processus appelant : si la machine a plusieurs processeurs, alors chaque processus léger pourra s'exécuter sur un processeur différent; sinon le parallélisme sera simulé...

Les paramètres de cette primitive sont :

- `*thread`, qui contient l'identifiant du nouveau processus léger. Cet identifiant peut être connu en appelant la fonction `pthread_self()`.
- `*attr`, qui contient les attributs du nouveau processus léger (en particulier sa priorité). Ce paramètre doit être créé et initialisé avec les primitives `pthread_attr_init` et `pthread_attr_set`. Si `*ATTR = NULL`, alors les attributs par défaut sont utilisés.
- `*start_routine` qui contient le nom de la procédure qui sera exécutée par ce processus léger. Cette procédure doit être définie dans le code du processus appelant, et doit prendre un seul paramètre de type `void *`.
- `*arg`, qui contient le paramètre (unique) à passer à `start_routine`. Ce paramètre doit être de type `void *`. Ainsi, si l'on veut passer à la routine `start_routine` un paramètre `x`, d'un type `T` quelconque, il faudra le "caster" en `void *` dans l'appel à `pthread_create`; la procédure `start_routine` recastera dans l'autre sens ce paramètre avant de l'utiliser.

Exemple 1 : Création de 2 threads (un qui affiche "Hello", et l'autre qui affiche "World").

```
void* ma_proc( void *ptr ){
    char *message;
    message = (char *) ptr; // on caste le paramètre dans le type qui va bien...
    printf("%s ", message);
}

main(){
    pthread_t thread1, thread2; // identifiants des threads
    char *m1 = "Hello";
```

```

    char *m2 = "World";
    pthread_create( &thread1, NULL, ma_proc, (void*) m1);
    pthread_create( &thread2, NULL, ma_proc, (void*) m2);
    exit(0);
}

```

Les deux threads sont exécutés de façon “concurrente”. Un premier problème est que l’on n’a aucune certitude sur l’ordre dans lequel ces deux threads s’exécutent, de sorte que "World" peut très bien s’afficher avant "Hello". On verra d’ici peu comment résoudre les problèmes de synchronisation entre processus à l’aide des sémaphores et variables de condition.

Le deuxième problème, probablement plus grave, est que lorsque le processus appelant exécute son instruction `exit(0)`, les threads qu’il a créés sont détruits, même s’ils n’ont pas fini de s’exécuter (ce qui est fort probable). Il faut donc que le processus créant les threads attende la fin de l’exécution des threads avant de mourir.

Attente de la fin d’exécution d’un processus léger. La primitive

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

suspend l’exécution du processus jusqu’à ce que le thread `th` ait terminé son exécution (c’est l’équivalent du `wait` entre processus).

Par défaut, un thread termine son exécution lorsqu’il a fini d’exécuter la procédure appelée. Si le thread veut communiquer une valeur de retour au processus qui l’a appelé, il doit explicitement appeler la primitive

```
void pthread_exit(void *retval);
```

qui termine son exécution et retourne la valeur `retval`. Dans ce cas, la valeur retournée est récupérée dans le deuxième paramètre de `pthread_join`. Il faudra bien évidemment “caster” les types (du type effectif vers `void *` dans `pthread_exit`, puis de `void *` vers le type effectif dans `pthread_join`).

Si au contraire un thread sait qu’il ne sera pas attendu par un `join`, alors il peut libérer ses ressources à la fin de son exécution en appelant la fonction `pthread_detach(pthread_t th)`.

Exemple 2 : Calcul de la somme des éléments d’un tableau. Chaque thread calcule la somme pour une portion du tableau, et transmet le sous-total au processus appelant. Le processus appelant récupère les sous-totaux et fait le total. Dans cette première version, on récupère les sous-totaux à l’aide de la primitive `pthread_exit`.

```

typedef struct {
    int nb;
    int *t;
} Tdonnees;

void* somme(void* donnees){
    Tdonnees *d;
    int i, total = 0;
    d = (Tdonnees*)donnees;
    for (i=0; i<d->nb; i++) total += (d->t)[i];
    pthread_exit((void *)&total);
}

main(){
    pthread_t thread1, thread2, thread3;
    int tab[15] = {8,3,9,5,7,12,7,45,9,3,45,12,6,6,9};
    Tdonnees d1, d2, d3;
    int total;
    void *v1, *v2, *v3;
}

```

```

d1.nb = 5; d1.t = &(tab[0]);
d2.nb = 5; d2.t = &(tab[5]);
d3.nb = 5; d3.t = &(tab[10]);
pthread_create( &thread1, NULL, somme, (void*) &d1);
pthread_create( &thread2, NULL, somme, (void*) &d2);
pthread_create( &thread3, NULL, somme, (void*) &d3);
pthread_join(thread1,&v1);
pthread_join(thread2,&v2);
pthread_join(thread3,&v3);
total = *((int*)v1) + *((int*)v2) + *((int*)v3);
printf("Le total est %d\n",total);
}

```

Une autre façon de transmettre des résultats au processus appelant est d'utiliser le paramètre de la procédure exécutée dans le thread de la façon suivante :

```

typedef struct {
    int nb;
    int *t;
    int total;
} Tdonnees;

void* somme(void* donnees){
    Tdonnees *d;
    int i, total = 0;
    d = (Tdonnees*)donnees;
    for (i=0; i<d->nb; i++) total += (d->t)[i];
    d->total = total
}

main(){
    pthread_t thread1, thread2, thread3;
    int tab[15] = {8,3,9,5,7,12,7,45,9,3,45,12,6,6,9};
    Tdonnees d1, d2, d3;
    int total;
    d1.nb = 5; d1.t = &(tab[0]);
    d2.nb = 5; d2.t = &(tab[5]);
    d3.nb = 5; d3.t = &(tab[10]);
    pthread_create( &thread1, NULL, somme, (void*) &d1);
    pthread_create( &thread2, NULL, somme, (void*) &d2);
    pthread_create( &thread3, NULL, somme, (void*) &d3);
    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);
    pthread_join(thread3,NULL);
    total = d1.total + d2.total + d3.total;
    printf("Le total est %d\n",total);
}

```

Enfin, une dernière façon de transmettre des résultats au processus appelant est d'utiliser des variables globales, déclarées en amont de la procédure exécutée par le thread. Il faudra faire attention dans ce cas qu'il n'y ait pas plusieurs threads différents qui accèdent en même temps à une même variable...

5.2 Verrous d'exclusion mutuelle (mutex)

Un mutex est un objet d'exclusion mutuelle (MUTual EXclusion device), très pratique pour protéger des données partagées par plusieurs threads. Un mutex peut être dans deux états :

déverrouillé ou verrouillé. Un mutex ne peut être verrouillé que par un seul thread à la fois. Un thread qui tente de verrouiller un mutex déjà verrouillé est suspendu jusqu'à ce que le mutex soit déverrouillé.

La primitive

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

initialise le sémaphore d'exclusion mutuelle `*mutex` selon les attributs spécifiés par `*mutexattr`. Si `mutexattr = NULL`, les paramètres par défaut sont utilisés (le comportement par défaut est `PTHREAD_MUTEX_FAST_NP` qui bloque un processus tentant de verrouiller un mutex déjà verrouillé jusqu'à ce que le mutex soit libéré)¹.

Le verrouillage d'un mutex se fait à l'aide de la primitive

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

Si `mutex` a déjà été verrouillé par un autre thread, alors cette procédure met le thread en attente, sinon `mutex` est verrouillé (et le processus continue son exécution).

Le déverrouillage se fait à l'aide de la primitive

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

qui a le comportement suivant :

- Si `mutex` est déjà déverrouillé, alors cette primitive ne fait rien.
- Sinon
 - S'il n'y a plus de thread en attente, alors `mutex` passe dans l'état déverrouillé
 - Sinon, un des threads en attente reprend son exécution, et `mutex` reste dans l'état verrouillé.

La destruction d'un mutex se fait à l'aide de la primitive

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Enfin, la primitive

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

retourne `EBUSY` si le mutex `*mutex` est déjà verrouillé; sinon le mutex est verrouillé par le thread appelant. Cette primitive permet donc de faire des "verrouillages non bloquants".

Exemple de threads avec exclusion mutuelle : Calcul du plus petit élément d'un tableau. Chaque thread calcule le plus petit élément d'une portion du tableau, et met à jour (si nécessaire) la variable globale `min`. La mise à jour de cette variable est faite dans une section critique.

```
typedef struct {
    int nb;
    int *t;
} Tdonnees;

int min = 1000;
pthread_mutex_t mutex;

void* calcule_min(void* donnees){
    Tdonnees *d;
    int i, m = 1000;
    d = (Tdonnees*)donnees;
    for (i=0; i<d->nb; i++) if ((d->t)[i]<m) m = (d->t)[i];
    pthread_mutex_lock(&mutex);
    if (m<min) min=m;
    pthread_mutex_unlock(&mutex);
}

main(){
```

1. Les primitives `pthread_mutexattr_init` et `int pthread_mutexattr_settype_np` permettent d'initialiser `*mutexattr` à d'autres valeurs.

```

pthread_t thread1, thread2, thread3;
int tab[15] = {8,3,9,5,7,12,7,45,9,3,45,12,6,6,9};
Tdonnees d1, d2, d3;
d1.nb = 5; d1.t = &(tab[0]);
d2.nb = 5; d2.t = &(tab[5]);
d3.nb = 5; d3.t = &(tab[10]);
pthread_mutex_init(&mutex, NULL);
pthread_create( &thread1, NULL, calcule_min, (void*) &d1);
pthread_create( &thread2, NULL, calcule_min, (void*) &d2);
pthread_create( &thread3, NULL, calcule_min, (void*) &d3);
pthread_join(thread1,NULL);
pthread_join(thread2,NULL);
pthread_join(thread3,NULL);
printf("Le minimum global est %d\n",min);
}

```

5.3 Les variables de condition

Les variables de condition permettent de synchroniser l'exécution de threads en suspendant leur exécution jusqu'à ce qu'une certaine condition soit vérifiée. Les opérations fondamentales sur les conditions sont :

- signaler la condition,
- attendre la condition (suspendre l'exécution du thread) jusqu'à ce qu'un autre thread signale la condition.

La primitive

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
```

initialise la variable condition `*cond`, en utilisant les attributs de condition spécifiés par `*cond_attr`, ou les attributs par défaut si `cond_attr = NULL` (...l'implémentation LinuxThreads ne supportant aucun attribut de conditions, le paramètre `cond_attr` est en fait ignoré...).

La primitive

```
int pthread_cond_signal(pthread_cond_t *cond);
```

relance l'exécution de l'un des threads attendant la variable condition `*cond`. S'il n'existe aucun thread répondant à ce critère, rien ne se produit. Si plusieurs threads attendent sur `*cond`, seul l'un d'entre eux sera relancé... mais il est impossible de savoir lequel!

La primitive

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

relance tous les threads attendant sur la variable condition `*cond`. Rien ne se passe s'il n'y a aucun thread attendant sur `*cond`.

La primitive

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

déverrouille atomiquement le verrou d'exclusion mutuelle `*mutex` (comme `pthread_unlock_mutex`), puis attend que la variable condition `*cond` soit signalée. L'exécution du thread est alors suspendue et ne consomme pas de temps CPU jusqu'à ce que la variable condition soit signalée. Quand ce signal arrive, la primitive `pthread_cond_wait` reverrouille `*mutex` (comme `pthread_lock_mutex`). Notons que le mutex doit être déjà verrouillé avant l'appel à `pthread_cond_wait`.

La raison de ce déverrouillage puis verrouillage automatique à l'intérieur du `wait` est que celui-ci sera généralement fait dans une section critique (protégeant la ou les variables sur lesquelles on souhaite synchroniser les threads); dans ce cas, si on déverrouillait ce mutex juste avant le `wait`, on prendrait le risque qu'un autre thread fasse exactement la même chose en même temps (et qu'il y ait une incohérence), tandis que si on ne déverrouillait pas le mutex avant le `wait` alors tous les autres threads seraient également bloqués en attente de ce déverrouillage... Ainsi, lors de l'exécution du `wait`, le déverrouillage du mutex et la suspension de l'exécution sont effectués atomiquement, ce qui garantit la cohérence de l'exécution.

Exemple de threads avec exclusion mutuelle et synchronisation. On veut décrémenter chaque élément d'un tableau par la valeur de son plus petit élément. Dans une première étape, chaque thread calcule le plus petit élément d'une portion du tableau, et met à jour la variable globale min (si nécessaire) à la fin. La mise à jour de cette variable est faite dans une section critique. Ensuite, chaque thread attend que les autres aient fini cette première étape pour retrancher la valeur de min aux éléments de sa portion de tableau.

```

typedef struct {
    int nb;
    int *t;
} Tdonnees;

int min = 1000; // valeur du plus petit élément du tableau
int cpt; // nombre de threads ayant terminé la première étape
pthread_mutex_t mutex; // sémaphore protégeant l'accès à min et cpt
pthread_cond_t synchro; // variable de condition pour déclencher la décrémentation

void* calcule_min_et_decremente(void* donnees){
    Tdonnees *d;
    int i;
    int m = 1000;
    d = (Tdonnees*)donnees;
    for (i=0; i<d->nb; i++) if ((d->t)[i]<m) m = (d->t)[i];
    pthread_mutex_lock(&mutex); // Début de section critique
    if (m<min) min=m;
    cpt++;
    if (cpt<3) pthread_cond_wait(&synchro,&mutex); // on attend les autres threads
    else pthread_cond_broadcast(&synchro); // tous les autres threads attendent => on les réveille
    pthread_mutex_unlock(&mutex); // Fin de section critique
    // tous les threads ont fini de calculer leur min
    for (i=0; i<d->nb; i++) (d->t)[i] -= min;
}

main(){
    pthread_t thread1, thread2, thread3;
    int tab[15] = {8,3,9,5,7,12,7,45,9,3,45,12,6,6,9};
    Tdonnees d1, d2, d3;
    d1.nb = 5; d1.t = &(tab[0]);
    d2.nb = 5; d2.t = &(tab[5]);
    d3.nb = 5; d3.t = &(tab[10]);
    pthread_mutex_init(&mutex, NULL);
    pthread_mutex_init(&mutexSynchro, NULL);
    pthread_cond_init(&synchro, NULL);
    pthread_create( &thread1, NULL, calcule_min_et_decremente, (void*) &d1);
    pthread_create( &thread2, NULL, calcule_min_et_decremente, (void*) &d2);
    pthread_create( &thread3, NULL, calcule_min_et_decremente, (void*) &d3);
    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);
    pthread_join(thread3,NULL);
    printf("Le tableau contient ");
    for (i=0; i<15; i++) printf("%d ",tab[i]); printf("\n");
}

```