Notes sur une introduction aux Threads

Master 1^{iere} année, Bruno Jacob

Janvier 2005

1 Introduction

Les processus sont des entités *a priori* indépendantes. Chaque processus dispose de ses données et de sa pile d'exécution et exécute un code non modifiable qui peut être partagé. Si l'on veut que des processus puissent communiquer ou partager des informations, alors il faut le demander explicitement au système d'exploitation par les outils que nous avons vus précédemment (tubes, files de messages et segments de mémoires partagées). Cet isolement des processus se répercute dans le coût de l'opération de changement de contexte qui nécessite des sauvegardes/restaurations de nombreuses informations.
Pour accélérer les changements de contexte entre ces applications et favoriser leurs partages de données, la notion de processus a été raffinée au concept de *threads* encore appelés processus légers (*light-weighted process*) ou fils d'exécution.
Ces notes sont une introduction à l'utilisation des processus légers.

1.1 Le concept d'activité (threads)

Si on reprend la notion de processus telle qu'on la retrouve dans les systèmes tels qu'UNIX, elle recouvre à la fois le concept de machine virtuelle fournissant des ressources (une unité d'encapsulation de ressources) et celui de programme en train de s'y exécuter (fil de contrôle). Le concept de processus classique est ainsi éclaté en deux :

- le concept de processus (appelé également acteur ou tâche selon les sytèmes) qui correspond à la machine virtuelle. Il contient et définit par exemple les ressources suivantes extraites d'un processus UNIX classique :
  - un espace d'adressage (régions de programme et de données),
  - un propriétaire et un groupe propriétaire,

  - des descripteurs de fichiers,
  - un répertoire de travail,
  - des handlers de signaux ;
- le concept d'activité (thread) qui définit le contexte d'une exécution d'un programme par un processus (sur les données fournies également par ce processus). Les ressources propres à une thread seront donc celles permettant de faire progresser l'exécution d'un programme. On pourra y retrouver :
  - les registres (ou leur copie),
  - les attributs d'ordonnancement (état, classe et priorité par exemple),
  - des signaux masqués et des signaux pendants,
  - une pile.

Une thread appartient à un processus et toutes les threads d'un même processus partagent les ressources de celui-ci. En particulier, elles travaillent sur les mêmes régions de données et programme. Donc

- une même adresse logique correspondra dans chaque thread à la même donnée physique
- les threads peuvent accéder à toutes les fonctions du programme. Elles peuvent donc exécuter des fonctions différentes ou éventuellement la même
- la fermeture d'un descripteur de fichier par l'une des threads d'un processus interdira l'utilisation de ce descripteur dans toutes les threads du processus.
- Les variables
  - **globales** peuvent être partagées par les threads
  - **locales** aux fonctions, et en particulier par la fonction `main` ne peuvent être partagées puisque qu'elles appartiennent à la pile

Le concept de processus classique correspond dans ce nouveau schéma à une procédure de démarrage contenant exactement une thread.

```
{int r ; main(argc, argv) ; exit(r) ;}
```

Certains systèmes imposent qu'il y ait toujours au moins une thread dans un processus, d'autres non (par exemple Chorus).

1.2 Avantages

Les avantages des threads se situent principalement dans la conception des programmes parallèles et le gain de rapidité de ces programmes.

1.2.1 Programmation parallèle

Le concept de thread est adapté à l'implémentation de différentes techniques utilisées dans le domaine de la programmation parallèle :

- la modularité : applications dans lesquelles il est possible d'identifier des composants distincts et leurs interactions afin de produire le service ou les résultats recherchés. Dans ce cas, plutôt que d'implémenter l'application comme une seule grosse entité, on implémente indépendamment les différents composants. Le concept de thread fournit un moyen simple pour partager les informations entre les composants et des mécanismes pour synchroniser leur exécution ;
- les modèles de logiciels :
  - le modèle client/serveur (maître /esclave) ; si on considère par exemple un serveur Internet développé au dessus de TCP en utilisant l'interface des sockets, la technique usuelle consistait à charger un nouveau processus (créé par fork) de gérer une nouvelle connection. Si pour la gestion de cette connection, des données sont partagées avec des processus gérant d'autres connections, il est nécessaire de solliciter le système (soit pour communiquer, soit pour attacher de la mémoire partagée). Utiliser des threads pour gérer les connections permet de communiquer naturellement et de réaliser le maximum d'opérations en mode utilisateur.
  - le modèle producteur/consommateur dont le mécanisme de tube (pipe) est une illustration ;
  - diviser pour mieux régner où le travail est réparti entre différentes entités indépendantes.

1.2.2 Gain de performances

De manière évidente, sur une machine multiprocesseurs, où un véritable parallélisme d'exécution est possible, la modularité des applications et/ou l'utilisation d'un des modèles d'application présenté précédemment améliore les performances. Sur des machines monoprocesseur, l'utilisation de threads conduit également à des gains de performances et une meilleure utilisation des ressources. En effet, créer une nouvelle thread est notablement moins coûteux que créer un processus ne serait-ce que par le fait que l'espace d'adressage ne fait pas partie des attributs d'une activité (mais du processus qui l'héberge). Il en va de même du changement de contexte entre threads d'un même processus qui ne sollicite pas le gestionnaire de mémoire.

1.3 Inconvénients

Un inconvénient, ou une précaution à prendre, dans l'utilisation des *threads* est qu'il est nécessaire de n'utiliser dans les applications que des fonctions qui permettent la réentrance, c'est-à-dire utilisables simultanément par plusieurs threads (et donc dites *Multi Threading Safe*).

1.4 Kernel threads *vs* User threads

Il s'agit là de deux approches vraiment différentes :

1.4.1 Users threads

Cette aproche es implémentation des threads est appalé *l'approche M-1 (Many-to-One).*
Avec les threads en mode utilisateur (user threads), l'état de la thread est maintenu en espace utilisateur. Aucune ressource du noyau n'est allouée à une thread. Un certain nombre d'opérations peuvent ainsi être réalisées indépendamment du système. Cela accélère évidemment le changement de contexte. En contrepartie, si le noyau ne contient lui-même qu'une seule thread, une opération bloquant une thread d'un processus (par exemple une lecture bloquante), toutes les threads du processus seront bloquées. Cette approche a été la première proposée dans les systèmes Unix de conception classique au travers de bibliothèques (en particulier les Posix threads, Pthreads en abrégé). Les informations concernant les threads appartenant à un processus sont gérées dans l'espace d'adressage du processus.
Cette approche est évidemment celle adoptée par les systèmes ne supportant pas le multithreading au niveau noyau.

Dans la figure FIG. 1 montre ce type d'implémentation, on voit que chaque processus possède sa propre table de threads

Avantages :

- Ils ont gérés "logiciellement" par une bibliothèque, ils peuvent donc s'exécuter indépendamment du système d'exploitation.
- Chaque processus dispose de son propore algorithme d'ordonnancement des threads
- Les threads évoluent plus rapidement dans l'espace utilisateur que dans le noyau car la gestion de la mémoire (pour les tables et les piles des threads) est plus lourde dans le noyau

Inconvénients :

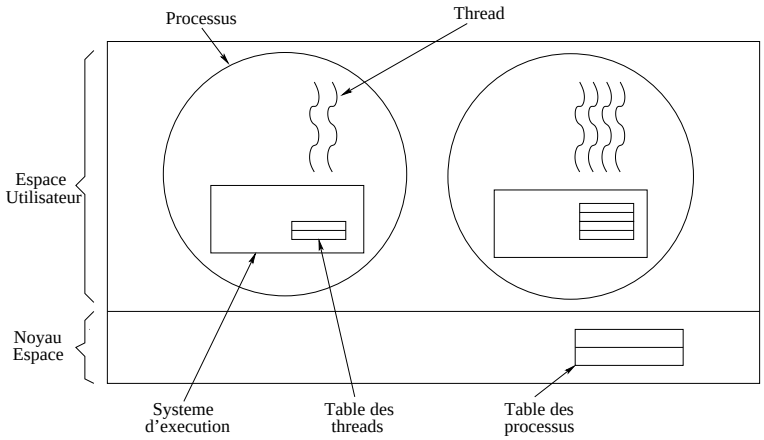

FIG. 1 – Un paquetage de threads au niveau utilisateur

- Aucun parallélisme n'est possible entre les threads utilisateurs d'une même thread noyau.
- Un appel système bloquant d'un thread (un `read` au clavier par exemple) bloque tout le processus, donc tous les threads. Pour pallier à ce problème, il faudrait réécrire les primitives des appels système pour les rendre non boquants (positionnement du bit `O_NONBLOCK` avec `fcntl` dans les attributs du fichier à lire) et scruter l'arrivée de données (par un `select` en BSD ou un `poll` en System V),
- Ce problème est aggravé par le fait que les programmeurs développent gnéralement les threads dans les applications où ceux ci bloquent souvent (serveur Web multithread).

1.4.2 Kernel threads

Cette approche d'implémentation est aussi appelée *l'approche 1-1 (One-to-One)*.
Avec les threads en mode superviseur (kernel threads, ou threads noyau) les threads sont des entités du système : le système possède un descripteur de chacune de ces threads et l'ordonnanceur du système travaille au niveau de ces entités. Dans le cas d'une architecture multiprocesseurs, cela permet une utilisation fine de ces différents processeurs. Les systèmes fournissant les threads en mode superviseur offrent une interface spécifique pour les manipuler.
La figure FIG. 2 montre ce type d'implémentation,
Cette approche est celle de NT, OS2 et Chorus par exemple.

Avantage : les threads sont traitées individuellement par le noyau (le blo-

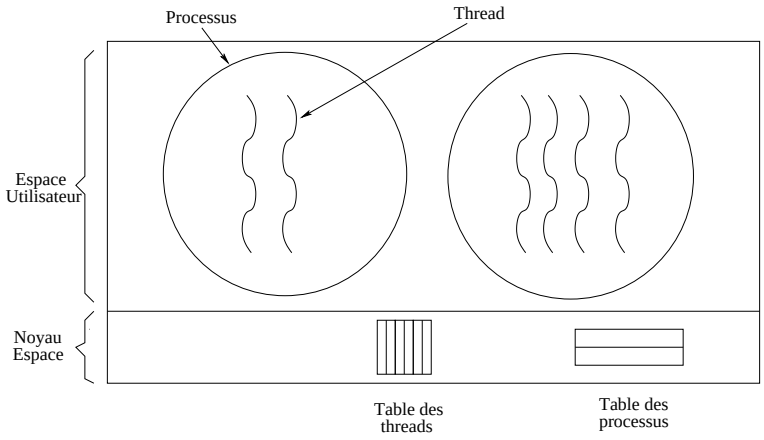

FIG. 2 – Un paquetage de threads au niveau noyau

cage d'une thread d'un processus ne bloquera pas toutes les threads du processus)

Inconvénient :

- la création et la destruction d'un thread est beaucoup plus lourde que dans l'espace utilisateur.
- la gestion des appels système est beaucoup plus complexe ; si ces opérations sont nombreuses, cela entraine uen surchage importante.

suppose la crétion de sa correspondante dans le noyau et des ressources correspondantes.

1.4.3 Threads hybrides

Cette approche d'implémentation est aussi appelée *l'approche M-M (Many-to-Many)*.

Différentes threads utilisateur sont multiplexées sur un nombre (inférieur ou égal) de threads noyau. DAns ce modèle, chaque thread noyau possède un ensemble donné de threads utilisateurs.qui l'utilisent tour à tour.
Il est possible dans ce système d'associer de manière permanente une thread utilisateur à une thread noyau ou non. La figure FIG. 3 montre ce type d'implémentation. C'est la solution adoptée dans Solaris.

2 Threads de Solaris

Dans la terminologie Solaris :

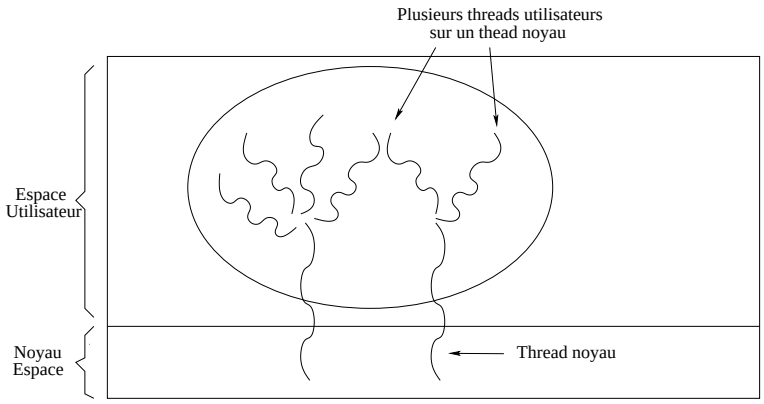

FIG. 3 – Multiplexage des threads utilisateurs et noyaux

- thread noyau = LWP (Lightweight Processes)
- thread utilisateur = user thread

Le système attribue un ou plusieurs LWP à chaque processus. Il répartit alors chaque thread utilisateur créé sur les LWP (unbound thread). L'utilisateur à également la possibilité de modifier ce comportement par défaut en attachant explicitement un thread utilisateur à un LWP du noyau (bound thread) comme le montre la figure FIG. 4 .

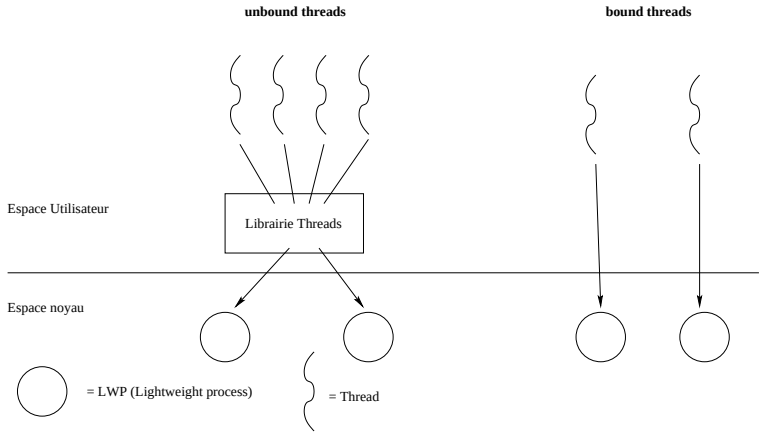

FIG. 4 – Threads utilisateurs et noyaux dans Solaris

3 Fonctions de gestion des threads

L'implémentation des threads est très dépendante du langage et du système utilisés. Afin de masquer les spécificités sous-jacentes des systèmes et que vos

programmes soient les plus portables possible, nous allons présenter ici les fonctions de gestion des threads de la norme POSIX (*Portable Operating System Interface*) appelés aussi les *pthreads*.

3.1 Généralités

3.1.1 Noms de fonctions

La norme POSIX a retenu pour les noms de fonctions le format suivant :

`pthread`[_*objet*]_*opération*[`_np`]

avec

- *objet* : type d'objet sur lequel s'applique la fonction. Les objet peuvent être :
  - `cond` désigne le type *variable de condition*
  - `mutex` désigne le type *sémaphore d'exclusion mutuelle (ou mutex)*
- *opération* opération à réa ;liser sur l'objet (par exemple `create`, `exit`,...)
- `np` indique que la fonction est non portable, qu'il s'agit de fonctions dépendantes de la machine et/ou du système utilisés.

Le format pour les types est le suivant :

`pthread_`*objet*`_t`

3.1.2 Identification

Une activité/pthread est identifiée par une variable de type `pthread_t`(un `unsigned int` dans Unix sur Solaris) ;
Une activité peut obtenir son identifiant par la fonction

```
pthread_t pthread_self(void);
```

3.1.3 Approche M-M

L'interface Posix prend en compte l'approche M-M en ajoutant un attribut spécifique à chaque pthread créée qui indique si on doit lui associer un thread noyau ou non. Cet attribut est appelé *attribut de contention* dont les valeurs possibles sont

- `PTHREAD_SCOPE_SYSTEM` : chaque thread utilisateur est associé à un seul thread noyau
  - Dans Solaris, cela correspond à un *bound thread*
  - Dans Linux, ce n'est pas une option, un thread utilisateur est obligatoirement associé à un thread noyau

  Donc, les processus et les threads sont en égale concurrence pour l'allocation du processeur

- `PTHREAD_SCOPE_PROCESS` : tous les threads utilisateurs sont répartis parmis l'ensemble (pool) des threads noyaux attribué à ce processus. C'est la m'ethode de création par défaut dans la norme POSIX.
  - Dans Solaris, cela correspond au *unbound thread*
  - Dans Linux, cette option de POSIX n'est pas implémentée

  Les threads possédant cet attribut de contention partagent donc un pool de threads noyau. Il est alors nécessaire de spécifier dans le processus qui creer les threads (typiquement dans la thread principale du processus) le degré de concurrence souhaité entre tous les threads au moyen de la fonction `pthread_setconcurrency`

3.1.4 Valeurs de retour

La majorité des fonctions de la bibliothèque des pthreads renvoie un entier
- $= 0$ si OK
- $\neq 0$ sinon. Dans ce cas `errno` est mis à jour et ses valeurs peuvent être :
  - `ENOMEM` : plus de place pour créer une autre thread
  - `EINVAL` : mauvais argument dans l'appel des fonctions
  - `EPERM` : permission non accordée
  - `EBUSY` : destruction d'un objet en train d'être utilisé, test de verrouillage
  - `EFAULT` : mauvais pointeur

3.1.5 Compilation

Création programme-objet

```
#include <pthread.h>
```

Édition de liens

1. Faire le lien avec bibliothèque `pthread`.

   ```
   gcc -mthreads file... -lpthread [ library... ]
   ```

2. Définir `_REENTRANT` soit
   - dans le code source en ajoutant

     ```
     #define _REENTRANT
     ```

   - à la compilation en ajoutant à l'option

     ```
     gcc ... -D_REENTRANT
     ```

3.2 Création

```
int pthread_create( pthread_t *thread,           /* ident. du thread cree */
                    const pthread_attr_t *attr, /* attributs ou NULL */
                    void *(*fonction, void*),   /* fonction a executer */
                    void *arg);                 /* parametres fonction */
```

Céer une nouvelle activité (d'identifiant `thread`) pour exécuter la fonction `fonction` appelée avec les paramètres contenus à l'adresse `arg` (qui peut correspondre à l'adresse d'une structure contenant plusieurs paramètres). Si `attr` est à `NULL` alors le thread adopte le comportement par défaut. Si on veut modifier ce comportement, on peut modifier les champs de `attr` de la manière suivante :

1. Récuperer les valeurs des champs par défaut avec `pthread_attr_init`
2. Modifier ces valeurs avec les méthodes de cette structure (les fonctions `get/set_attr_*`)

Fonction de création/destruction des attributs de threadd :

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

Les valeurs par défault sont les suivantes :

Nom des champs	Valeur par défaut	Description
`contentionscope`	`PTHREAD_SCOPE_PROCESS`	concurrence de ressources dans le processus
`detachstate`	`PTHREAD_CREATE_JOINABLE`	les autres threads peuvent consulter le code retour
`stackaddr`	`NULL`	pile allouée par le système
`stacksize`	`NULL`	taille de la pile de 1 ou 2 megaoctets
`priority`	`0`	priorité de la thread
`policy`	`SCHED_OTHER`	ordonnancement défini par le système
`inheritsched`	`PTHREAD_EXPLICIT_SCHED`	l'ordonnacement et les paramètres du thread ne sont pas hérités de son père mais définis explicitement
`guardsize`	`PAGESIZE`	taille du secteur pour la pile

Les méthodes permettant de consulter et de modifier ces champs sont les suivantes :

```
pthread_attr_getdetachstate(const pthread_attr_t *attr , int * etat );
pthread_attr_getguardsize(const pthread_attr_t *attr, size_t * taille);
pthread_attr_getinheritsched(const pthread_attr_t *attr, int * herite);
pthread_attr_getschedparam(const pthread_attr_t *attr,
                           struct sched_param *param);
pthread_attr_getschedpolicy(const pthread_attr_t *attr, int * ordre);
pthread_attr_getscope(const pthread_attr_t *attr, int * concurrence);
pthread_attr_getstackaddr(const pthread_attr_t *attr, void **addrpile));
```

```
pthread_attr_getstacksize(const pthread_attr_t *attr, size_t * taille);

pthread_attr_setdetachstate(pthread_attr_t *attr, int )etat;
pthread_attr_setguardsize(pthread_attr_t *attr , size_t taille );
pthread_attr_setinheritsched(pthread_attr_t *attr, int herite);
pthread_attr_setschedparam(pthread_attr_t *attr,
                           const struct sched_param *param);
pthread_attr_setschedpolicy(pthread_attr_t *attr, int ordre);
pthread_attr_setscope(pthread_attr_t *attr, int concurrence);
pthread_attr_setstackaddr(pthread_attr_t *attr, void *addrpile));
pthread_attr_setstacksize(pthread_attr_t *attr, size_t taille);
```

3.3 Terminaison

```
void pthread_exit(void *cr); /* adresse du code retour ou NULL */
```

Termine l'activité appelante en fournissant le code de retour pointé par `cr`. Si l'activité se termine sans appel à `pthread_exit` alors `pthread_exit(NULL)` est implicitemet exécuté.

3.4 Attente terminaison

```
int pthread_join( pthread_t thread, /* identifiant du thread */
                  void **cr);       /* zone de recuperation du code retour */
```

Attend la terminaison de l'activité indiquée et récupère le code retour. L'activité ne doit pas être détachée.

3.5 Libération des ressources

Attention : lorsqu'une activité se termine, elle ne disparait pas, contrairement à ce qui se passe avec les processus zombis qui disparaîssent dès que leur père a réalisé un appel à `wait` pour acquérir leur code de retour.
Une activité, elle, peut être attendue par plusieurs autres threads (par un appel à `pthread_join`) du même processus. Donc les ressources qui lui sont allouées ne sont pas restituées (en particulier sa pile d'exécution) dès sa terminaison ou au premier appel à `pthread_join`). La libération de ses ressources doit être explicitement demandée par la fonction :

```
int pthread_detach(pthread_t thread); /* identifiant du thread */
```

Un appel à cette fonction ne termine pas l'activité en cours d'exécution. Elle indique qu'à sa terminaison ses ressources devront être restituées au processus auquelle elle appartient.

4 Exemples

4.1 Utilisation des pthreads seules

Nous reprenons ici l'exemple du compteur dans le cours sur les sémaphores, partagé par plusieures activités.

4.1.1 Code source

Le partage de la variable `cpt` est réalisé simplement en la déclarant en variable globale, espace d'adressage commun à toutes les threads de ce processus.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

unsigned long int cpt ;

static
void fonc()
{
      int m = 0;
      while(m < 10000000)
      {
            m++;
            cpt++;
      }
      printf("m = %d cpt = %lu\n", m, cpt);
      pthread_exit(NULL);
}

int
main()
{
      int i ;
      pthread_attr_t attr ;
      pthread_t thread_id[2];
      /*-----*/

      /* -- init */
      cpt = 0;

      /* Creation de 2 fils processus legers */
```

```
    pthread_attr_init(&attr) ;
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    pthread_setconcurrency(2) ;
    pthread_create( &thread_id[0], &attr , (void *)fonc , (void *)NULL ) ;
    pthread_create( &thread_id[1], &attr , (void *)fonc , (void *)NULL ) ;

    /* Attente terminaison des fils */
    for(i=0 ; i<2 ; i++ )
        pthread_join(thread_id[i],NULL);

    /* -- affichage ressource critique */
    printf("Valeur finale de cpt : %lu\n", cpt);

    exit(0);
}
```

4.1.2 Exécution

- Une première exécution donne

```
m = 10000000 cpt = 4742250
m = 10000000 cpt = 5583090
Valeur finale de cpt : 5583090
```

- pour la deuxième :

```
m = 10000000 cpt = 4732333
m = 10000000 cpt = 5190726
Valeur finale de cpt : 5190726
```

On voit que l'on retombe sur le même problème d'accès à une ressource critique. Les pthreads ne gèrent pas cela, il faut le faire explicitement.

4.2 Utilisation threads + sémaphores

Ici, on utilise les sémaphores UNIX pour gérer l'accès à la variable partagée `ctp`

4.2.1 Code source

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
```

```
unsigned long int cpt ;

static
void fonc()
{
      int m = 0;
      while(m < 10000000)
      {
            m++;
            cpt++;
      }
      printf("m = %d cpt = %lu\n", m, cpt);
      pthread_exit(NULL);
}

int
main()
{
      int i ;
      pthread_attr_t attr ;
      pthread_t thread_id[2];
      /*------*/

      /* -- init */
      cpt = 0;

      /* Creation de 2 fils processus legers */
      pthread_attr_init(&attr) ;
      pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
      pthread_setconcurrency(2) ;
      pthread_create( &thread_id[0], &attr , (void *)fonc , (void *)NULL ) ;
      pthread_create( &thread_id[1], &attr , (void *)fonc , (void *)NULL ) ;

      /* Attente terminaison des fils */
      for(i=0 ; i<2 ; i++ )
            pthread_join(thread_id[i],NULL);

      /* -- affichage ressource critique */
      printf("Valeur finale de cpt : %lu\n", cpt);

      exit(0);
}
```

4.2.2 Exécution

Une exécution de cette solution donne un résultat correct :

```
m = 1000000 *p_cpt = 1600644
m = 1000000 *p_cpt = 2000000
```

```
Valeur finale de cpt : 2000000
```

Avec un temps :

```
real        8.8
user        3.5
sys         9.2
```

5 Sémaphores `mutex`

Les *mutex* sont sont les sémaphores d'exclusion mutuelle : c'est à dire qu'un seul processus à la fois peut exécuter une section de code critique. Dans le cas général il s'agit donc d'un ensemble d'un seul sémaphore qui est initialisé à la valeur 1.

La bibliothèque des threads POSIX fournit aussi un mécanisme de sémaphore mutex (appelé aussi *mécanisme de verrouillage*) pour controler les accès aux variables globales partagées par toutes les activités.

5.1 Création

La création d'un mutex de pthread peut se faire par l'appel de la fonction

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

Si `attr` est `NULL` alors les attributs par défaut sont utilisés. Dans ce cas cela équivaut à l'affectation

```
pthread_mutex_t mutex= PTHREAD_MUTEX_INITIALIZER
```

5.2 Entrée

L'entrée dans la section critique, ou la décrémentation de la valeur du sémaphore, ou encore le verrouillage de cette zone est réalisée par

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

- si $M = 1$ alors $M = M - 1$ et le thread continue son excécution (il peut entrer en section critique)
- si $M = 0$ alors M ne change pas et le thread est bloqué jusqu'à ce que $M = 1$

5.3 Sortie

La sortie de la section critique, ou l'incrémentation de la valeur du sémaphore, ou encore le déverrouillage de cette zone est réalisée par

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

5.4 Test

Le thread courant peut tester si il est possible d'entrer en section critique avant de le faire effectivement. L'intérêt es que, si ce n'est pas possible, l'activité n'est pas bloquée. Donc si un thread teste un mutex M

- si $M = 1$ alors $M = M - 1$ et le thread continue son excécution (il peut entrer en section critique)
- si $M = 0$ alors M ne change pas et le thread n'est pas bloqué mais il re,coit une valeur de retour correspondant à ue erreur.

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

5.5 Destruction

La destruction du sémaphore mutex est rélisée par

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

5.6 Remarque

On peut paramétrer les mutex de manière à ce qu'ils deviennent des sémaphores plus généraux en utilisant le paramètre `attr` de `pthread_mutex_init`

6 Exemple d'utilisation pthreads et mutex

On reprend l'exmeple du compyteur patrtagé par plusieurs threads. L'accès à cette varaible peut être géré par un sémaphore d'exclusion mutuelle. On peut donc utiliser le `mutex` de la bibliothèque des `pthreads`.

6.1 Code source

Le `mutex` est lui aussi en variable globale pour qu'il puisse être connu de toutes les threads.

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

/* Ressource critique  */
unsigned long int cpt ;

/* Mutex */
pthread_mutex_t mutex  = PTHREAD_MUTEX_INITIALIZER ;

static
void fonc()
{
     int m = 0;
     while(m <1000000 )
     {
          m++;
          /* Entree section critique */
          pthread_mutex_lock(&mutex);
          /* Utilisation ressource critique */
          cpt++;
          /* Sortie section critique */
          pthread_mutex_unlock(&mutex);
     }
     printf("m = %d *p_cpt = %lu\n", m, cpt);
     pthread_exit(NULL);
}

int
main()
{
     int i ;
     pthread_attr_t attr ;
     pthread_t thread_id[2];
     /*-----*/

     /* Init. ressource critique */
     cpt = 0;

     /* Creation de 2 fils processus legers */
     pthread_attr_init(&attr) ;
     pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
     pthread_setconcurrency(2) ;
     pthread_create( &thread_id[0], &attr , (void *)fonc , (void *)NULL ) ;
     pthread_create( &thread_id[1], &attr , (void *)fonc , (void *)NULL ) ;
```

```
    /* Attente terminaison des fils */
    for(i=0 ; i<2 ; i++ )
        pthread_join(thread_id[i],NULL);

    /* -- affichage ressource critique */
    printf("Valeur finale de cpt : %lu\n", cpt);

    /* Destruction mutex */
    pthread_mutex_destroy(&mutex);

    exit(0);
}
```

6.2 Exécution

L'exécution de ce code donne

```
m = 1000000 *p_cpt = 1665818
m = 1000000 *p_cpt = 2000000
Valeur finale de cpt : 2000000
```

avec le temps

```
real        0.4
user        0.4
sys         0.0
```

On voit ici que, dans le cas de la réalisation d'un sémaphore d'exclusion mutuelle, les objets mutex de la bibliothèque des pthreads sont plus performants que les sémaphores généraux du système.