TP3 : Les threads

Les threads constituent une bases de la programmation système sous Unix : l'étude des *threads Posix* (*Posix* est le nom d'une famille de standards qui indique un code standard).

2-1-Qu'est ce qu'un thread ?

Dans la plupart des systèmes d'exploitation, chaque processus possède un espace d'adressage et un **thread** de contrôle unique, le **thread principal**. Du point de vue programmation, ce dernier exécute le **main**.

Vous avez pu remarquer, lors de notre étude des processus, qu'en général, le système réserve un processus à chaque application, sauf quelques exceptions. Beaucoup de programmes exécutent plusieurs activités en parallèle, du moins en apparent parallélisme, comme nous l'avons vu précédemment. Comme à l'échelle des processus, certaines de ces activités peuvent se bloquer, et ainsi réserver ce blocage à un seul thread séquentiel, permettant par conséquent de ne pas stopper toute l'application.

Ensuite, il faut savoir que le principal avantage des threads par rapport aux processus, c'est la facilité et la rapidité de leur création. En effet, tous les threads d'un même processus partagent le même espace d'adressage, et donc toutes les variables. Cela évite donc l'allocation de tous ces espaces lors de la création, et il est à noter que, sur de nombreux systèmes, la création d'un thread est environ cent fois plus rapide que celle d'un processus.

Au-delà de la création, la superposition de l'exécution des activités dans une même application permet une importante accélération quant au fonctionnement de cette dernière.

Sachez également que la communication entre les threads est plus aisée que celle entre les processus, pour lesquels on doit utiliser des notions compliquées comme les tubes (voir chapitre suivant).

Le mot « *thread* » est un terme anglais qui peut se traduire par « *fil d'exécution* ». L'appellation de « *processus léger* » est également utilisée.

Compilation

Toutes les fonctions relatives aux *threads* sont incluses dans le fichier d'en-tête `<pthread.h>` et dans la bibliothèque `libpthread.a` (soit `-lpthread` à la compilation).

Exemple :

Écrivez la ligne de commande qui vous permet de compiler votre programme sur les *threads* constitué d'un seul fichier **main.c** et avoir en sortie un exécutable nommé **monProgramme**.

Correction:

```
gcc -lpthread main.c -o monProgramme
```

Et n'oubliez pas d'ajouter #include <pthread.h> au début de vos fichiers.

2-2-Manipulation des threads

2-2-1-Créer un thread

Pour créer un thread, il faut déjà déclarer une variable le représentant. Celle-ci sera de type `pthread_t` (qui est, sur la plupart des systèmes, un `typedef` d'`unsigned long int`).

Ensuite, pour créer la tâche elle-même, il suffit d'utiliser la fonction :

```
#include <pthread.h>

int pthread_create(pthread_t * thread, pthread_attr_t * attr,

           void *(*start_routine) (void *), void *arg);
```

Ce prototype est un peu compliqué, c'est pourquoi nous allons récapituler ensemble.

- La fonction renvoie une valeur de type **int** : 0 si la création a été réussie ou une autre valeur s'il y a eu une erreur.
- Le premier argument est un pointeur vers l'identifiant du **thread** (valeur de type `pthread_t`).
- Le second argument désigne les attributs du thread. Vous pouvez choisir de mettre le thread en état *joignable (par défaut) ou détaché, et choisir sa politique d'ordonnancement (usuelle, temps-réel...).* Dans nos exemples, on mettra généralement NULL.
- Le troisième argument est un pointeur vers la fonction à exécuter dans le thread. Cette dernière devra être de la forme `void *fonction(void* arg)` et contiendra le code à exécuter par le thread.
- Enfin, le quatrième et dernier argument est l'argument à passer au thread.

2-2-2-Supprimer un thread

Et qui dit créer dit supprimer à la fin de l'utilisation.
Cette fois, ce n'est pas une fonction casse-tête :

```
#include <pthread.h>



void pthread_exit(void *ret);
```

Elle prend en argument la valeur qui doit être retournée par le thread, et doit être placée en dernière position dans la fonction concernée.

2-2-3-Première application

Voici un premier code qui réutilise toutes les notions des threads que nous avons vu jusque-là.

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

void *thread_1(void *arg)
```

```c
{
    printf("Nous sommes dans le thread.\n");

    /* Pour enlever le warning */
    (void) arg;
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t thread1;

    printf("Avant la création du thread.\n");

    if(pthread_create(&thread1, NULL, thread_1, NULL) == -1) {
    perror("pthread_create");
    return EXIT_FAILURE;
    }

    printf("Après la création du thread.\n");

    return EXIT_SUCCESS;
}
```

On compile, on exécute. Et là, zut... Le résultat, dans le meilleur des cas, affiche le message de thread en dernier. Dans le pire des cas, celui-ci ne s'affiche même pas (ce qui veut dire que le return s'est exécuté avant le thread...). Ce qui est normal, puisqu'en théorie, comme avec les processus, le thread principal ne va pas attendre de lui-même que le thread se termine avant d'exécuter le reste de son code.

Par conséquent, il va falloir lui en faire la demande. Pour cela, dans `pthread`, il y a la fonction `pthread_join`.

- **Attendre la fin d'un thread**

Voici le prototype de cette fameuse fonction :

```c
#include <pthread.h>

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

Elle prend donc en paramètre l'identifiant du thread et son second paramètre, un pointeur, permet de récupérer la valeur retournée par la fonction dans laquelle s'exécute le thread (c'est-à-dire l'argument de `pthread_exit`).

Exercice résumé

.

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

void *thread_1(void *arg)
{
    printf("Nous sommes dans le thread.\n");

    /* Pour enlever le warning */
    (void) arg;
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t thread1;

    printf("Avant la création du thread.\n");

    if (pthread_create(&thread1, NULL, thread_1, NULL)) {
    perror("pthread_create");
    return EXIT_FAILURE;
    }

    if (pthread_join(thread1, NULL)) {
    perror("pthread_join");
    return EXIT_FAILURE;
    }

    printf("Après la création du thread.\n");

    return EXIT_SUCCESS;
}
```

Et le résultat est enfin uniforme :

```
Avant la création du thread.
Nous sommes dans le thread.
Après la création du thread.
```

2-3-Exclusions mutuelles

2-3-1-Problématique

Avec les threads, toutes les variables sont partagées : c'est la **mémoire partagée**. Mais cela pose des problèmes. En effet, quand deux threads cherchent à modifier deux variables en même temps, que se passe-t-il ? Et si un thread lit une variable quand un autre thread la modifie ?

C'est assez problématique. Par conséquent, nous allons voir une mécanisme de synchronisation : les **mutex**, un des outils permettant l'**exclusion mutuelle**.

2-3-2-Les mutex

Concrètement, un **mutex** est en C une variable de type `pthread_mutex_t`. Elle va nous servir de verrou, pour nous permettre de protéger des données. Ce verrou peut donc prendre deux états : **disponible** et **verrouillé**.

Quand un thread a accès à une variable protégée par un mutex, on dit qu'*il tient le mutex*. Bien évidemment, il ne peut y avoir qu'un seul thread qui tient le mutex en même temps.

Le problème, c'est qu'il faut que le mutex soit accessible en même temps que la variable et dans tout le fichier (vu que différents threads s'exécutent dans différentes fonctions). La solution la plus simple consiste à déclarer les mutex en variable globale. Mais il existe une autre solution plus élégante : déclarer le mutex dans une structure avec la donnée à protéger.

Voici un exemple :

```
typedef struct data {

    int var;

    pthread_mutex_t mutex;

} data;
```

Ainsi, nous pourrons passer la structure en paramètre à nos threads, grâce à la fonction `pthread_create`.

En pratique comment manipuler les mutex grâce à pthread ?

2-3-3-Initialiser un mutex

Conventionnellement, on initialise un mutex avec la valeur de la constante PTHREAD_MUTEX_INITIALIZER, déclarée dans `pthread.h`.

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

typedef struct data {
    int var;
    pthread_mutex_t mutex;
} data;

int main(void)
{
    data new_data;

    new_data.mutex = PTHREAD_MUTEX_INITIALIZER;

    return EXIT_SUCCESS;
}
```

2-3-4-Verrouiller un mutex

L'étape suivante consiste à établir une **zone critique**, c'est-à-dire la zone où plusieurs threads ont l'occasion de modifier ou de lire une même variable en même temps. Une fois cela fait, on verrouille le mutex grâce à la fonction :

```c
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mut);
```

2-3-5-Déverrouiller un mutex

A la fin de la zone critique, il suffit de déverrouiller le mutex.

```c
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mut);
```

2-3-6-Détruire un mutex

Une fois le travail du mutex terminé, on peut le détruire :

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mut);
```

2-4-Les conditions

Lorsqu'un *thread* doit patienter jusqu'à ce qu'un événement survienne dans un autre *thread*, on emploie une technique appelée la condition.Quand un *thread* est en attente d'une condition, il reste bloqué tant que celle-ci n'est pas réalisée par un autre *thread*.

Comme avec les *mutex*, on déclare la condition en variable globale, de cette manière :

```
pthread_cond_t nomCondition = PTHREAD_COND_INITIALIZER;
```

Pour attendre une condition, il faut utiliser un *mutex* :

```
int pthread_cond_wait(pthread_cond_t *nomCondition, pthread_mutex_t
*nomMutex);
```

Pour réveiller un *thread* en attente d'une condition, on utilise la fonction :

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

Exemple :

Créez un code qui crée deux *threads* : un qui incrémente une variable compteur par un nombre tiré au hasard entre 0 et 10, et l'autre qui affiche un message lorsque la variable compteur dépasse 20.

Correction :

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

pthread_cond_t condition = PTHREAD_COND_INITIALIZER; /* Création de la condition */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; /* Création du mutex */

void* threadAlarme (void* arg);
void* threadCompteur (void* arg);

int main (void)
{
    pthread_t monThreadCompteur;
    pthread_t monThreadAlarme;

    pthread_create (&monThreadCompteur, NULL, threadCompteur, (void*)NULL);
    pthread_create (&monThreadAlarme, NULL, threadAlarme, (void*)NULL); /* Création des
threads */

    pthread_join (monThreadCompteur, NULL);
    pthread_join (monThreadAlarme, NULL); /* Attente de la fin des threads */

    return 0;
}

void* threadCompteur (void* arg)
{
    int compteur = 0, nombre = 0;

    srand(time(NULL));

    while(1) /* Boucle infinie */
    {
        nombre = rand()%10; /* On tire un nombre entre 0 et 10 */
        compteur += nombre; /* On ajoute ce nombre à la variable compteur */

        printf("\n%d", compteur);

        if(compteur >= 20) /* Si compteur est plus grand ou égal à 20 */
        {
            pthread_mutex_lock (&mutex); /* On verrouille le mutex */
            pthread_cond_signal (&condition); /* On délivre le signal : condition remplie
*/
            pthread_mutex_unlock (&mutex); /* On déverrouille le mutex */

            compteur = 0; /* On remet la variable compteur à 0 */
        }

        sleep (1); /* On laisse 1 seconde de repos */
    }

    pthread_exit(NULL); /* Fin du thread */
}

void* threadAlarme (void* arg)
```

```
{
    while(1) /* Boucle infinie */
    {
        pthread_mutex_lock(&mutex); /* On verrouille le mutex */
        pthread_cond_wait (&condition, &mutex); /* On attend que la condition soit remplie
*/
        printf("\nLE COMPTEUR A DÉPASSÉ 20.");
        pthread_mutex_unlock(&mutex); /* On déverrouille le mutex */
    }

    pthread_exit(NULL); /* Fin du thread */
}
```

Résultat :

```
lucas@lucas-Deskstop:~/Documents$ ./monprog

4
9
18
26
LE COMPTEUR A DÉPASSÉ 20.
9
18
23
LE COMPTEUR A DÉPASSÉ 20.
0
3
5
9
12
19
23
LE COMPTEUR A DÉPASSÉ 20.
0
8
9
10
17
25
LE COMPTEUR A DÉPASSÉ 20.
2
10
10
16
25
LE COMPTEUR A DÉPASSÉ 20.
8
10
18
26
LE COMPTEUR A DÉPASSÉ 20.
0
7
9
^C
```

Terminons ce chapitre par quelques moyens mnémotechniques qui peuvent vous permettre de retenir toutes les notions que l'on a apprises :

- Vous pouvez remarquer que toutes les variables et les fonctions sur les threads commencent par **pthread**_
- pthread_create : *create* = *créer* en anglais (donc *créer le thread)*
- pthread_exit : *exit* = *sortir* (donc *sortir du thread*)
- pthread_join : *join* = *joindre* (donc *joindre le thread*)
- PTHREAD_MUTEX_INITIALIZER : *initializer* = *initialiser* (donc *initialiser le mutex*)
- pthread_mutex_lock : *lock* = *vérouiller* (donc *vérouiller le mutex*)
- pthread_mutex_unlock : *unlock* = *dévérouiller* (donc *dévérouiller le mutex*)
- pthread_cond_wait : *wait* = *attendre* (donc *attendre la condition*)

complements au chapitre 2 : threads

_Exclusion mutuelle_semaphore

5.1 Pointeurs de fonction

Un pointeur de fonctions en C est une variable qui permet de désigner une fonction C. Comme n'importe quelle variable, on peut mettre un pointeur de fonctions soit en variable dans une fonction, soit en paramètre dans une fonction.

On déclare un pointeur de fonction comme un prototype de fonction, mais on ajoute une étoile ($*$) devant le nom de la fonction. Dans l'exemple suivant, on déclare dans le main un pointeur sur des fonctions qui prennent en paramètre un `int`, et un pointeur sur des fonctions qui retournent un `int`.

```
#include <stdio.h>

int SaisisEntier(void)
{
   int n;
   printf("Veuillez entrer un entier : ");
   scanf("%d", &n);
   return n;
}

void AfficheEntier(int n)
{
   printf("L'entier n vaut %d\(\backslash\)n", n);
}

int main(void)
{
  void (*foncAff)(int); {/* d claration d'un pointeur foncAff */
  int (*foncSais)(void); {/*d claration d'un pointeur foncSais */
  int entier;

  foncSais = SaisisEntier;  {/* affectation d'une fonction */
  foncAff = AfficheEntier; {/* affectation d'une fonction */
```

```
  entier = foncSais(); {/* on ex cute la fonction */
  foncAff(entier);   {/* on ex cute la fonction */
  return 0;
}
```

Dans l'exemple suivant, la fonction est passée en paramètre à une autre fonction, puis exécutée.

```
#include <stdio.h>

int SaisisEntier(void)
{
   int n;
   printf("Veuillez entrer un entier : ");
   scanf("%d", &n);
   getchar();
   return n;
}

void AfficheDecimal(int n)
{
   printf("L'entier n vaut %d\(\backslash\)n", n);
}

void AfficheHexa(int n)
{
   printf("L'entier n vaut %x\(\backslash\)n", n);
}

void ExecAffiche(void (*foncAff)(int), int n)
{
  foncAff(n);  {/* ex cution du param tre */
}

int main(void)
{
  int (*foncSais)(void); {/*d claration d'un pointeur foncSais */
  int entier;
  char rep;

  foncSais = SaisisEntier;  {/* affectation d'une fonction */
  entier = foncSais(); {/* on ex cute la fonction */

  puts("Voulez-vous afficher l'entier n en d cimal (d) ou en hexa (x) ?");
  rep = getchar();
  {/* passage de la fonction en param tre : */
  if (rep == 'd')
```

```
    ExecAffiche(AfficheDecimal, entier);
  if (rep == 'x')
    ExecAffiche(AfficheHexa, entier);

  return 0;
}
```

Pour prévoir une utilisation plus générale de la fonction `ExecAffiche`, on peut utiliser des fonctions qui prennent en paramètre un `void*` au lieu d'un `int`. Le `void*` peut être ensuite reconverti en d'autres types par un *cast*.

```
void AfficheEntierDecimal(void *arg)
{
   inte n = (int)arg; /* un void* et un int sont sur 4 octets */
   printf("L'entier n vaut \%d\n", n);
}

void ExecFonction(void (*foncAff)(void* arg), void *arg)
{
  foncAff(arg);  /* exécution du paramètre */
}

int main(void)
{
   int n;
   ...
   ExecFonction(AfficheEntierDecimal, (void*)n);
   ...
}
```

On peut utiliser la même fonction `ExecFonction` pour afficher tout autre chose que des entiers, par exemple un tableau de `float`.

```
typedef struct
   {
      int n;  /* nombre d'éléments du tableau */
      double *tab;  /* tableau de double */
   }TypeTableau;

void AfficheTableau(void *arg)
{
   inte i;
   TypeTableau *T = (TypeTableau*)arg; /* cast de pointeurs */
   for (i=0 ; i<T->n ; i++)
      {
          printf("%.2f", T->tab[i]);
      }
}
```

```
void ExecFonction(void (*foncAff)(void* arg), void *arg)
{
  foncAff(arg);  /* exécution du paramètre */
}

int main(void)
{
   TypeTableau tt;
   ...
   ExecFonction(AfficheTableau, (void*)&tt);
   ...
}
```

5.2 Thread Posix (sous linux)

5.2.1 Qu'est-ce qu'un thread ?

Un *thread* (ou *fil d'exécution* en français) est une parie du code d'un programme (une fonction), qui se déroule parallèlement à d'autre parties du programme. Un premier interêt peut être d'effectuer un calcul qui dure un peu de temps (plusieurs secondes, minutes, ou heures) sans que l'interface soit bloquée (le programme continue à répondre aux signaux). L'utilisateur peut alors intervenir et interrompre le calcul sans taper un `ctrl-C` brutal. Un autre intérêt est d'effectuer un calcul parallèle sur les machines multi-processeur. Les fonctions liées aux thread sont dans la bibliothèque `pthread.h`, et il faut compiler avec la librairie `libpthread.a` :

```
$ gcc -lpthread monprog.c -o monprog
```

5.2.2 Création d'un thread et attente de terminaison

Pour créer un thread, il faut créer une fonction qui va s'exécuter dans le thread, qui a pour prototype :

```
void *ma_fonction_thread(void *arg);
```

Dans cette fonction, on met le code qui doit être exécuté dans le thread. On crée ensuite le thread par un appel à la fonction `pthread_create`, et on lui passe en argument la fonction `ma_fonction_thread` dans un pointeurs de fonction (et son argument `arg`). La fonction `pthread_create` a pour prototype :

```
int pthread_create(pthread_t *thread, pthread_attr_t *attributs,
                   void * (*fonction)(void *arg), void *arg);
```

Le premier argument est un passage par adresse de l'identifiant du thread (de type `pthread_t`). La fonction `pthread_create` nous retourne ainsi l'identifiant du thread, qui l'on utilise ensuite pour désigner le thread. Le deuxième argument `attributs` désigne les attributs du thread, et on peut mettre `NULL` pour avoir les attibuts par défaut. Le troisième argument est un pointeur

sur la fonstion à exécuter dans le thread (par exemple `ma_fonction_thread`, et le quatrième argument est l'argument de la fonction de thread.

Le processus qui exécute le `main` (l'équivalent du processus père) est aussi un thread et s'appelle le *thread principal*. Le thread principal peut attendre la fon de l'exécution d'un autre thread par la fonction `pthread_join` (similaire à la fonction `wait` dans le fork. Cette fonction permet aussi de récupérer la valeur retournée par la fonction `ma_fonction_thread` du thread. Le prototype de la fonction `pthread_join` est le suivant :

```
int pthread_join(pthread_t thread, void **retour);
```

Le premier paramètre est l'dentifiant du thread (que l'on obtient dans `pthread_create`), et le second paramètre est un `passage par adresse d'un pointeur` qui permet de récupérer la valeur retournée par `ma_fonction_thread`.

5.2.3 Exemples

Le premier exemple crée un thread qui dort un nombre de secondes passé en argument, pendant que le thread principal attend qu'il se termine.

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

void *ma_fonction_thread(void *arg)
{
  int nbsec = (int)arg;
  printf("Je suis un thread et j'attends %d secondes\(\backslash\)n", nbsec);
  sleep(nbsec);
  puts("Je suis un thread et je me termine");
  pthread_exit(NULL);  {/* termine le thread proprement */
}

int main(void)
{
  int ret;
  pthread_t my_thread;
  int nbsec;
  time_t t1;
  srand(time(NULL));
  t1 = time(NULL);
  nbsec = rand()%10; {/* on attend entre 0 et 9 secondes */
  {/* on cr e le thread */
  ret = pthread_create(&my_thread, NULL,
                       ma_fonction_thread, (void*)nbsec);
  if (ret != 0)
    {
```

```
      fprintf(stderr, "Erreur de cr ation du thread");
      exit (1);
    }
  pthread_join(my_thread, NULL); {/* on attend la fin du thread */
  printf("Dans le main, nbsec = %d\(\backslash\)n", nbsec);
  printf("Duree de l'operation = %d\(\backslash\)n", time(NULL)-t1);
  return 0;
}
```

Le deuxième exemple crée un thread qui lit une valeur entière et la retourne au `main`.

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

void *ma_fonction_thread(void *arg)
{
  int resultat;
  printf("Je suis un thread. Veuillez entrer un entier\(\backslash\)n");
  scanf("%d", &resultat);
  pthread_exit((void*)resultat);  {/* termine le thread proprement */
}

int main(void)
{
  int ret;
  pthread_t my_thread;
  {/* on cr e le thread */
  ret = pthread_create(&my_thread, NULL,
                       ma_fonction_thread, (void*)NULL);
  if (ret != 0)
    {
      fprintf(stderr, "Erreur de cr ation du thread");
      exit (1);
    }
  pthread_join(my_thread, (void*)&ret); {/* on attend la fin du thread */
  printf("Dans le main, ret = %d\(\backslash\)n", ret);
  return 0;
}
```

5.3 Donnée partagées et exclusion mutuelle

Lorsqu'un nouveau processus est créé par un `fork`, toutes les données (variables globales, variables locales, mémoire allouée dynamiquement), sont dupliquées et copiées, et le processus père et le processus fils travaillent ensuite sur des variables différentes.

Dans le cas de threads, la mémoire est *partagée*, c'est à dire que les variables globales sont partagées entre les différents threads qui s'exécutent en parallèle. Cela pose des problèmes lorsque deux threads différents essaient d'écrire et de lire une même donnée.

Deux types de problèmes peuvent se poser :

- Deux threads concurrents essaient en même temps de modifier une variable globale ;
- Un thread modifie une structure de donnée tandis qu'un autre thread essaie de la lire. Il est alors possible que le thread lecteur lise la structure alors que le thread écrivain a écrit la donnée à moitié. La donnée est alors incohérente.

Pour accéder à des données globales, il faut donc avoir recours à un mécanisme d'exclusion mutuelle, qui fait que les threads ne peuvent pas accéder en même temps à une donnée. Pour celà, on introduit des données appelés *mutex*, de type `pthread_mutex_t`.

Un thread peut verrouiller un `mutex`, avec la fonction `pthread_mutex_lock()`, pour pouvoir accéder à une donnée globale ou à un flot (par exemple pour écrire sur la sortie `stdout`). Une fois l'accès terminé, le thread dévérouille le mutex, avec la fonction `pthread_mutex_unlock()`. Si un thread A essaie de vérouiller le un mutex alors qu'il est déjà verrouillé par un autre thread B, le thread A reste bloqué sur l'appel de `pthread_mutex_lock()` jusqu'à ce que le thread B dévérouille le mutex. Une fois le mutex dévérouillé par B, le thread A verrouille immédiatement le mutex et son exécution se poursuit. Cela permet au thread B d'accéder tranquillement à des variables globales pendant que le thread A attend pour accéder aux mêmes variables.

Pour déclarer et initialiser un mutex, on le déclare en variable globale (pour qu'il soit accessible à tous les threads) :

```
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
```

La fonction `pthread_mutex_lock()`, qui permet de verrouiller un mutex, a pour prototype :

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

Il faut éviter de verrouiller deux fois un même mutex dans le même thread sans le déverrouiller entre temps. Il y a un risque de blocage définitif du thread. Certaines versions du système gèrent ce problème mais leur comportement n'est pas portable.

La fonction `pthread_mutex_unlock()`, qui permet de déverrouiller un mutex, a pour prototype :

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

Dans l'exemple suivant, différents threads font un travail d'une durée aléatoire. Ce travail est fait alors qu'un mutex est verrouillé.

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

```
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;

void* ma_fonction_thread(void *arg);

int main(void)
{
  int i;
  pthread_t thread[10];
  srand(time(NULL));

  for (i=0 ; i<10 ; i++)
      pthread_create(&thread[i], NULL, ma_fonction_thread, (void*)i);

  for (i=0 ; i<10 ; i++)
      pthread_join(thread[i], NULL);
  return 0;
}

void* ma_fonction_thread(void *arg)
{
  int num_thread = (int)arg;
  int nombre_iterations, i, j, k, n;
  nombre_iterations = rand()%8;
  for (i=0 ; i<nombre_iterations ; i++)
    {
      n = rand()%10000;
      pthread_mutex_lock(&my_mutex);
    printf("Le thread num ro %d commence son calcul\(\backslash\)n", num_thread);
      for (j=0 ; j<n ; j++)
        for (k=0 ; k<n ; k++)
          {}
     printf("Le thread numero %d a fini son calcul\(\backslash\)n", num_thread);
      pthread_mutex_unlock(&my_mutex);
    }
  pthread_exit(NULL);
}
```

Voici un extrait de la sortie du programme. On voit qu'un thread peut travailler tranquillement sans que les autres n'écrivent.

```
...
Le thread numéro 9 commence son calcul
Le thread numero 9 a fini son calcul
Le thread numéro 4 commence son calcul
Le thread numero 4 a fini son calcul
Le thread numéro 1 commence son calcul
```

```
Le thread numero 1 a fini son calcul
Le thread numéro 7 commence son calcul
Le thread numero 7 a fini son calcul
Le thread numéro 1 commence son calcul
Le thread numero 1 a fini son calcul
Le thread numéro 1 commence son calcul
Le thread numero 1 a fini son calcul
Le thread numéro 9 commence son calcul
Le thread numero 9 a fini son calcul
Le thread numéro 4 commence son calcul
Le thread numero 4 a fini son calcul
...
```

En mettant en commentaire les lignes avec `pthread_mutex_lock()` et `pthread_mutex_unlock()`, on obtient :

```
...
Le thread numéro 9 commence son calcul
Le thread numero 0 a fini son calcul
Le thread numéro 0 commence son calcul
Le thread numero 1 a fini son calcul
Le thread numéro 1 commence son calcul
Le thread numero 4 a fini son calcul
Le thread numero 8 a fini son calcul
Le thread numéro 8 commence son calcul
Le thread numero 8 a fini son calcul
Le thread numéro 8 commence son calcul
Le thread numero 1 a fini son calcul
Le thread numéro 1 commence son calcul
Le thread numero 3 a fini son calcul
Le thread numéro 3 commence son calcul
Le thread numero 3 a fini son calcul
Le thread numéro 3 commence son calcul
Le thread numero 5 a fini son calcul
Le thread numero 9 a fini son calcul
...
```

On voit que plusieurs threads interviennent pendant le calcul du thread numéro 9 et 4.

5.4 Sémaphores

En général, une section critique est une partie du code où un processus ou un thread ne peut rentrer qu'à une certaine condition. Lorsque le processus (ou un thread) entre dans la section critique, il modifie la condition pour les autres processus/threads.

Par exemple, si une section du code ne doit pas être exécutée simultanément par plus de n threads. Avant de rentrer dans la section critique, un thread doit vérifier qu'au plus n-1 threads y sont déjà. Lorsqu'un thread entre dans la section critique, il modifie la conditions

sur le nombre de threads qui se trouvent dans la section critique. Ainsi, un autre thread peut se trouver empêché d'entrer dans la section critique.

La difficulté est qu'on ne peut pas utiliser une simple variable comme compteur. En effet, si le test sur le nombre de thread et la modification du nombre de threads lors de l'entrée dans la section critique se font séquentiellement par deux instructions, si l'on joue de malchance un autre thread pourrait tester le condition sur le nombre de threads justement entre l'exécution de ces deux instructions, et deux threads passeraient en même temps dans la section critiques. Il y a donc nécessité de tester et modifier la condition de manière atomique, c'est à dire qu'aucun autre processus/thread de peut rien exécuter entre le test et la modification. C'est une opération atomique appelée *Test and Set Lock*.

Les sémaphores sont un type `sem_t` et une ensemble de primitives de base qui permettent d'implémenter des conditions assez générales sur les sections critiques. Un sémaphore possède un compteur dont la valeur est un entier positif ou nul. On entre dans une section critique si la valeur du compteur est strictement positive.

Pour utiliser une sémaphore, on doit le déclarer et l'initialiser à une certaine valeur avec la fonction `sem_init`.

```
\inte sem_init(sem_t *semaphore, \inte partage, {\bf unsigned} \inte valeur)
```

Le premier argument est un passage par adresse du sémaphore, le deuxième argument indique si le sémaphore peut être partagé par plusieurs processus, ou seulement par les threads du processus appelant (`partage` égale 0). Enfin, le troisième argument est la valeur initiale du sémaphore.

Après utilisation, il faut systématiquement libérer le sémaphore avec la fonction `sem_destroy`.

```
int sem_destroy (sem_t *semaphore)
```

Les primitives de bases sur les sémaphores sont :

- `sem_wait` : Reste bloquée si le sémaphore est nul et sinon décrémente le compteur (opération atomique) ;
- `sem_post` : incrémente le compteur ;
- `sem_getvalue` : récupère la valeur du compteur dans une variable passée par adresse ;
- `sem_trywait` : teste si le sémaphore est non nul et décrémente le sémaphore, mais sans bloquer. Provoque une erreur en cas de valeur nulle du sémaphore. Il faut utiliser cette fonction avec précaution car elle est prompte à générer des bogues.

Les prototypes des fonctions `sem_wait`, `sem_post` et `sem_getvalue` sont :

```
\inte sem_wait (sem_t * semaphore)
\inte sem_post(sem_t *semaphore)
\inte sem_getvalue(sem_t *semaphore, \inte *valeur)
```

Exemple. Le programme suivant permet au plus n sémaphores dans la section critique, où n en passé en argument.

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

sem_t semaphore; {/* variable globale : s maphore */

void* ma_fonction_thread(void *arg);

int main(int argc, char **argv)
{
  int i;
  pthread_t thread[10];
  srand(time(NULL));

  if (argc != 2)
    {
      printf("Usage : %s nbthreadmax\(\backslash\)n", argv[0]);
      exit(0);
    }
  sem_init(&semaphore, 0, atoi(argv[1]));  {/* initialisation */

  {/* cr ation des threads */
  for (i=0 ; i<10 ; i++)
      pthread_create(&thread[i], NULL, ma_fonction_thread, (void*)i);

  {/* attente */
  for (i=0 ; i<10 ; i++)
      pthread_join(thread[i], NULL);
  sem_destroy(&semaphore);
  return 0;
}

void* ma_fonction_thread(void *arg)
{
  int num_thread = (int)arg;  {/* num ro du thread */
  int nombre_iterations, i, j, k, n;
  nombre_iterations = rand()%8+1;
  for (i=0 ; i<nombre_iterations ; i++)
    {
      sem_wait(&semaphore);
      printf("Le thread %d entre dans la section critique\(\backslash\)n",
              num_thread);
      sleep(rand()%9+1);
      printf("Le thread %d sort de la section critique\(\backslash\)n",
              num_thread);
```

```
      sem_post(&semaphore);
      sleep(rand()%9+1);
    }
  pthread_exit(NULL);
}
```

Exemples de trace :

```
\ gcc -lpthread semaphore.c -o semaphore
\  ./semaphore 2
Le thread 0 entre dans la section critique
Le thread 1 entre dans la section critique
Le thread 0 sort de la section critique
Le thread 2 entre dans la section critique
Le thread 1 sort de la section critique
Le thread 3 entre dans la section critique
Le thread 2 sort de la section critique
Le thread 4 entre dans la section critique
Le thread 3 sort de la section critique
Le thread 5 entre dans la section critique
Le thread 4 sort de la section critique
Le thread 6 entre dans la section critique
Le thread 6 sort de la section critique
Le thread 7 entre dans la section critique
Le thread 7 sort de la section critique
Le thread 8 entre dans la section critique
...
```

Autre exemple avec trois threads dans la section critique :

```
\ ./semaphore 3
Le thread 0 entre dans la section critique
Le thread 1 entre dans la section critique
Le thread 2 entre dans la section critique
Le thread 1 sort de la section critique
Le thread 3 entre dans la section critique
Le thread 0 sort de la section critique
Le thread 4 entre dans la section critique
Le thread 2 sort de la section critique
Le thread 5 entre dans la section critique
Le thread 3 sort de la section critique
Le thread 6 entre dans la section critique
Le thread 6 sort de la section critique
Le thread 7 entre dans la section critique
Le thread 5 sort de la section critique
Le thread 8 entre dans la section critique
```