TP1: Threads POSIX

Programmation multi-threadée

★ Bibliothèque pthread

Prototypes des fonctions principales

```
pthread_t // data type to identify a thread
pthread_mutex_t // data type for a mutex

int pthread_create (pthread_t* thread, const pthread_attr_t* attr, void* (*start_routine)(void*), void* arg);
// Exemple: pthread_create(&monID, NULL, fonction, NULL);

int pthread_join (pthread_t thread, void** retval); // Le thread appelant attend la fin du thread en paramètre
int pthread_detach (pthread_t thread_cible);        // Le thread cible ne peut plus être joint

int pthread_mutex_init (pthread_mutex_t* mutex, const pthread_mutexattr_t* attr);
// Exemple 1: pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
// Exemple 2: pthread_mutex_t lock2;
//            pthread_mutex_init(&lock2);
int pthread_mutex_lock (pthread_mutex_t* mutex);
int pthread_mutex_unlock (pthread_mutex_t* mutex);
int pthread_mutex_destroy (pthread_mutex_t* mutex);
```

★ Exercice 1: Comprendre du code avec des threads

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

pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
long long count = 0;

void increment_count() {
  pthread_mutex_lock(&count_mutex);
  count = count + 1;
  pthread_mutex_unlock(&count_mutex);
}

long long get_count() {
  pthread_mutex_lock(&count_mutex);
  long long c = count;
  pthread_mutex_unlock(&count_mutex);
  return (c);
}
void* thread_funA(void* ignored) {
  int i;
  while ((i = get_count()) < 50000) {
    if (i%10000==0) printf("A: i = %d\n", i);
    increment_count();
  }
  return NULL;
}
void *thread_funB(void *ignored) {
  int i;
  while ((i = get_count()) < 100000) {
    if (i % 10000 == 0) printf("B: i = %d\n", i);
    increment_count();
  }
  return NULL;
}
int main(int argc, char **argv) {
  pthread_t pthA, pthB;
  pthread_create(&pthA, NULL, thread_funA, NULL);
  pthread_create(&pthB, NULL, thread_funB, NULL);
  pthread_join(pthA, NULL);
  pthread_join(pthB, NULL);
}
```

Le programme ci-contre crée deux threads dont l'objectif est d'incrémenter un compteur partagé jusqu'à atteindre la valeur cible. Les accès au compteur sont protégés par un mutex, c'est-à-dire un verrou d'exécution exclusive. Quand l'un des threads passe une ligne `pthread_mutex_lock()`, il devient propriétaire du verrou. L'autre thread sera bloqué s'il tente aussi d'acquérir le verrou, jusqu'à ce que le propriétaire relâche le verrou avec la fonction `pthread_mutex_unlock()`.

Mais le programme a (parfois) l'affichage ci-dessous. Les deux threads voient parfois les mêmes valeurs malgré le mécanisme de verrouillage.

```
A: i = 0
B: i = 0
B: i = 10000
B: i = 20000
A: i = 30000
B: i = 30000
A: i = 40000
B: i = 40000
B: i = 50000
B: i = 60000
B: i = 70000
B: i = 80000
B: i = 90000
```

▷ **Question 1:** Que c'est-il passé?

▷ **Question 2:** Comment corriger le problème ?

★ Exercice 2: Écrire de petits programmes avec des threads

▷ **Question 1:** *Création et attente de threads*

Écrivez un programme ayant le comportement suivant :

1. Des threads sont créés (leur nombre étant passé en paramètre lors du lancement du programme) ;
2. Chaque thread affiche un message (par example "hello world!") ;
3. Le thread "principal" attend la terminaison des différents threads créés.

▷ **Question 2:** *Identification des threads*

Modifiez le programme de la question précédente pour que chaque thread affiche :

- son PID (avec `getpid()`) ;
- La valeur opaque retournée par `pthread_self`, par exemple avec :
  `printf("%p\n", (void *) pthread_self());`

▷ **Question 3:** *Passage de paramètres.*

Modifiez le programme de la question précédente pour passer son numéro d'ordre à chaque thread. Chaque thread doit ensuite l'afficher. Vérifiez que le numéro d'ordre affiché par chaque thread est bien différent (corrigez votre programme le cas échéant).

▷ **Question 4:** *Exclusion mutuelle.*

Déclarez une variable globale `somme` initialisée à 0. Chaque thread doit, dans une boucle de 1 000 000 itérations, ajouter son numéro d'ordre à cette variable globale. Le thread principal doit lancer tous les threads, puis attendre la terminaison de chacun avant d'afficher la valeur finale de `somme` (vos threads doivent s'exécuter de façon concurrente et non les uns après les autres).

Avec 5 threads (numérotés de 0 à 4), on devrait obtenir (0+1+2+3+4)*1 000 000 = 10 000 000. Corrigez votre programme s'il n'affiche pas systématiquement ce résultat.

▷ **Question 5:** *Attendre des résultats des threads.*

Le problème de la solution précédente est le nombre d'opérations de synchronisation, bien trop important. On pourrait être tenté de faire un seul gros verrouillage pour protéger l'intégralité de la boucle dans chaque thread. Mais cela reviendrait à exécuter les threads les uns après les autres, sans parallélisme.

Nous allons maintenant mettre en place une version parallèle, mais avec moins d'interaction entre les threads pour une meilleur efficacité. Pour cela, chaque thread devra incrémenter 1 000 000 fois une variable locale en ajoutant son identifiant à chaque fois, puis retourner ce résultat dans le `pthread_exit()`. Le code correspondant est esquissé à droite. Il serait certes possible de faire une multiplication au lieu de 1 000 000 additions, mais ce n'est pas la question.

```
void un_thread(void *id) {
  int var = 0;
  for (int i=0; i< 1000000; i++) {
    var += *id;
  }
  /* Ajouter ici l'appel à pthread_exit() nécessaire */
}
```

Votre fonction `main()` doit lancer tous les threads, récupérer les valeurs retournées par chaque thread avec `pthread_join()`, puis additionner ces valeurs. Cette forme d'organisation du code s'appelle un *fork-join.*
Indication: l'adresse retournée par `pthread_exit()` doit pointer vers une zone mémoire stable.

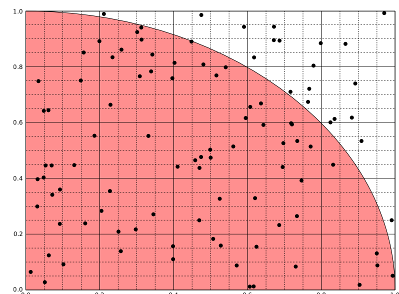
Corrigez votre programme s'il n'affiche pas le résultat attendu, et utilisez valgrind pour vérifier qu'il est correct. Vous utiliserez le réglage par défaut de valgrind pour vérifier les accès mémoire `valgrind ./monprog` ainsi que l'outil helgrind pour vérifier les synchronisations `valgrind --tool=helgrind ./monprog`.

▷ **Question 6:** *Paramètres complexes (optionnelle).*

Modifiez votre programme pour ne plus utiliser de variables globales. Il faut donc passer les adresses des différentes variables nécessaires en argument aux threads. On utilisera pour cela une structure comme suit:

```
struct thread_args {
  int i;
  int * somme;
  pthread_mutex_t * mutex;
};
```

★ Exercice 3: Approximation de π (à ne pas faire en 2023).



L'objectif de cet exercice est d'approximer la valeur de π avec la méthode de monte-carlo. Il s'agit de tirer des points aléatoires dans le carré unité, et calculer la proportion d'entre situé sur le disque centré sur l'origine et de rayon 1. La proportion de points sur le disque par rapport au nombre de points généré est égale à l'aire du quart de disque, c'est-à-dire à $\frac{\pi}{4}$. L'illustration ci-contre est issue de la page Wikipedia correspondante (CC-BY-SA).

▷ **Question 1:** Écrivez une fonction tirant 1 000 000 points aléatoires sur le carré unité, avant d'afficher la proportion d'entre eux placés sur le disque unité.

▷ **Question 2:** Écrivez un programme multi-threadé démarrant autant de threads qu'il y a de coeurs sur la machine (cf. `get_nproc()`). Chaque thread répétera à l'infini le code suivant:

- générer 1 000 000 points aléatoires et compter combien sont placés sur le disque unité.
- metttre à jour deux variables globales protégées par mutex. L'une reprendra la quantité totale de points générées par l'ensemble des threads tandis que l'autre servira à compter le nombre de ces points placés sur le disque unité.

De son coté, le programme principal affichera toutes les 10 secondes une approximation de π calculé à partir des deux variables globales.

★ Exercice 4: Communication inter-threads (à ne pas faire)

Observez le programme `distributeur.c` fourni. Son action est la suivante:

- lit le flux de caractères arrivant sur l'entrée standard en séparant les chiffres et les lettres
- effectue l'opération appropriée en fonction du type de caractère (sommer les chiffres; réaliser un spectre de fréquence pour les lettres)
- affiche le résultat obtenu

Pour cela, le programme est composé de trois entités : un distributeur (en charge de la répartition des caractères) ; un additionneur (opérant sur les chiffres) ; un compteur (opérant sur les lettres).

▷ **Question 1:** Adaptez ce programme pour que les fonctions d'additionneur, de compteur et de distributeur soient assurées par des threads séparés communiquant à travers des tableaux partagés en mémoire.