TP 3 : Threads, synchronisation, concurrence non bloquante

On va voir comment utiliser les verrous et les variables de condition en C pour réaliser des programmes multi-threads sûrs, et quelques fois efficaces.

On va utiliser la librairie `pthread` dont toutes les fonctions commencent par `pthread_` et sont documentées dans les pages `man` ou sur le web, par exemple ici.

Exercice 1 : Echauffement

Récupérer l'archive contenant les fichiers à compléter ici. Ouvrez le fichier `echauffement.c` Vous y trouverez un programme qui lance deux threads qui chacun incremente $N = 100000$ fois de suite une variable globale `c` .

Question 1

Vérifier que vous comprenez bien les différentes étapes pour lancer et terminer les threads. Compiler avec l'option `-pthread` et vérifier que la valeur du compteur affichée à la fin n'est pas le double de N.

Question 2

Déclarer un verrou (en variable globale) et compléter les fonctions : `enter_critical` , `exit_critical` , `init_mutexes` et `destroy_mutexes` .

Vous aurez besoin de choses de la forme `pthread_mutex_...` ; on utilisera toujours les mutex non réentrants et on initialisera le mutex `mut` en faisant `pthread_mutex_init(&mut,NULL);` .

Compilez et vérifiez que le compteur atteint désormais la valeur souhaitée.

Exercice 2 : A table

On va maintenant s'intéresser à la situation où les deux threads se partagent une table à $M = 10$ entrées. On pourra imaginer que cette table est par exemple une base de données, ou une des nombreuses tables utilisées au niveau système, ou quelque chose d'autre de bien utile.

Ouvrez le fichier `atable.c` Pour faire simple, nous avons supposé que c'est une table de compteurs déclarée dans une variable globale.

```
int c[M];
```

Le travail palpitant de nos deux petits threads consiste maintenant à échanger N fois de suites deux entrées `i` et `j` de la table tirées au hasard.

```
void atomic_swap(int i, int j){

  enter_critical(i, j);

  int t = c[i];
  c[i] = c[j];
  c[j] = t;

  exit_critical(i, j);

}
```

Dans de nombreux cas, les deux threads travailleront sur des parties disjointes de la table, et tout se passera bien, donc il serait dommage de mettre un verrou qui protège toute la table. On va donc chercher d'autres solutions.

Question 1

Déclarez une table de verrous de taille $M = 10$ et complétez les quatre fonctions qui gèrent ces verrous et les entrées et sorties de section critique. Vérifiez que toutes les valeurs entre 0 et $M - 1$ sont bien dans la table à la fin du programme.

Nos deux threads ont maintenant une nouvelle lubie : faire des permutations circulaires sur l'ensemble des entrées comprises entre `i` et `j` inclus (cf. fichier `atable2.c`

```
void atomic_perm(int i, int j, int tid){

  if (i>j) {
    int t = i;
    i = j;
    j = t;
  }

  enter_critical(i,j,tid);

  int n;
  int fst;
  fst = c[i];
  for(n=i; n<j; n++) c[n] = c[n+1];
  c[j] = fst;

  exit_critical(i,j,tid);

}
```

On décide d'abandonner l'idée d'un verrou par entrée dans la table. A la place, on va utiliser une variable globale `use` qui permettra à chaque thread d'informer l'autre thread de l'intervalle d'entrées sur lequel il souhaite travailler.

```
int use[2][2]
```

Par exemple, si le thread d'identifiant `tid` $\in \{0, 1\}$ travaille sur les entrées entre 5 et 7, on aura `use[tid][0]` égal à 5 et `use[tid][1]` égal à 7.

Question 2

Complétez les fonctions d'entrées et sorties de section critique en protégeant au besoin la variable `use` *par un verrou. Si il n'est pas possible de rentrer en section critique, le thread fera de l'attente active (spin locking).*

Exercice 3 : Des nains et des géants

Nos threads ont encore changé de dada. Cette fois-ci, ils sont plus nombreux, et ils se divisent en deux catégories :

— les nains, qui font des modifications simultanées sur deux entrées de la table ;

— les géants, qui permutent circulairement toute la table.

Pour que tout se beau monde ne se marche pas sur les pieds, on va à nouveau utiliser des verrous : à la fois un verrou global sur toute la table, mais aussi des verrous locaux pour chaque entrée de la table (si les nains peuvent travailler en parallèle, autant ne pas les faire attendre). On aura donc une fonction `enter_critical_local(int i, int j)` pour les entrées en section critiques de nains, et une fonction `enter_critical_global()` pour les géants, et de manière symmétrique des fonctions distinctes pour les sorties de section critique.

Question 1

Complétez le fichier `hierarchical_mutex.c`*. Pour éviter l'attente active, vous aurez peut-être envie d'utiliser des variables de condition.*

Pour vous aider, voici ci-après un petit exemple d'utilisation d'une variable de condition. Notez qu'on utilise ici `pthread_cond_broadcast` afin de réveiller (une fois, et tour à tour) tous les threads qui ont fait un wait avant ce broadcast. Quand le thread est réveillé, il possède le verrou associé à la variable de condition, et quand il s'endort, il relache ce verrou. Mais pourquoi a-t-on cette boucle while autour du `wait` ?

```
pthread_mutex_t rsrc_lock;
pthread_cond_t rsrc_add;
unsigned int resources;

get_resources(int amount)
{
    pthread_mutex_lock(&rsrc_lock);
    while (resources < amount) {
        pthread_cond_wait(&rsrc_add, &rsrc_lock);
    }
    resources -= amount;
    pthread_mutex_unlock(&rsrc_lock);
}

add_resources(int amount)
{
    pthread_mutex_lock(&rsrc_lock);
    resources += amount;
    pthread_cond_broadcast(&rsrc_add);
    pthread_mutex_unlock(&rsrc_lock);
}
```