TD nº 6 - Processus légers

Exercice 1. *Premier exemple*

1 En vous insipirant de l'exemple vu en cours (les transparents sont disponibles sur l'ENT), écrivez un programme qui démarre trois threads qui affichent « `Bonjour\n` ».

Indication : Il faut penser à mettre les bonnes options au moment de la compilation.

2 Modifiez le programme de la question précédent pour que chaque thread reçoive en argument un numéro unique (0, 1 et 2) et qu'il affiche son numéro après le message (« `Bonjour (1)\n` » pour le thread numéro 1).

Indication : La fonction exécutée par un thread ne peut recevoir qu'un seul argument qui est nécessairement de type `(void *)`. Il faut dont lui passer l'adresse d'un entier et récupérer cet argument dans une variable de type `(int *)` dans la fonction.

Attention à bien créer un indice différent pour chaque thread, et non pas passer l'adresse de la même variable à chacun (il faut faire un tableau contenant les indices à passer à chaque thread et passer l'adresse de la case du tableau correspondant à chaque thread).

Exercice 2. *Synchronisation de threads*

On considère le programme suivant :

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

void *start_routine1 (void * arg) {
  printf ("Début du thread 1\n");
  sleep (1);
  printf ("Fin du thread 1\n");
  pthread_exit (0);
}

void *start_routine2 (void * arg) {
  printf ("Début du thread 2\n");
  sleep (2);
  printf ("Fin du thread 2\n");
  pthread_exit (0);
}

int main (int argc, char **argv) {
  pthread_t th1, th2;
  void *ret;
  if (pthread_create (&th1, NULL, start_routine1, NULL) < 0) {
    perror("");
    exit (1);
  }
  if (pthread_create (&th2, NULL, start_routine2, NULL) < 0) {
    perror("");
    exit (1);
  }
  (void)pthread_join (th1, &ret);
  (void)pthread_join (th2, &ret);
  return 0;
}
```

1 Recopiez, compilez et exécutez le programme. Interprétez le résultat observé.

On va maintenant utiliser un mécanisme pour contrôler l'ordre d'exécution des threads en utilisant des sémaphores (que vous verrez plus en détail l'an prochain) :

```
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h> // nouvelle bibliothèque
static sem_t semaphore; // déclaration d'une variable globale


void *start_routine1 (void * arg) {
  printf ("Début du thread 1\n");
  sleep (1);
  sem_wait(&semaphore); // attendre le sémaphore
  printf ("Fin du thread 1\n");
  pthread_exit (0);
}

void *start_routine2 (void * arg) {
  printf ("Début du thread 2\n");
  sleep (2);
  printf ("Fin du thread 2\n");
  sem_post(&semaphore); // libérer le sémaphore
  pthread_exit (0);
}

int main (int argc, char **argv) {
  pthread_t th1, th2;
  sem_init(&semaphore, 0, 0); // initialiser le sémaphore (pas libre)

  if (pthread_create (&th1, NULL, start_routine1, NULL) < 0) {
    perror("");
    exit (1);
  }
  if (pthread_create (&th2, NULL, start_routine2, NULL) < 0) {
    perror("");
    exit (1);
  }
  (void)pthread_join (th1, NULL);
  (void)pthread_join (th2, NULL);
  return 0;
}
```

2 Exécutez cette nouvelle version. Essayez de comprendre l'utilisation de la variable `semaphore`.

3 Écrivez un programme qui lance 10 threads numérotés de 0 à 9 en respectant les contraintes suivantes :

— chaque thread affiche son propre numéro et se termine ;
— les threads sont créés dans l'ordre croissant ;
— la séquence des numéros affichés par les threads doit être dans l'ordre décroissant (le thread 9 affiche son numéro en premier, le thread 0 en dernier) ;
— il ne faut pas ralentir les threads à l'aide de la fonction `sleep`.

Indication : Il faut préparer un tableau de threads et un tableau de sémaphores. Chaque sémaphore s'occupe de passer la main d'un thread au suivant. Ainsi, chaque thread doit connaître son numéro, attendre que le sémaphore correspondant au thread avant lui son numéro soit disponible avant d'afficher son numéro, puis libérer le sémaphore suivant.