Théorie et pratique de la concurrence – Master 1 Informatique
TP 1 : Threads en C

Un *thread* est un processus léger, c'est-à-dire un processus qui a ses propres compteur programme et pile d'exécution, mais aucune des données "lourdes" associées en général à un processus (par exemple les tables des pages mémoire).
L'interface de la bibliothèque `Pthreads` a été proposée dans les années 1990 comme un standard POSIX (d'où le préfixe P) pour les routines de programmation multi-*thread* en C. Différentes implémentations de ce standard sont actuellement disponibles sur les systèmes de type Unix. Cette bibliothèque contient une douzaine de fonctions pour le contrôle des *threads* et pour la synchronisation.

Création et terminaison de *threads*

Pour utiliser la bibliothèque `Pthreads` dans un programme C il faut, premièrement, inclure son fichier de déclarations standard :

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

Deuxièmement, il faut déclarer un ou plusieurs descripteurs de *threads* comme suit :

```
pthread_t pid, cid;   /* descripteurs de thread */
```

Finalement, on crée les *threads* comme suit :

```
pthread_create(&pid, NULL, start_func, arg);
```

Le premier argument est l'adresse d'un descripteur de *thread*, le deuxième argument (que nous mettrons toujours à `NULL` dans ces TP) est un attribut de thread de type `pthread_attr_t *` (ces attributs servent par exemple à définir la taille de la pile ou encore à imposer une politique d'ordonnancement), le troisième argument est le nom de la fonction à exécuter au lancement du thread, il doit être de type `void *start_func(void *arg)` et le quatrième argument correspond à l'argument passé à la fonction `start_func` au moment de l'appel, il doit être de type `void *` (cet argument peut aussi être `NULL`). Si la création est effectuée correctement, la fonction renvoie 0, sinon un code d'erreur. Au moment de l'appel à la fonction `pthread_create`, le *thread* correspondant est lancé et peut exécuter la fonction `start_func`.

Un *thread* se termine en appelant :

```
pthread_exit(value);
```

où l'argument est de type `void *` et peut être égal à `NULL`.

Un processus parent peut attendre la terminaison d'un fils en effectuant :

```
pthread_join(pid, value_ptr);
```

où le deuxième argument est de type `void **` et correspond à une adresse pour le stockage de la valeur de retour du *thread* (avec `pthread_exit`).

Afin d'observer le comportement d'entrelacement des threads, vous pouvez les faire dormir un certain temps grâce à la fonction `unsigned int sleep(unsigned int seconds)`.

Voilà un exemple de programmes C avec des threads :

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

void *print_message1(void *ptr);
void *print_message2(void *ptr);

int main(){
  pthread_t th1, th2;
  const char* s1="I am thread 1";
  const char* s2="I am thread 2";

  int r1=pthread_create(&th1,NULL,print_message1,(void *)s1);
  int r2=pthread_create(&th2,NULL,print_message2,(void *)s2);

  pthread_join(th1,NULL);
  pthread_join(th2,NULL);
  return 0;
}

void *print_message1(void *ptr){
  char *mess=(char *)ptr;
  sleep(2);
  printf("%s \n",mess);
  return NULL;
}

void *print_message2(void *ptr){
  char *mess=(char *)ptr;
  printf("%s \n",mess);
  return NULL;
}
```

Rappel : Pour compiler un programme utilisant des threads POSIX, vous devez utiliser l'option de compilation `-pthread`

Exercices

Exercice 1: *Un premier essai*

1. Reprenez le programme ci-dessus et exécutez le.
2. Ajoutez un thread à ce programme qui affichera `I am thread 3`.
3. Pensez à une solution compacte, pour exécuter n threads (l'entier n étant donné en argument au moment du lancement du programme), chaque thread `i` devra afficher le message `I am thread i` (sans dormir avant).
4. Prenez le code précédent et faîtes en sorte maintenant que chacun des n threads dorme de façon aléatoire entre 0 et 5 secondes (vous pouvez utilisez la fonction `int rand()` pour cela).

Exercice 2: *Différences entre processus lourds et threads*

On considère le programme ci-dessous.

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

int main(){
  int x=0;
  int pid = fork();
  printf("Valeur de fork = %d ",pid);
  printf("processus %d de pere %d\n", (int) getpid(), (int)%getppid());
  if ( pid == 0 ) {
    x=25;
    printf("Valeur de x pour le fils : %d\n",x);
    printf("fin du processus fils\n");
  }
  else {
    while(x!=25){
      sleep(2);
      printf("Valeur de x pour le pere : %d\n",x);
    }
    printf("fin du processus pere\n");
  }
  return 0;
}
```

1. Exécutez le programme précédent. Qu'observez-vous ? Pouvez-vous justifier ce comportement ?
2. Reprenez le programme en utilisant des threads là où des processus lourds sont utilisés. Faites attention à la façon dont vous transmettez la variable `x` aux threads. Qu'observez-vous ?

Exercice 3: *Observation d'entrelacement*

1. Programmez deux threads qui affichent de façon indépendante sur chaque ligne un identifiant et les numéros de 1 à 100.
2. Modifiez le programme précédent pour faire dormir chaque thread un nombre aléatoire de seconde entre 0 et 2 secondes avant l'écriture de chaque ligne.
3. Modifiez maintenant le programme pour que ces deux threads partagent une variable et chacun incrémente la variable puis affiche la valeur en boucle jusqu'à ce que la valeur de la variable (initialisée à 0) soit 200. Observez différents comportements.

Exercice 4: *Problèmes lié à la mémoire partagée*

On souhaite programmer un système composé d'un producteur et de plusieurs consommateurs (chaque entité correspondant à un thread). Les variables partagées sont une variable entière `flag`, correspondant à un booléen, valant `0` (pour faux) et `1` (pour vrai), et initialisée à `0`, et, une autre variable entière `counter` représentant un compteur initialisé à `0`. Le producteur réalise en boucle les actions suivantes :
– il vérifie que `flag` est faux,
– il incrémente `counter`,
– il met le booléen `flag` à vrai.
Chaque consommateur a une identité distincte (par exemple un entier) et suit le comportement suivant en boucle :

– il attend que la variable `flag` soit vraie (cela signifie que le producteur a produit une ressource)
– il affiche son nom et la valeur de `counter`,
– il met la variable `flag` à faux.
On souhaite observer que plusieurs consommateurs lisent la même valeur de la variable (on verra plus tard comment éviter ce genre de comportement) et aussi que certains consommateurs ne lisent jamais la variable. Ces comportements sont en effet possible compte tenu de la politique d'entrelacement. Pour pouvoir observer ces comportements, vous devrez faire dormir certains processus à des moments opportuns. Programmez ce système afin de pouvoir observer les comportements susmentionnés.

Exercice 5: *Observation d'interblocage*

Pour cet exercice, il vous est demandé de programmer deux threads qui ont accès à deux variables entières partagées `flag1` et `flag2` encodant des booléens (la valeur de ces variables est donc `0` ou `1`) et qui respectent les comportements suivants. Le premier thread fait en boucle les actions suivantes :
– il attend que `flag1` soit vraie,
– il met `flag1` à faux,
– il attend que `flag2` soit vraie,
– il met `flag2` à faux,
– il affiche un message caractéristique (permettant de faire la différence entre les deux threads).
Le deuxième thread fait la même chose en inversant l'accès aux variables `flag1` et `flag2`. Trouvez un moyen de montrer une situation d'interblocage dans laquelle au bout d'un moment aucun des deux threads n'affiche son message caractéristique.