INF4033 Lab 5 :
Introduction aux Pthread

Alexandre BRIÈRE
alexandre.briere@esiea.fr

1 Introduction

1.1 Objectifs

- Mettre en pratique l'utilisation de la bibliothèque *Pthread*.
- Résoudre des problèmes de concurrence mémoire.

Pour le parallélisme, vous n'utiliserez dans ce TP que les fonctions présentes dans la bibliothèque *Pthread* (fork, exec, etc. ne sont pas autorisés).

1.2 Pour commencer

Tous les exercices vous demanderons d'écrire des codes sources. Pour les compiler, vous pouvez soit invoquer `gcc` directement en utilisant :

```
$ gcc fichier_source.c -o executable_sortie -lpthread
```

Soit en utilisant les règles implicites de `make`. Pour cela il faut préalablement affecter dans votre environnement les deux variables suivantes :

```
$ export CFLAGS=-lpthread && export CC=gcc
```

1.3 Astuces

Pour connaitre le prototype exact des fonctions de la bibliothèque Pthread, pensez à la commande man avec par exemple :

```
$ man pthread_create
```

Pensez bien à vérifier toutes les valeurs de retour de la bibliothèque Pthread.

2 Prise en main

2.1 Création de tâches

Écrivez le programme suivant : la tâche principale crée N tâches et passe en paramètre à chaque tâche l'ordre de sa création (la première tâche aura le numéro 0, la suivante 1, etc.). Les N tâches affichent le numéro passé en argument puis retournent à la tâche principale cette valeur multipliée par 10. La tâche principale attend la terminaison des N tâches et affiche les N valeurs retournées.

2.2 Cohérence d'une variable

Reprenez le code de la question précédente. Les tâches doivent ajouter la valeur passée en paramètre à une variable globale. La tâche principale, après avoir attendu les N tâches, affiche la valeur de cette variable globale. Vérifiez bien que la valeur affichée est la bonne. N'hésitez pas à augmenter le nombre de tâches pour avoir beaucoup de concurrence.

3 Exercice 2 : Producteur/Consommateur

Dans cet exercice, l'objectif est de faire communiquer deux groupes de tâches : les producteurs et les consommateurs. Les producteurs produisent des valeurs aléatoires et les communiquent aux consommateurs pour qu'ils puissent les afficher. Pour transmettre ces valeurs, les tâches utiliseront une structure partagée (`fifo`). Un producteur soumet une valeur dans cette file grâce à la fonction `put`. Un consommateur retire une valeur grâce à la fonction `get`.

3.1 Première partie

Récupérez le fichier `consoprod.c`, il contient la déclaration de la structure `fifo` ainsi que les fonctions permettant de l'utiliser. Pour faire cette question, vous devez :

- implémenter les fonctions de consommation et de production;
- lancer ces tâches dans la fonction `main` puis attendre leur terminaison.

Attention : certaines variables de la `fifo` sont lues et écrites par plusieurs tâches. Rajoutez les protections nécessaires pour ne pas avoir d'incohérences mémoires. Vous pouvez rajouter des variables dans la structure `fifo`. Si ces variables doivent être initialisées, faites-le dans la fonction `init_fifo`. De plus, vous pouvez utiliser la fonction `sched_yield` pour laisser une autre tâche s'exécuter.

3.2 Deuxième partie

Faites les modifications nécessaires pour exécuter plusieurs consommateurs et producteurs en même temps. Pour cela, modifiez la valeur des constantes `NB_CONSO` et `NB_PROD` (votre code de la question précédente est peut-être déjà suffisant). Vous pouvez rajouter des variables dans la structure `fifo`. Si ces variables doivent être initialisées, faites-le dans la fonction `init_fifo`.

3.3 Optimisation

Faites les changements nécessaires pour qu'une tâche productrice/consommatrice en attente soit notifiée lorsqu'une case de la `fifo` se libère/remplie. Vous pouvez rajouter des variables dans la structure `fifo`. Si ces variables doivent être initialisées, faites-le dans la fonction `init_fifo`.

4 Code source

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

#define SIZE_FIFO 7
#define NB_PROD 1
#define NB_CONSO 1
#define NB_MESSAGE 2000

int min(int a, int b) {
    return a < b ? a : b;
}

// Structure Fifo communication entre consommateurs et producteurs
struct fifo {
    int tab[SIZE_FIFO]; // Tableau d'elements stockes
    int ptr_lecteur;    // Index de la prochaine case a lire
    int ptr_ecrivain;   // Index de la prochaine case a ecrire
    int nb_elem;        // Nombre d'element present dans la fifo
} fifo;

// Fonction d'initialisation de la fifo
void init_fifo(struct fifo *f) {
    f->nb_elem = 0;
    f->ptr_lecteur = 0;
    f->ptr_ecrivain = 0;
}

// Fonction permettant d'inserer une valeur dans la fifo
void put(struct fifo *f, int a) {

    while (f->ptr_lecteur == f->ptr_ecrivain && f->nb_elem != 0){

    }

    f->tab[f->ptr_ecrivain] = a;
    f->ptr_ecrivain = (f->ptr_ecrivain + 1) % SIZE_FIFO;
    f->nb_elem++;
}

// Fonction permettant de recuperer une valeur depuis la fifo
int get(struct fifo *f) {

    int value;

    while (f->nb_elem == 0) {

    }

    value = f->tab[f->ptr_lecteur];
    f->ptr_lecteur = (f->ptr_lecteur + 1) % SIZE_FIFO;
    f->nb_elem--;

    return value;
}
```

```c
// Fonction de production
void *prod(void* arg) {

    /* TO BE COMPLETED */

}

// Fonction de consommation
void *conso(void* arg) {

    /* TO BE COMPLETED */

}

// Fonction principale
int main() {

    /* TO BE COMPLETED */

    return 0;
}
```

5 Mémo : API Pthread

```
///////////
// Types //
///////////

pthread_t
pthread_attr_t
pthread_mutex_t
pthread_cond_t
struct sched_param

////////////////////////
// Gestion des threads //
////////////////////////

// Creation d'un thread
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*function)(void *), void *arg)

// Change le comportement de terminaison d'un thread
int pthread_detach(pthread_t thread)

// Compare l'ID de deux threads
int pthread_equal(pthread_t t1, pthread_t t2)

// Termine le thread appelant
void pthread_exit(void *value_ptr)

// Le thread appelant attend la terminaison du thread t_id
int pthread_join(pthread_t t_id, void **value_ptr)

// Annule l'execution du thread
int pthread_cancel(pthread_t t_id)

// Permet d'executer une UNIQUE fois une fonction
int pthread_once(pthread_once_t *once_control, void(*init_routine)(void))

// Renvoie l'ID du thread appelant
pthread_t pthread_self(void)

// Permet de specifier le comportement avant et apres un fork()
int pthread_atfork(void (*prepare)(void), void (*parent)(void),
                   void(*child)(void))

/////////////////////////
// Gestion des attributs //
/////////////////////////

// Detruit un attribut
int pthread_attr_destroy(pthread_attr_t *attr)

// Renvoie la regle d'heritage de la politique d'ordonnancement
int pthread_attr_getinheritsched(const pthread_attr_t *attr,
                                 int *inheritsched)

// Renvoie les parametres de la politique d'ordonnancement
int pthread_attr_getschedparam(const pthread_attr_t *attr,
                               struct sched_param *param)
```

```c
// Renvoie la politique d'ordonnancement
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy)

// Renvoie la taille de la pile
int pthread_attr_getstacksize(const pthread_attr_t *attr,
                              size_t *stacksize)

// Renvoie l'adresse de la pile
int pthread_attr_getstackaddr(const pthread_attr_t *attr,
                              void **stackaddr)

// Renvoie le comportement du thread en cas de terminaison
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate)

// Initialise les attributs d'un thread avec les valeurs par defaut
int pthread_attr_init(pthread_attr_t *attr)

// Definie les regles d'heritage de la politique d'ordonnancement
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched)

// Definie les parametres de la politique d'ordonnancement
int pthread_attr_setschedparam(pthread_attr_t *attr,
                               const struct sched_param *param)

// Definie la politique d'ordonnancement
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)


// Definie la taille de la pile
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize)

// Definie l'adresse de base de la pile
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr)

// Definie le comportement du thread en cas de terminaison
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)

//////////////////////
// Gestion des Mutex //
//////////////////////

// Detruit les attributs d'un mutex
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr)

// Initialise les parametres d'un mutex avec les valeurs par defaut
int pthread_mutexattr_init(pthread_mutexattr_t *attr)

// Detruit un mutex
int pthread_mutex_destroy(pthread_mutex_t *mutex)

// Initialise un mutex
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr)

// Essaye d'acquerir un mutex BLOQUANT
int pthread_mutex_lock(pthread_mutex_t *mutex)

// Essaye d'acquerir un mutex NON-BLOQUANT
int pthread_mutex_trylock(pthread_mutex_t *mutex)
```

```
// Relache le mutex
int pthread_mutex_unlock(pthread_mutex_t *mutex)

////////////////////////////
// Gestion des conditions //
////////////////////////////

// Initialise les attributs d'une condition
int pthread_condattr_init(pthread_condattr_t *attr)

// Detruit les attributs d'une condition
int pthread_condattr_destroy(pthread_condattr_t *attr)

// Debloque tous les thread qui attendent sur la condition
int pthread_cond_broadcast(pthread_cond_t *cond)

// Detruit la condition
int pthread_cond_destroy(pthread_cond_t *cond)

// Initialise la condition
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr)

// Debloque un thread attendant sur la condition
int pthread_cond_signal(pthread_cond_t *cond)

// Bloque l'execution jusqu'a la condition OU pendant le temps time
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           const struct timespec *time)

// Bloque l'execution jusqu'a la condition
int pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *mutex)


////////////////////////////////////////
// Gestion des verrous lecture/ecriture //
////////////////////////////////////////

// Detruit un verrou lecture/ecriture
int pthread_rwlock_destroy(pthread_rwlock_t *lock)

// Initialise un verrou lecture/ecriture
int pthread_rwlock_init(pthread_rwlock_t *lock,
                        const pthread_rwlockattr_t *attr)

// Essaye d'acquerir un verrou lecture/ecriture pour lire BLOQUANT
int pthread_rwlock_rdlock(pthread_rwlock_t *lock)

// Essaye d'acquerir un verrou lecture/ecriture pour lire NON-BLOQUANT
int pthread_rwlock_tryrdlock(pthread_rwlock_t *lock)

// Essaye d'acquerir un verrou lecture/ecriture pour ecrire NON-BLOQUANT
int pthread_rwlock_trywrlock(pthread_rwlock_t *lock)

// Relache un verrou lecture/ecriture
int pthread_rwlock_unlock(pthread_rwlock_t *lock)

// Essaye d'acquerir un verrou lecture/ecriture pour ecrire BLOQUANT
int pthread_rwlock_wrlock(pthread_rwlock_t *lock)
```

```
// Detruit un verrour d'ecriture
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr)

////////////////////////////////////
// Gestion des donnees specifiques //
////////////////////////////////////

// Cree une clef specifique a chaque thread
int pthread_key_create(pthread_key_t *key, void (*routine)(void *))

// Detruit la clef
int pthread_key_delete(pthread_key_t key)

// Renvoie la valeur de la clef pour le thread appelant
void * pthread_getspecific(pthread_key_t key)

// Fixe la valeur de la clef pour le thread appelant
int pthread_setspecific(pthread_key_t key, const void *value_ptr)
```