Systèmes d'exploitation **GLO-2001**

H16

RAPPORT – Travail Pratique Nº 2
Travaux avancés avec les threads Linux.

fait par: Alexandre Picard-Lemieux nom 111 103 625 matricule

Gaël Dostie nom 111096568 matricule

Résultat: _____ / 100

(Note : ce rapport est écrit de façon à vous faciliter la vie. En cas d'omission ou de différence entre ce rapport vierge et l'énoncé du TP, l'énoncé a priorité).

1. Niveaux de priorités des threads dans Linux (/15 pts)

1.1 Programmation de threads avec niveaux de priorités (/**18 pts**)

a)
```
main(): Création du thread 0
main(): Création du thread 1
main(): Création du thread 2
setpriority():0
setpriority():0
errno: 0
main(): Création du thread 3
errno: 0
main(): Création du thread 4
setpriority():0
errno: 0
setpriority():0
errno: 0
setpriority():0
errno: 0
```

b)
```
main(): Création du thread 0
main(): Création du thread 1
setpriority():0
setpriority():0
errno: 0
main(): Création du thread 2
errno: 0
```

main(): Création du thread 3
setpriority():0
errno: 0
main(): Création du thread 4
setpriority():0
errno: 0
setpriority():0
errno: 0

c)
main(): Création du thread 0
main(): Création du thread 1
main(): Création du thread 2
setpriority():0
main(): Création du thread 3
errno: 0
setpriority():0
setpriority():0
main(): Création du thread 4
errno: 0
errno: 0
setpriority():0
errno: 0
setpriority():0
errno: 0

d)
main(): Création du thread 0
main(): Création du thread 1
main(): Création du thread 2
setpriority():-1
main(): Création du thread 3
setpriority():-1
setpriority():0
errno: 0
errno: 13
errno: 13
main(): Création du thread 4
setpriority():0
errno: 0
setpriority():0
errno: 0

e)

main(): Création du thread 0
main(): Création du thread 1
setpriority():0
setpriority():0
main(): Création du thread 2

errno: 0
errno: 0
main(): Création du thread 3
setpriority():0
errno: 0
main(): Création du thread 4
setpriority():0
errno: 0
setpriority():0
errno: 0

1.2 Observation du temps d'exécution des threads avec différents niveaux de priorités (/ **12 pts**)

a)

Thread	Niveau de priorité PR	PID	% CPU utilisé
0	20	1527	19.9
1	20	1528	19.9
2	20	1529	19.9
3	20	1530	19.9
4	20	1531	19.9

b)

Thread	Niveau de priorité PR	PID	% CPU utilisé
0	20	1864	29.6
1	21	1865	23.6
2	22	1866	18.9
3	23	1867	15.3
4	24	1868	12.3

c)

Thread	Niveau de priorité PR	PID	% CPU utilisé	Pourcentage théorique CFS
0	20	1874	39.9	40.2
1	22	1875	25.9	25.7
2	24	1876	16.3	16.6
3	26	1877	10.6	10.7
4	28	1878	6.6	6.8

Détail des calculs pour CFS :

Priorité	Poids (w)	Fraction
120	1024	1024 / 2546 = 0.402
122	655	655 / 2546 = 0.257
124	423	423 / 2546 = 0.166
126	272	272 / 2546 = 0.107
128	172	171 / 2546 = 0.068
Total	2546	1

d)

Thread	Niveau de priorité PR	PID	% CPU utilisé
0	20	1880	24.6
1	20	1881	24.6
2	20	1882	24.6
3	22	1883	15.9
4	24	1884	10.3

e)

Thread	Niveau de priorité PR	PID	% CPU utilisé
0	16	1887	40.2
1	18	1888	25.6
2	20	1889	16.3
3	22	1890	10.3
4	24	1891	6.6

2. Variables de conditions et mutex pour l'implémentation d'un thread pool (25 pts)

```
#include "ThreadPool.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/* Thunk : In computer programming, a thunk is a subroutine that is created, often automatically,
    to assist a call to another subroutine. Thunks are primarily used to represent an additional
  calculation that a subroutine needs to execute, or to call a routine that does not support the
```

```
  usual calling mechanism. http://en.wikipedia.org/wiki/Thunk */
typedef struct {
    ThreadPool *pThreadPool; // pointeur sur l'objet ThreadPool
    int ThreadNum; // Numéro du thread, de 0 à n
} threadArg;

void *Thunk(void *arg) {
    threadArg *pThreadArg = (threadArg *)arg;
    ThreadPool *pThreadPool;
    pThreadPool = static_cast<ThreadPool*>(pThreadArg->pThreadPool);
    pThreadPool->MyThreadRoutine(pThreadArg->ThreadNum);
}


/* void ThreadPool(unsigned int nThread)
 Ce constructeur doit initialiser le thread pool. En particulier, il doit initialiser les variables
 de conditions et mutex, et démarrer tous les threads dans ce pool, au nombre spécifié par nThread.
 IMPORTANT! Vous devez initialiser les variables de conditions et le mutex AVANT de créer les threads
 qui les utilisent. Sinon vous aurez des bugs difficiles à comprendre comme des threads qui ne débloque
 jamais de phtread_cond_wait(). */
ThreadPool::ThreadPool(unsigned int nThread) {
    // Cette fonction n'est pas complète! Il vous faut la terminer!
    // Initialisation des membres
    PoolDoitTerminer = false;
    nThreadActive = nThread;
    bufferValide = true;
    buffer = 0;

    // Initialisation du mutex et des variables de conditions.
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&CondThreadRienAFaire,0);
    pthread_cond_init(&CondProducteur,0);

    // Création des threads. Je vous le donne gratuit, car c'est un peu plus compliqué que vu en classe.
    pTableauThread       = new pthread_t[nThread];
    threadArg *pThreadArg = new threadArg[nThread];
    int i;
    for (i=0; i < nThread; i++) {
        pThreadArg[i].ThreadNum = i;
        pThreadArg[i].pThreadPool = this;
    printf("ThreadPool(): en train de creer thread %d\n",i);
        int status = pthread_create(&pTableauThread[i], NULL, Thunk, (void *)&pThreadArg[i]);
        if (status != 0) {
            printf("oops, pthread a retourne le code d'erreur %d\n",status);
            exit(-1);
    }
   }
```

```
}

/* Destructeur ThreadPool::~ThreadPool()
   Ce destructeur doit détruire les mutex et variables de conditions. */
ThreadPool::~ThreadPool() {
    // À compléter
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&CondProducteur);
    pthread_cond_destroy(&CondThreadRienAFaire);
    delete [] pTableauThread;
}

/* void ThreadPool::MyThreadRoutine(int myID)
   Cette méthode est celle qui tourne pour chacun des threads crées dans le constructeur, et qui est
    appelée par la fontion thunk. Cette méthode est donc effectivement le code du thread consommateur,
    qui ne doit quitter qu'après un appel à la méthode Quitter(). Si le buffer est vide, MyThreadRoutine
    doit s'arrêter (en utilisant une variable de condition). Le travail à accomplir est un sleep() d'une
   durée spécifiée dans le buffer.
   */
void ThreadPool::MyThreadRoutine(int myID) {
    // À compléter
    printf("Thread %d commence!\n", myID);

    while (!PoolDoitTerminer) {
        pthread_mutex_lock(&mutex);

        if (!bufferValide) {
            pthread_cond_wait(&CondThreadRienAFaire,&mutex);
        }

        if(!PoolDoitTerminer) {
            printf("Thread %d récupère l'item %d!\n",myID,buffer);

            int currentBuffer = buffer;
            bufferValide = false;

            pthread_cond_signal(&CondProducteur);
            pthread_cond_signal(&CondThreadRienAFaire);

            pthread_mutex_unlock(&mutex);

            printf("Thread %d va dormir %d sec.\n",myID,currentBuffer);

            sleep(currentBuffer);
        }
    }

    printf("############ Thread %d termine!################\n",myID);
```

```
}

/* void ThreadPool::Inserer(unsigned int newItem)
   Cette méthode est appelée par le thread producteur pour mettre une tâche à exécuter dans le buffer
   (soit le temps à dormir pour un thread). Si le buffer est marqué comme plein, il faudra dormir
   sur une variable de condition. */
void ThreadPool::Inserer(unsigned int newItem) {
    // À compléter
    pthread_mutex_lock(&mutex);

    if (bufferValide) {
        pthread_cond_wait(&CondProducteur,&mutex);
    }

    bufferValide = true;
    buffer = newItem;
    pthread_cond_signal(&CondThreadRienAFaire);
    pthread_mutex_unlock(&mutex);
}

/* void ThreadPool::Quitter()
   Cette fonction est appelée uniquement par le producteur, pour indiquer au thread pool qu'il n'y
   aura plus de nouveaux items qui seront produits. Il faudra alors que tous les threads terminent
   de manière gracieuse. Cette fonction doit bloquer jusqu'à ce que tous ces threads MyThreadRoutine
   terminent, incluant ceux qui étaient bloqués sur une variable de condition. */
void ThreadPool::Quitter() {
    // À compléter
    pthread_mutex_lock(&mutex);

    while (bufferValide) {
        pthread_cond_wait(&CondThreadRienAFaire,&mutex);
    }

    PoolDoitTerminer = true;
    pthread_cond_broadcast(&CondThreadRienAFaire);
    pthread_mutex_unlock(&mutex);


    for (int i = 0; i < nThreadActive; i++) {
        pthread_join(pTableauThread[i],NULL);
    }
}
```

Sortie d'écran

./threadpool 3 10
Programme de test avec 3 threads et 10 items.
ThreadPool(): en train de creer thread 0
ThreadPool(): en train de creer thread 1
Thread 0 commence!
Thread 0 récupère l'item 0!
Thread 0 va dormir 0 sec.
Thread 1 commence!
ThreadPool(): en train de creer thread 2
(0.527) main: Je produis item numero 0 avec valeur 1.
main: item inséré.
(0.527) main: Je produis item numero 1 avec valeur 2.
Thread 2 commence!
Thread 0 récupère l'item 1!
Thread 0 va dormir 1 sec.
Thread 1 récupère l'item 1!
Thread 1 va dormir 1 sec.
main: item inséré.
(0.527) main: Je produis item numero 2 avec valeur 3.
Thread 2 récupère l'item 2!
Thread 2 va dormir 2 sec.
main: item inséré.
(0.528) main: Je produis item numero 3 avec valeur 4.
Thread 0 récupère l'item 3!
Thread 0 va dormir 3 sec.
Thread 1 récupère l'item 4!
Thread 1 va dormir 4 sec.
main: item inséré.
(1.528) main: Je produis item numero 4 avec valeur 1.
main: item inséré.
(1.528) main: Je produis item numero 5 avec valeur 2.
Thread 2 récupère l'item 1!
Thread 2 va dormir 1 sec.
main: item inséré.
(2.528) main: Je produis item numero 6 avec valeur 3.
Thread 2 récupère l'item 2!
Thread 2 va dormir 2 sec.
main: item inséré.
(3.528) main: Je produis item numero 7 avec valeur 4.
Thread 0 récupère l'item 3!
Thread 0 va dormir 3 sec.
main: item inséré.
(4.528) main: Je produis item numero 8 avec valeur 1.
Thread 1 récupère l'item 4!
Thread 1 va dormir 4 sec.
main: item inséré.

(5.528) main: Je produis item numero 9 avec valeur 2.
Thread 2 récupère l'item 1!
Thread 2 va dormir 1 sec.
main: item inséré.
(5.528) main: Destruction du thread pool.
Thread 2 récupère l'item 2!
Thread 2 va dormir 2 sec.
############ Thread 0 termine!################
############ Thread 2 termine!################
############ Thread 1 termine!################
(9.528) main: FIN!

4. Implémentation partielle d'une libraire de thread utilisateur (/60 pts)

Le listing du code source

Indiquez si certaines fonctions n'ont pas été implémentés ou sont susceptibles de planter

Sortie d'écran de l'exécution du code `TestThread.c`.