Systèmes d'exploitation **GLO-2001**

H16

RAPPORT – Travail Pratique Nº 2

Travaux avancés avec les threads Linux.

fait par:

Gabriel Laliberté	111 131 994
Gabriel Chantal	111 126 561

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)

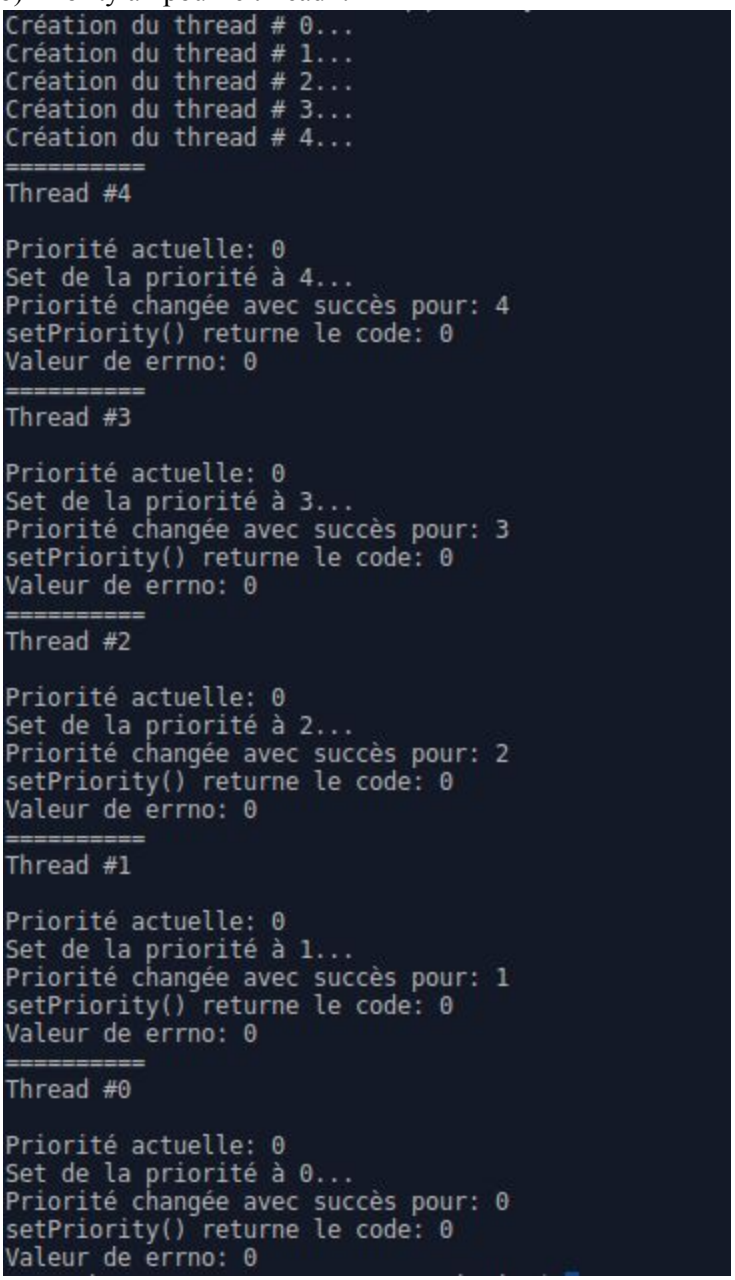
1.1 Programmation de threads avec niveaux de priorités (/**18 pts**)

Sorties d'écran:

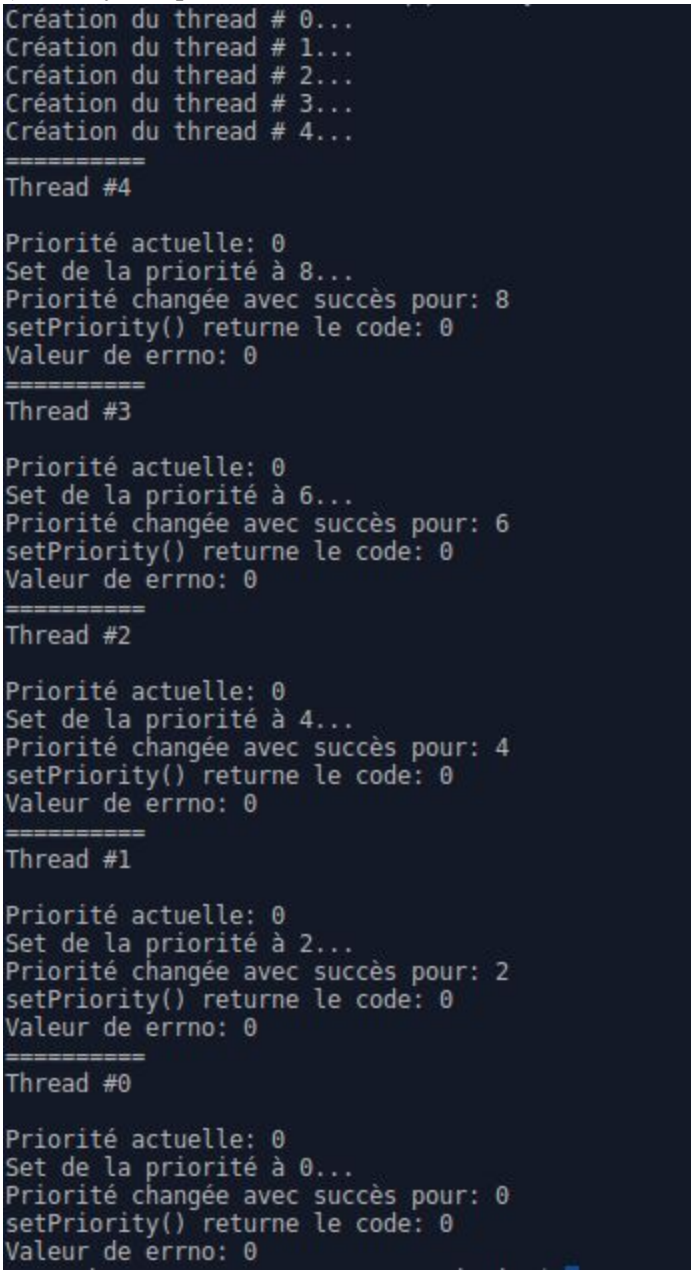
a) Priority à 0 pour tous les threads:

```
Création du thread # 0...
Création du thread # 1...
Création du thread # 2...
Création du thread # 3...
Création du thread # 4...
==========
Thread #4

Priorité actuelle: 0
Set de la priorité à 0...
Priorité changée avec succès pour: 0
setPriority() returne le code: 0
Valeur de errno: 0
==========
Thread #3

Priorité actuelle: 0
Set de la priorité à 0...
Priorité changée avec succès pour: 0
setPriority() returne le code: 0
Valeur de errno: 0
==========
Thread #2

Priorité actuelle: 0
Set de la priorité à 0...
Priorité changée avec succès pour: 0
setPriority() returne le code: 0
Valeur de errno: 0
==========
Thread #1

Priorité actuelle: 0
Set de la priorité à 0...
Priorité changée avec succès pour: 0
setPriority() returne le code: 0
Valeur de errno: 0
==========
Thread #0

Priorité actuelle: 0
Set de la priorité à 0...
Priorité changée avec succès pour: 0
setPriority() returne le code: 0
Valeur de errno: 0
```

b) Priority à i pour le thread i:



c) Priority à 2i pour le thread i:



d) Priority à (2i-4) pour le thread i:

```
Création du thread # 0...
Création du thread # 1...
Création du thread # 2...
Création du thread # 3...
Création du thread # 4...
==========
Thread #4

Priorité actuelle: 0
Set de la priorité à 4...
Priorité changée avec succès pour: 4
setPriority() returne le code: 0
Valeur de errno: 0
==========
Thread #3

Priorité actuelle: 0
Set de la priorité à 2...
Priorité changée avec succès pour: 2
setPriority() returne le code: 0
Valeur de errno: 0
==========
Thread #2

Priorité actuelle: 0
Set de la priorité à 0...
Priorité changée avec succès pour: 0
setPriority() returne le code: 0
Valeur de errno: 0
==========
Thread #1

Priorité actuelle: 0
Set de la priorité à -2...
CHANGEMENT DE PRIORITÉ ÉCHOUÉ.
setPriority() returne le code: -1
Valeur de errno: 13
==========
Thread #0

Priorité actuelle: 0
Set de la priorité à -4...
CHANGEMENT DE PRIORITÉ ÉCHOUÉ.
setPriority() returne le code: -1
Valeur de errno: 13
```

e) Sudo priority (2i-4) pour le thread i:

```
etu1@ubuntu:~/Documents/TP2/q1priority$ sudo ./t1
Création du thread # 0...
Création du thread # 1...
Création du thread # 2...
Création du thread # 3...
Création du thread # 4...
==========
Thread #4

Priorité actuelle: 0
Set de la priorité à 4...
Priorité changée avec succès pour: 4
setPriority() returne le code: 0
Valeur de errno: 0
==========
Thread #3

Priorité actuelle: 0
Set de la priorité à 2...
Priorité changée avec succès pour: 2
setPriority() returne le code: 0
Valeur de errno: 0
==========
Thread #2

Priorité actuelle: 0
Set de la priorité à 0...
Priorité changée avec succès pour: 0
setPriority() returne le code: 0
Valeur de errno: 0
==========
Thread #1

Priorité actuelle: 0
Set de la priorité à -2...
Priorité changée avec succès pour: -2
setPriority() returne le code: 0
Valeur de errno: 0
==========
Thread #0

Priorité actuelle: 0
Set de la priorité à -4...
Priorité changée avec succès pour: -4
setPriority() returne le code: 0
Valeur de 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**	**2399**	**19.9**
1	**20**	**2400**	**19.9**
2	**20**	**2401**	**19.5**
3	**20**	**2402**	**19.5**
4	**20**	**2403**	**19.2**

b)

Thread	Niveau de priorité PR	PID	% CPU utilisé
0	**20**	**2419**	**29.8**
1	**21**	**2420**	**23.8**
2	**22**	**2421**	**18.9**
3	**23**	**2422**	**15.2**
4	**24**	**2423**	**11.9**

c)

Thread	Niveau de priorité PR	PID	% CPU utilisé	Pourcentage théorique CFS
0	**20**	**2436**	**40.5**	**40.2**
1	**22**	**2437**	**25.6**	**25.7**
2	**24**	**2438**	**16.3**	**16.6**
3	**26**	**2439**	**10.6**	**10.7**
4	**28**	**2440**	**6.6**	**6.8**

Détail des calculs pour CFS :
Pour trouver le pourcentage d'utilisation alloué à chaque thread, il faut d'abord trouver la valeur *w* associée à chaque thread selon sa priorité. Pour les threads de 0 à 4, on obtient respectivement les valeurs de 1024, 655, 423, 272 et 172 pour *w*. Il suffit alors de faire un ratio de la valeur actuelle de *w* divisée par le total de tous les threads (2546) pour trouver le pourcentage théorique. Comme on peut le remarquer dans la colonne des pourcentages, leur valeur est très proche de celle actuellement utilisée par le CPU, soit à une marge d'erreur maximale de 0,3%. On peut donc conclure que le calcul théorique est très juste par rapport au pourcentage actuellement utilisé par le CPU.

d)

Thread	Niveau de priorité	PID	% CPU utilisé

	PR		
0	**20**	**2460**	**24.6**
1	**20**	**2461**	**24.6**
2	**20**	**2462**	**24.6**
3	**22**	**2463**	**15.6**
4	**24**	**2464**	**10.3**

e)

Thread	Niveau de priorité PR	PID	% CPU utilisé
0	**16**	**2491**	**40.1**
1	**81**	**2492**	**25.5**
2	**20**	**2493**	**16.2**
3	**22**	**2494**	**10.6**
4	**24**	**2495**	**6.6**

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

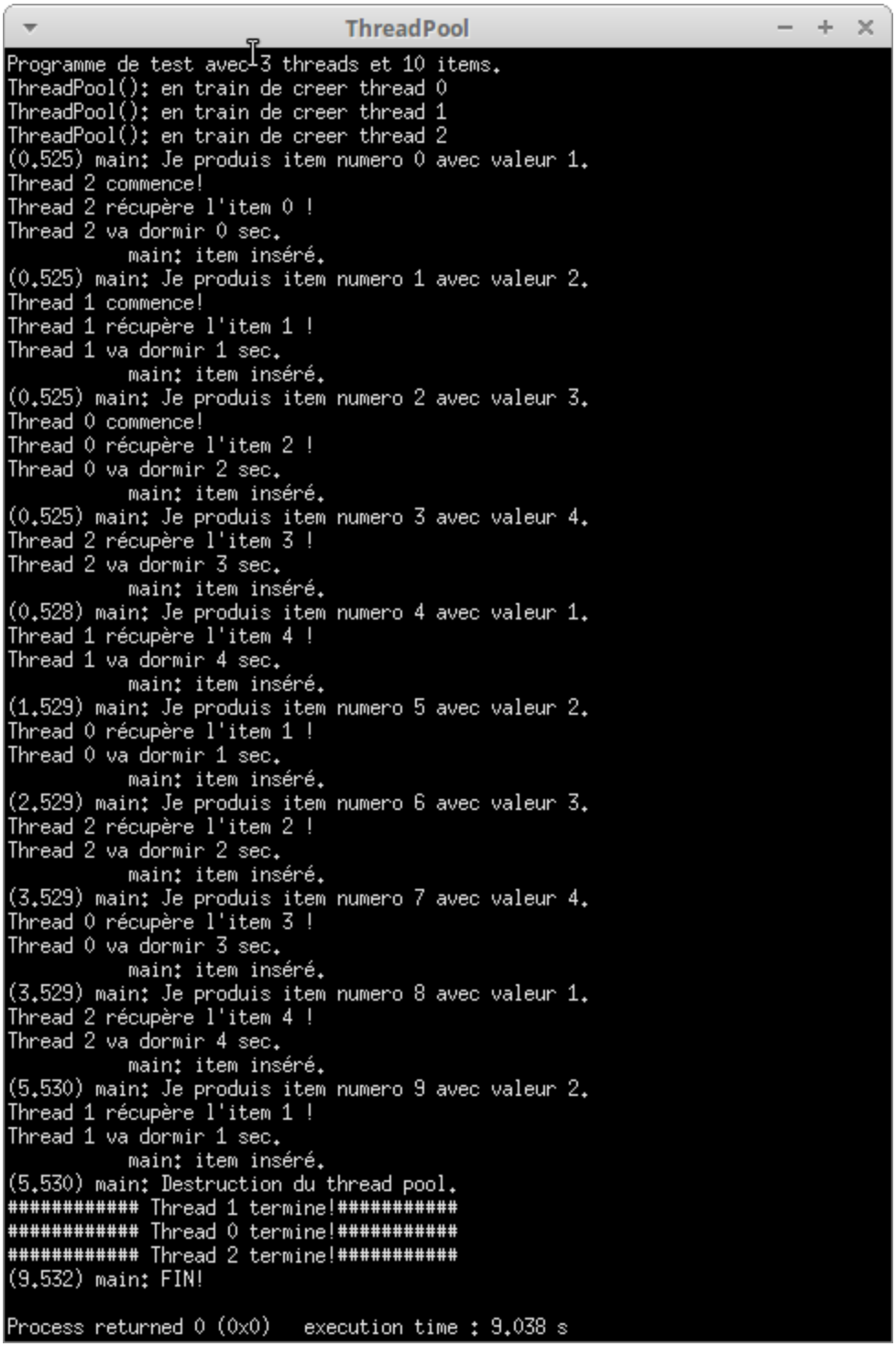
Incluez le code source ici (en plus de le mettre dans le zip).

```
#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
            this->nThreadActive = nThread;
            this->bufferValide = true;
            this->buffer = 0;
    // Initialisation du mutex et des variables de conditions.
            pthread_mutex_init(&this->mutex,0);
            this->PoolDoitTerminer = false;
            pthread_cond_init(&this->CondProducteur, 0);
            pthread_cond_init(&this->CondThreadRienAFaire, 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_cond_destroy(&this->CondProducteur);
   pthread_cond_destroy(&this->CondThreadRienAFaire);
   pthread_mutex_destroy(&this->mutex);
}

/* 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
          unsigned int temp;
          printf("Thread %d commence!\n", myID);
          while(!this->PoolDoitTerminer){
          pthread_mutex_lock(&this->mutex);
          while(!this->bufferValide){
          //printf("waiting condRienAFaire\n");
          pthread_cond_wait(&this->CondThreadRienAFaire, &this->mutex);
          }
          temp = this->buffer;
          this->buffer = 0;
          this->bufferValide = false;
          printf("Thread %d récupère l'item %d !\n", myID, temp);
```

```
        printf("Thread %d va dormir %d sec.\n", myID, temp);
        pthread_cond_signal(&this->CondProducteur);
        pthread_mutex_unlock(&this->mutex);
        sleep(temp);

        }
        printf("############ Thread %d termine!###########\n", myID);
        this->nThreadActive--;
        pthread_exit(0);

}

/* 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(&this->mutex);
    while(this->bufferValide){
    //printf("waiting condProd\n");
        pthread_cond_wait(&this->CondProducteur, &this->mutex);
    }
    this->buffer = newItem;
    this->bufferValide = true;
    pthread_cond_signal(&this->CondThreadRienAFaire);
        pthread_mutex_unlock(&this->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
        this->PoolDoitTerminer = true;
        pthread_cond_broadcast(&this->CondProducteur);
        pthread_cond_broadcast(&this->CondThreadRienAFaire);
        for(int i= this->nThreadActive - 1; i >= 0; i--){
        //pthread_cond_broadcast(&this->CondThreadRienAFaire);
        pthread_join(pTableauThread[i], 0);
        }
}
```

Incluez la sortie d'écran du programme.



3. 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`.