Systèmes d'exploitation **GLO-2001**
Philippe Giguère H16

Travail Pratique N^{o} 2
Travaux avancés avec les threads Linux.
Version 1.00

Le rapport

Vous devez utiliser obligatoirement le rapport TP2-rapport-glo2001-h16.rtf rempli comme réponse et sauvegardé dans le format PDF. Les tablettes mises à la disposition des correcteurs sont équipées pour les fichiers PDF. Pour toutes les questions de programmation, tous les codes sources doivent être fournis. Dans votre remise dans PIXEL, incluez ce rapport ainsi que les codes source, zippés dans un seul fichier. Notez que le code doit compiler sur la machine virtuelle fournie, puisque cette machine est celle du correcteur. Des erreurs de compilations entraîneront des pénalités jusqu'à 30 %.

Assurez-vous de commenter votre code raisonnablement, si vous devez écrire du code. Un code mal documenté pourrait être pénalisé.

Pour la machine virtuelle, utilisez le compte **etu1** avec le mot de passe suivant : « **etudiant** » (sans accents)

Ceci est la première partie du TP 2.
Il manque encore la question 3.

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

1.1 Programmation de threads avec niveaux de priorités (**12 pts**)

Nous avons vu en classe que l'ordonnanceur *Completely Fair Scheduler* (CFS) de Linux permet d'accorder différents niveaux de priorités à nos threads. Pour cette question, écrivez un programme en C sous Linux pour créer 5 threads POSIX avec différents niveaux de priorités. Les threads exécutent simplement une boucle sans fin, par exemple `while(1);`. Pour changer la priorité d'un thread, faites les appels systèmes suivants dans chaque thread:

```
ThreadID = syscall(SYS_gettid);
int ret = setpriority(PRIO_PROCESS, ThreadID, priority);
```

La valeur de `priority` sera entre -20 et 19. Les valeurs positives indiquent une baisse de la valeur de priorité statique $P_{statique}$ utilisée par l'ordonnanceur. Les valeurs négatives permettent d'hausser la priorité dans l'ordonnancement, mais ne sont permises que pour un programme démarré par un `superuser` (avec la commande `sudo`), afin qu'un utilisateur normal ne puisse pas s'approprier une part non-équitable du temps de CPU.

Faites des tests avec les quatre combinaisons de `priority` suivantes, en tant qu'utilisateur non-administrateur :

a) aucun changement de priorité pour tous les threads (donc valeur de 0 pour `priority`)
b) thread 0 à 0, thread 1 à 1, etc... jusqu'à 4 (intervalle de 1)
c) thread 0 à 0, thread 1 à 2, etc... jusqu'à 8 (intervalle de 2)
d) thread 0 à -4, thread 1 à -2, thread 2 à 0, thread 3 à 2 et thread 4 à 4

Faites un cinquième test et utilisez les valeurs de `priority` suivantes, en exécutant le programme avec les droits administrateurs (commande `sudo`) :

e) thread 0 à -4, thread 1 à -2, thread 2 à 0, thread 3 à 2 et thread 4 à 4
(Utilisez la commande "`sudo nom_de_programme`" pour faire tourner en mode `superuser`)

Imprimez à l'écran le code de retour de `setpriority()`, ainsi que la valeur de la variable global `errno` pour chaque thread créé. Incluez les sorties d'écran du programme dans le rapport. Pour plus d'information à ce sujet, veuillez consulter le site http://linux.die.net/man/2/setpriority. N'oubliez pas d'inclure les *headers* suivants, afin de pouvoir compiler :

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>
#include <sys/syscall.h>
```

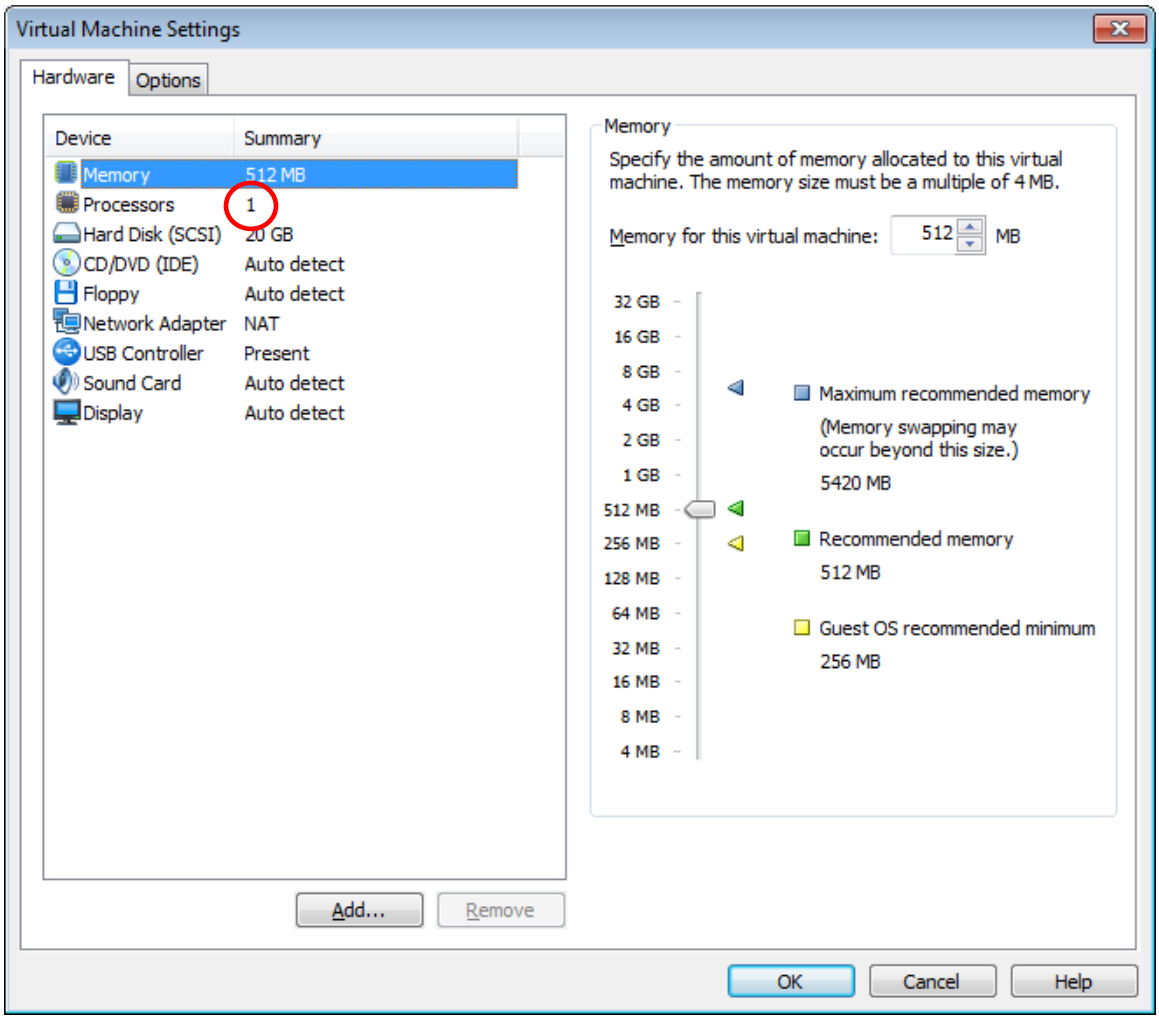
Incluez le code source dans votre rapport et dans le fichier .zip.

Assurez-vous que la fonction `main` fasse un `phtread_join()` afin de ne pas se terminer.

1.2 Observation du temps d'exécution des threads avec différents niveaux de priorités (**3 pts**)

Pour 2016, changer le score à plus, genre 6 ou 7 pts

Pour ces tests, assurez-vous que votre machine virtuelle n'utilise qu'un seul processeur. Par défaut, elle a été réglée de cette façon dans la configuration originale suivante, avec la deuxième ligne (Processors) indiquant 1 :



En utilisant la commande **`top -H`** dans une autre fenêtre terminal (puisque la première est occupée par votre programme qui tourne sans fin), observez le pourcentage de CPU utilisé par chacun des threads, pour les 5 cas précédents 2.1 a) b) c) d) e), (si possible). Pour faciliter la capture de cette information, faites `ctrl-c` pour arrêter le programme **`top`** quand les valeurs de pourcentage se sont stabilisées.

Inclure la valeur de la priorité `PR` et les `%CPU` dans votre rapport, à deux décimales près. Au passage, remarquez comment chaque thread possède son propre numéro d'identification **`PID`**.

Calculez les pourcentages théoriques que chaque thread devrait recevoir selon l'ordonnanceur Completely Fair Scheduler pour le cas c). Ajoutez ces valeurs dans le tableau. Donnez les détails des calculs dans votre rapport. Est-ce que les valeurs théoriques sont proches des résultats que vous avez obtenus?

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

Les fichiers pour cette question se trouvent dans le tarball `q2threadpool.tar` .

Pour cette question, vous allez implémenter en C++ un objet qui sera une version très simplifiée d'un *thread pool*. Un *thread pool* est un objet qui contient un nombre (généralement fixe) de threads, prêts à exécuter une tâche. Lorsque cette tâche est terminée, au lieu de quitter avec `pthread_exit()`, ils vont plutôt bloquer sur une variable de condition (`CondThreadRienAFaire`). Lorsqu'une nouvelle tâche est disponible, un des threads va l'exécuter. Ainsi, pour une application mutlithread qui cherche à paralléliser un grand nombre de petites tâches, nous sauvons du temps car les threads seront créées une seule fois et ne seront jamais détruites (sauf lorsque demandées explicitement par l'appelant).

Cet objet `ThreadPool` sera donc une forme de producteur-consommateur. Un seul thread producteur (la fonction `main`) produira des items qui seront donnés au `ThreadPool` et consommés par les threads de ce pool. La tâche à effectuer sera simple : faire un `sleep()` de la durée spécifiée par le producteur! Aussi, afin de limiter la complexité du TP, la taille du buffer contenant les tâches à effectuer sera de 1. Donc vous n'aurez pas à vous préoccuper d'implémenter une file pour stocker des tâches en attente. Vous pourrez baser la logique de votre code en grande partie sur l'exemple du manuel de la figure 2.32 à la page 139, car celui-ci contient un buffer d'une seule case.

Bien entendu, vous devrez utiliser un mutex dans les méthodes pour éviter les conditions de concurrence entre les threads consommateurs et l'unique thread producteur du `main` qui appelle `Inserer()`. Les variables partagées dans ce problème sont les membres de l'objet `ThreadPool`. Les différentes méthodes à implémenter sont les suivantes :

`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 en argument. 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()`. J'ai moi-même perdu une heure à cause de cela… Une partie du code vous est fournie dans le répertoire `q2threadpool`, car il n'est pas évident de démarrer un thread sur un objet C++ à l'aide d'un `Thunk`.

`~ThreadPool()`
Ce destructeur doit désinitialiser les mutex et variables de conditions.

`void ThreadPool::MyThreadRoutine(int myID)`
Cette méthode est celle qui tourne pour chacun des threads crées dans le constructeur, et qui est appelée par le `Thunk` (voir code). La première chose que cette méthode doit faire est d'afficher le message suivant :

`Thread` ***xx*** `commence!`

où ***xx*** est le numéro du thread passé par l'argument `myID`. Cette méthode est donc effectivement votre code du thread consommateur, qui ne doit quitter qu'après que la méthode `Quitter()` ait été appelée par quelqu'un d'autre. Si le buffer est vide, la méthode doit s'arrêter en utilisant la variable de condition CondThreadRienAFaire. Si le buffer contient quelque chose, il faut copier la valeur du buffer dans une variable locale, marquer le buffer comme vide et afficher le message suivant à l'écran :

`Thread` ***xx*** `récupère l'item` ***y***`!`

où ***xx*** est le numéro du thread myID et ***y*** est l'item lui-même. Par la suite, ce thread doit dormir pour le temps spécifié dans la valeur qui était dans le buffer. Juste avant de dormir, il devra afficher à l'écran

`Thread` ***xx*** `va dormir` ***y*** `sec.`

Bien entendu, ne faites pas la tâche du `sleep` dans la section critique, car cela empêcherait d'autres items d'être inséré par le producteur ou d'être consommés par d'autres threads `MyThreadRoutine`. L'idée ici est de pouvoir paralléliser les sleep. Lorsqu'un thread quitte, il devra afficher le message suivant :

`############ Thread` ***xx*** `termine!################`

`void Inserer(unsigned int newItem)`
Cette méthode est appelée par le thread producteur (externe au `ThreadPool`) pour mettre une tâche à exécuter dans le buffer (soit le temps à dormir pour un thread). Si le buffer est marqué comme plein/valide, il faudra dormir sur la variable de condition `CondProducteur`.

`void Quitter(void)`
Cette fonction est appelée uniquement par le producteur, pour indiquer à `ThreadPool` 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 et ceux qui vont retourner de leur `sleep`. Pour indiquer que l'on cherche à détruire cet objet, le drapeau `PoolDoitTerminer` est mis à `true`. Indice : c'est un bon endroit pour utiliser `pthread_cond_broadcast()` et aussi pour utiliser pthread_join()…

Autres informations

IMPORTANT! Assumez qu'il n'y aura jamais plus qu'un thread producteur, afin de simplifier le code. Ainsi, vous n'avez pas besoin de tenir compte du cas où un thread producteur voudrait `Inserer()` un objet dans une file pleine (donc en attente sur une variable de condition) pendant qu'un autre thread producteur fait l'appel de la méthode `Quitter()`.

Je n'ai pas mentionné où faire les `phtread_cond_signal`. Cela devrait être évident.

Vous ne pouvez pas modifier le fichier `ThreadPoolMain.cpp`. Vous pouvez modifier les autres fichiers à votre guise.

Voici la sortie d'écran de ma solution pour la commande `./threadpool 3 10` . Les valeurs entre parenthèses sont des timestamps. L'ordre et le timing d'exécution des threads sont bien entendus aléatoires à cause de l'ordonnanceur. La durée totale d'exécution devrait être similaire pour votre programme avec ces mêmes arguments.

```
Programme de test avec 3 threads et 10 items.
ThreadPool(): en train de creer thread 0
ThreadPool(): en train de creer thread 1
ThreadPool(): en train de creer thread 2
Thread 0 commence!
(0.047) main: Je produis item numero 0 avec valeur 1.
          main: item inséré.
(0.047) main: Je produis item numero 1 avec valeur 2.
Thread 0 récupère l'item 1!
Thread 0 va dormir 1 sec.
Thread 2 commence!
          main: item inséré.
(0.047) main: Je produis item numero 2 avec valeur 3.
Thread 1 commence!
Thread 2 récupère l'item 2!
Thread 2 va dormir 2 sec.
          main: item inséré.
(0.047) main: Je produis item numero 3 avec valeur 4.
Thread 1 récupère l'item 3!
          main: item inséré.
(0.047) main: Je produis item numero 4 avec valeur 1.
Thread 1 va dormir 3 sec.
Thread 0 récupère l'item 4!
Thread 0 va dormir 4 sec.
          main: item inséré.
```

```
(1.048) main: Je produis item numero 5 avec valeur 2.
Thread 2 récupère l'item 1!
Thread 2 va dormir 1 sec.
          main: item inséré.
(2.050) main: Je produis item numero 6 avec valeur 3.
Thread 1 récupère l'item 2!
Thread 1 va dormir 2 sec.
          main: item inséré.
(3.050) main: Je produis item numero 7 avec valeur 4.
Thread 2 récupère l'item 3!
Thread 2 va dormir 3 sec.
          main: item inséré.
(3.052) main: Je produis item numero 8 avec valeur 1.
Thread 0 récupère l'item 4!
Thread 0 va dormir 4 sec.
          main: item inséré.
(5.050) main: Je produis item numero 9 avec valeur 2.
Thread 1 récupère l'item 1!
Thread 1 va dormir 1 sec.
          main: item inséré.
(5.050) main: Destruction du thread pool.
Thread 0 récupère l'item 2!
Thread 0 va dormir 2 sec.
############ Thread 2 termine!#################
############ Thread 0 termine!#################
############ Thread 1 termine!#################
(9.050) main: FIN!
```