Travail Pratique 2

Utilisation des *threads*

À remettre, par le portail du cours le lundi 14 mars 2022 avant 23h59

Instructions

- Ce travail doit être fait en équipe de 2 ou 3.
- Tous les fichiers sources remis doivent compiler sur la machine virtuelle du cours. Les programmes qui ne compilent pas seront fortement pénalisés.
- Pour accéder à la machine virtuelle, utilisez l'utilisateur « etu1 » et le mot de passe « etudiant ».
- Un gabarit de rapport est fourni avec l'énoncé. Afin de faciliter la correction, vous devez obligatoirement utiliser ce gabarit pour l'écrire.

0. Programmation avec *threads* (10 pts bonus mais le total est sur 100 pts)

Cet exercice vous donne l'occasion d'apprendre à séparer une tâche en plusieurs *threads* pour en accélérer l'exécution sur des systèmes multi-cœurs. La tâche à accomplir est de multiplier deux matrices, ***G*** et ***H***. Le produit des deux matrices sera la matrice résultante ***R***. ***G*** sera de traille *N*×*M*, ***H*** de taille *M*×*P*. Le programme lancera *N*×*P* threads, et chacun des threads fera une partie de la multiplication matricielle.

Chaque thread devra recevoir deux vecteurs: la i^{ème} rangée de ***G*** et la j^{ème} colonne de ***H***. Les rangées de ***G*** et les colonnes de ***H*** sont en fait des vecteurs de même dimension. Chaque thread aura à calculer le produit scalaire de ces deux vecteurs, et ranger le résultat dans la matrice résultante ***R***.

À titre d'exemple, si nous avions les matrices

$$G = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \mathrm{H} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix}$$

en entrée, le produit de ces deux matrices serait

$$GH = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} = R.$$

Le premier thread serait alors responsable de calculer *a*, le deuxième *b*, et ainsi de suite.

Pour simplifier le problème, les tailles *M*, *N* et *P* sont définies statiquement au début du programme. Du code pour créer des matrices avec les bonnes dimensions est fourni dans le fichier `threads.c`. Il ne vous reste qu'à multiplier les matrices!

Toute l'information nécessaire sera passée lors de la création du *thread* avec une `struct` du type `ParametresThread`, qui a la définition suivante.

```
typedef struct {
      int iThread;
      int Longueur;
      double Vecteur1[M];
      double Vecteur2[M];
      double *pResultat;
} ParametresThread;
```

Dans `ParametreThread`, `iThread` est l'indice du *thread*, `Longueur` et la taille de `Vecteur1` et `Vecteur2`, et `pResultat` est l'emplacement mémoire où ranger le résultat du produit scalaire. Lorsqu'un *thread* termine, il écrit son résultat dans `pResultat` et quitte avec `pthread_exit(NULL)`.

Vous devez créer une copie de cette structure pour chaque *thread*, par exemple avec un tableau de type `ParametresThread`. Vous passerez ensuite l'adresse d'une des entrées du tableau en argument lorsque vous créerez le *thread*:

```
status = pthread_create(&threads[i], NULL, ProduitScalaire, (void *) &mesParametres[i]);
```

La fonction `ProduitScalaire` est celle exécutée par le *thread*. Vous devez coder cette fonction pour exécuter le calcul.

N'oubliez pas de *linker* la librairie `pthread` quand vous compilez votre programme, par exemple avec la commande `$ gcc MonProgramme.c -o MonProgramme -lpthread`.

Fournissez dans votre remise un fichier `threads.c` modifié de sorte qu'il fasse le travail demandé.

1. Niveau de priorité des *threads* dans Linux (30 pts)

Cette section vous invite à en apprendre plus sur le fonctionnement de l'ordonnanceur de Linux.

1.1 Programmation de *threads* avec niveaux de priorité (20 pts)

Pour cette question, écrivez un programme `priority.c` (dans la machine virtuelle du cours) qui crée cinq threads POSIX avec différents niveaux de priorité. Organisez votre programme en seulement deux fonctions, une fonction `main` et une fonction `work` qui contient le travail à faire par les threads. Les threads exécutent simplement une boucle sans fin, par exemple un `while(1)`. Pour changer la priorité d'un thread, il faut faire appel aux fonctions suivantes à partir de ce dernier.

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

Votre programme doit imprimer la valeur de retour de `setpriority` et la valeur de la variable globale `errno` pour chaque thread créé.

Pour accomplir le travail demandé vous aurez probablement besoin d'inclure les headers suivants.

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

La valeur de `priority` sera entre -20 et 19. Les valeurs positives indiquent une baisse de la valeur de priorité $P_{statique}$ utilisée par l'ordonnanceur. Les valeurs négatives permettent de hausser la priorité dans l'ordonnancementé

Faites des tests avec différentes combinaisons de priorités :

a) aucun changement de priorité pour les *threads* (autrement dit, une priorité de 0 pour tous les threads)

b) `priority` de 0 pour le premier thread, 1 pour le deuxième, etc.

c) `priority` de 0 pour le premier thread, 2 pour le deuxième, etc.

d) `priority` de -4 pour le premier thread, -2 pour le deuxième, 0 pour le troisième, etc.

e) même qu'en d) mais avez le programme lancé en mode *superuser* (`sudo`).

f) expliquez le résultat.

Pour chacune des combinaisons de priorité, inscrivez la sortie de votre programme au rapport. Incluez un fichier `priority.c` contenant votre code source dans votre remise.

1.2 Expérimentations avec différents algorithmes d'ordonnancement (10 pts)

Pour cette section, assurez-vous que votre machine virtuelle ne dispose que d'un seul cœur de calcul.

Le système d'exploitation Linux permet aux processus de demander à ce que leur algorithme d'ordonnancement soit changé. Pour demander un tel changement, on peut utiliser la fonction suivante, définie dans le header `<sched.h>`.

```
int sched_setscheduler(pid_t pid, int policy,
    const struct sched_param* param);
```

Cette fonction est documentée en détail dans la page web suivante : https://www.systutorials.com/docs/linux/man/2-sched_setscheduler/. Voici tout de même quelques instructions pour vous aider à l'utiliser. Donner 0 en guise de `pid` applique le changement d'ordonnanceur au processus courant. `policy` peut prendre diverses valeurs en argument, dont `SCHED_FIFO` et `SCHED_RR (RR est pour Round Robin)`. Ici `SCHED_FIFO et SCHED_RR sont en fait des entiers qui sont définis via #define.` `param` est une `struct` mais elle ne contient qu'une valeur qui nous intéresse, `sched_priority`. Sa définition ressemble donc à ce qui suit.

```
typedef struct {
...
    int sched_priority;
...
} param;
```

Pour bien initialiser la valeur de `sched_priority`, on doit appeler `int sched_get_priority_max(int policy)` ou `int sched_get_priority_min(int policy)` ou fournir quelque chose compris entre les deux.

> **Astuce** En général, lorsqu'un appel système échoue, vous pouvez obtenir plus de détails en affichant la valeur de la variable `errno` après l'appel. `errno` est malheureusement un entier et est difficilement interprétable. Vous pouvez utiliser la fonction `char* strerror(int error)` pour obtenir une valeur interprétable.

First In First Out (FIFO)

Dans votre fichier `priority.c`, écrivez une fonction ayant le prototype suivant :

```
void scheduler_to_fifo();
```

Cette fonction doit faire deux choses : changer l'ordonnanceur du processus courant vers un ordonnanceur *First In First Out* (FIFO) et imprimer à l'écran si le changement était un succès ou un échec.

Modifiez votre programme `priority.c` de manière à ce que l'ordonnancement soit de type FIFO. Donnez la même valeur de priorité à chaque *thread*. Inscrivez la sortie de votre programme au rapport.

> **Note** Votre programme pourrait avoir besoin des droits d'administration pour changer son algorithme d'ordonnancement.

Round Robin

Modifiez la fonction écrite dans la section FIFO pour qu'elle accepte un entier qui représente un ordonnanceur en argument. Son prototype devrait maintenant être le suivant.

```
void change_scheduler(int scheduler);
```

Rappelez-vous que cette fonction doit aussi afficher à l'écran le succès ou l'échec du changement de l'ordonnanceur.

Modifiez `priority.c` pour qu'il utilise un ordonnanceur *Round Robin.* Donnez la même priorité à chaque *thread*.

Expliquez la différence entre les sorties du FIFO et du Round Robin.

Assurez-vous que le fichier `priority.c de votre remise comprenne les fonctions main, change_scheduler et work.`

2. Problème de producteurs consommateurs (66 pts)

Cette question est en C++! Pour compiler le code fourni, utilisez la commande suivante :

```
$ g++ -o file file.cxx file-main.cxx -lpthread -std=c++11
```

Il se pourrait que le compilateur g++ ne soit pas installé sur votre machine virtuelle. Dans ce cas, vous pouvez l'installer avec la commande suivante :

```
$ sudo apt install g++
```

Cette section vous donne l'occasion de vous servir de variables conditionnelles. Votre tâche sera d'implémenter une file qui est *thread-safe*, c'est-à-dire qu'elle peut être utilisée par plusieurs *threads* et tout de même s'exécuter correctement.

Quatre fichiers sont fournis avec l'énoncé : `file-main.cxx`, `file.cxx`, `file.h` et `file-abstraite.h`. Vous devez modifier *seulement* les fichiers `file.h` et `file.cxx` pour cette question. En fait, les autres fichiers ne doivent pas être inclus dans votre remise.

Le fichier `file.h` définit une classe `File`. Cette dernière a quatre méthodes que vous devez compléter :

- `File::File()`, le constructeur;
- `void File::Insere(const ItemFile& item)`, une méthode qui insère item dans la file;
- `bool File::Retire(ItemFile& item)`, une méthode qui retire un élément de la file et qui l'assigne à la variable de sortie `item`;
- `void File::Termine()`, une méthode que l'utilisateur doit appeler pour indiquer que plus aucun élément ne sera ajouté à la file dans le futur.

De plus, le fichier `file.h` déclare les variables privées que le correcteur a utilisé dans sa solution. Vous êtes libres de les utiliser telles quelles ou de modifier les membres de `File` à votre guise.

Votre file doit avoir une capacité maximale de 10 éléments. Si l'utilisateur appelle `Insere` et qu'il n'y a pas d'espace disponible, l'appel à `Insere` doit bloquer jusqu'à ce qu'il y ait un espace disponible. Si l'utilisateur appelle `Retire` et qu'il n'y a aucun élément dans la queue, l'appel doit bloquer jusqu'à ce qu'il y ait un élément à consommer.

> **Indice** Les mots « *bloquer jusqu'à…* » sont un fort indicateur que vous aurez besoin de variables conditionnelles pour accomplir cette tâche.

On vous garantit que, une fois que l'utilisateur n'aura plus aucun item à ajouter à la file, il va appeler la fonction `Termine()`. Ainsi, il vous est possible d'avertir des threads en attente qu'ils peuvent se terminer parce qu'ils n'auront plus de données à consommer.

> **Indice** Les mots « *avertir des threads en attente…* » sont un fort indicateur que vous aurez besoin d'un *broadcast* pour accomplir cette tâche.

Si la méthode Termine a été appelée et que la file est vide, la méthode `Retire` ne peut plus faire son travail. Dans cette situation elle doit retourner false. Cela permet aux *threads* consommateur de savoir s'ils peuvent se terminer parce que plus rien ne sera ajouté à la file. La méthode Retire retourne toujours `true` dans les autres cas.

À titre indicatif, le fichier `file.cxx` complété par le correcteur tient sur 50 lignes. Le fichier `file-main.cxx` fourni vous permet de tester votre implémentation. Ce main affiche successivement les numéros de série des items ajoutés et retirés dans la file. Pendant vos tests, assurez- vous que :

- l'exécution se termine;
- les éléments sont retirés de la file dans le bon ordre (les numéros de série sont toujours en ordre croissant);
- tous les éléments que vous ajoutez à la file sont ensuite retirés.

Testez votre File pour les cas de figure suivants. Dans tous les cas, votre File doit avoir une capacité de 10 items.

Nombre de producteurs	Nombre de consommateurs	Nombre d'items
3	3	50
10	5	100
5	10	100
20	20	500

Inscrivez la sortie console du premier cas dans votre rapport. Si vous n'arrivez pas à faire fonctionner certains des cas, indiquez-le au rapport; vous aurez alors des points partiels.

Instructions de remise (4 pts)

4 points sont donnés pour le respect et la qualité des biens livrables.

Votre remise devrait contenir les fichiers suivants :

- Un fichier `rapport.pdf` contenant votre rapport. Ce fichier doit obligatoirement être basé sur le gabarit de rapport fourni avec l'énoncé.
- Un fichier nommé `threads.c` contenant le programme demandé à la section 0. Ce fichier doit être basé sur le fichier `threads.c` qui était fourni avec l'énoncé.
- Un fichier `priority.c` contenant le code écrit pour la section 1. Ce fichier devrait contenir les fonctions suivantes :
  - `change_scheduler`. La fonction qui permet de changer l'ordonnanceur d'un processus.
  - `work`. La fonction qui détermine le travail accompli par un thread.
  - `main`.
  - Aucune autre fonction.
- Un fichier `file.cxx` et un fichier `file.h`. Ils doivent contenir une classe `File` qui répond aux exigences de la section 2.
- Aucun autre fichier.