Département d'informatique et de génie logiciel
Pavillon Pouliot

GLO-2001
Systèmes d'exploitation pour ingénieurs
Travail pratique 2 – Hiver 2019
Travaux avancés avec les threads Linux

Conditions

Machine virtuelle: Vous devez utiliser la VM indiquée dans Contenu et activités > Machine virtuelle sur le portail de cours.

Norme : Le travail doit respecter la norme de programmation départementale présentée dans pixel.fsg.ulaval.ca >Applications >Utilitaires >Documents institutionnels>Normes de programmation

Remise : Vous devez suivre la procédure de remise de travaux disponible sur le portail de cours. Vous y trouverez aussi les dates de remise pour le travail préliminaire et le travail final.

Travail à faire **exclusivement en équipe de 2**. Toute communication entre équipes ou avec toute personne autre que l'enseignant est **strictement défendue**. Les politiques sur le plagiat, le français et le retard énoncées dans le plan de cours sont totalement applicables.

Vous avez des questions ? (ou même si vous n'en avez pas, allez regarder celles des autres) Vérifiez le document des questions fréquentes et des précisions sous l'énoncé dans la page du Travail sur le portail. Il se peut que votre question ait déjà été répondue. Si votre réponse ne s'y trouve pas, alors contactez l'enseignant. Il est là pour ça ☺

Biens livrables

1) Rapport de temps préliminaire TP2
   a) En utilisant le gabarit Excel fourni sur le portail de cours, faites un rapport de temps préliminaire (un **document Excel**) incluant pour chaque question (sous-questions à tous les niveaux) du travail une estimation des heures de travail que vous devrez effectuer pour terminer le travail.
   b) Remettre sur le portail de cours dans la page Rapport de temps préliminaire tp 2.
2) Travail pratique 2,
   a) Dans un fichier .zip
      i) Le rapport consiste en un compte rendu des manipulations et des exercices demandés. Le travail est divisé en plusieurs sections composées parfois de plusieurs points. Évitez de faire des captures d'écrans en format image, car elles génèrent des fichiers de plusieurs Mo. Sélectionnez plutôt le texte et copiez-le dans le document. Il faut également inclure les réponses aux questions et vos explications ou commentaires lorsqu'ils sont demandés. Vous devez utiliser obligatoirement le fichier "TP2-20191HSErapport.docx " pour le rapport, afin d'uniformiser la présentation pour le correcteur. 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 pouvant aller jusqu'à 50 %.
      ii) Votre rapport de temps final. C'est-à-dire le rapport de temps préliminaire (gabarit Excel .xlsx) dont la colonne temps réel a été complétée ainsi que le ratio temps réel divisé par temps estimé
      iii) Tous les fichiers de code source .c/.cpp et autres que vous avez créés, utilisés ou modifiés.
   b) Remettre sur le portail de cours dans la page Travail pratique 2 > Boîte de dépôt de la section Évaluations et résultats.

Objectifs

Ce travail vise à vérifier l'atteinte, en tout ou en partie, de ces objectifs du plan de cours :

1. Connaître les rôles, les fonctions et les composantes d'un système d'exploitation, et comprendre les principes théoriques de son fonctionnement. Ces principes comprennent :
   1. Les processus et threads,
   2. Les mécanismes d'ordonnancement des processus,

2. Comprendre les mécanismes, les méthodes et algorithmes utilisés dans la mise en application de ces principes théoriques.
3. Pouvoir utiliser certains systèmes d'exploitation courants (Linux, Windows).
4. Pouvoir développer des logiciels utilisant ou effectuant des fonctions systèmes
5. Apprendre à évaluer la charge de travail
6. Développer de bonnes aptitudes de communication

Contexte

1. Niveaux de priorités des threads dans Linux (20 pts)

1.1 Programmation de threads avec niveaux de priorités

Nous avons vu que l'ordonnanceur 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. Votre fichier doit s'appeler `nospriorites.c`.

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>
#include <sys/resource.h>
```

Chaque thread démarre, affiche un message l'identifiant et qui indique qu'il est démarré. Change sa priorité et affiche les informations demandées. Puis exécute simplement une boucle sans fin, par exemple `while(1);`. Le travail fait par les threads est dans la fonction nommée `work`.

Pour changer la priorité d'un thread, faites les appels système suivants dans chaque thread:

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

Affichez à l'écran le code de retour de `setpriority()` (pour plus d'information, veuillez consulter le site http://linux.die.net/man/2/setpriority.), ainsi que la valeur de la variable global `errno` pour chaque thread créé. Ajouter du texte pour permettre d'identifier chaque valeur.

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

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 1, thread 1 à 2, 2→3, 3 → 4, 4 → 5
c) thread 0 à 9, thread 1 à 7, 2→5, 3 → 3, 4 → 1
d) thread 0 à 0, thread 1 à -4, 2 → -2, 3 →3, 4 → 4

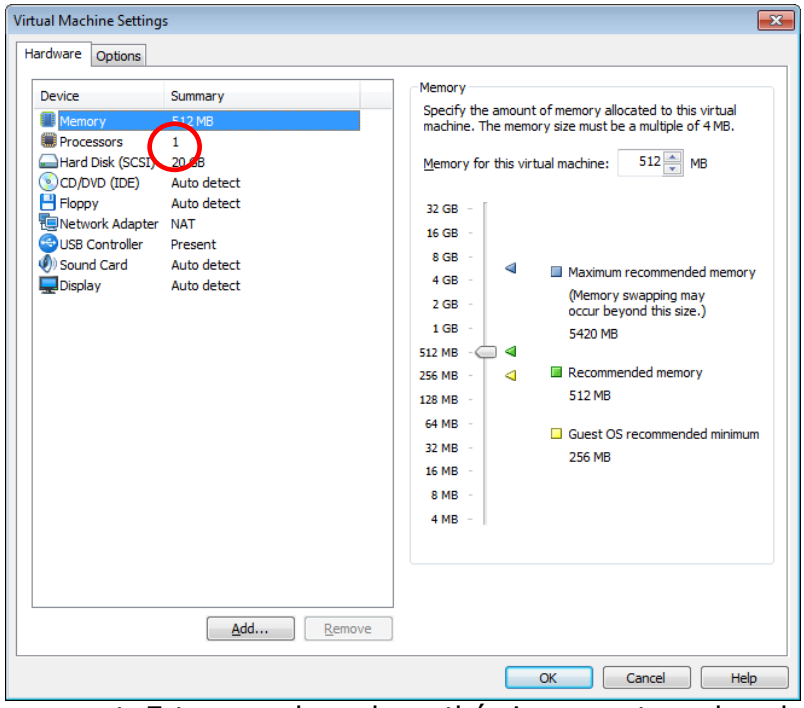
Faites un cinquième test et utilisez les valeurs de `priority` suivantes, en exécutant le programme avec les droits administrateur (commande `sudo`) :

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

Pour chaque test, dans votre rapport, identifier le test et inclure les sorties d'écran du programme. Joindre le fichier `nospriorites.c` à la remise.

1.2 Observation du temps d'exécution des threads avec différents niveaux de priorités

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 1.1 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`, de nice `ni` et les `%CPU` dans votre rapport. 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 b). 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? Commentez votre réponse.

1.3 Expérimentations avec différents algorithmes d'ordonnancement

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)` et 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 `nospriorites.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.
Copiez votre programme `nospriorites.c` en `nosprioritesfifo.c`. Modifier ce nouveau fichier 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. En utilisant la commande `$ top -H`, remplissez un tableau d'utilisation de processeur similaire à celui fait pour la section 1.2.

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

Round Robin

Copiez votre programme `nosprioritesfifo.c` en `nosprioritesrr.c`. 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 `nosprioritesrr.c` pour qu'il utilise un ordonnanceur *Round Robin.* Donnez la même priorité à chaque *thread*. Remplissez un tableau de l'utilisation de processeur attendue et constatée pour chaque thread.

Assurez-vous que les fichiers `nosprioritesfifo.c` et `nosprioritesrr.c` de votre remise comprennent les fonctions `main`, `change_scheduler` et `work`.

2. Implémentations de différents verrous (20 pts)

Nous allons reprendre l'exemple de la condition de concurrence lors de la mise à jour d'une variable globale partagée par deux threads. Nous étudierons l'effet d'une condition de concurrence mal gérée. Deux threads seront mis en compétition pour modifier une variable globale qui sera protégée par des mécanismes de verrou de bonne ou mauvaise qualité. Le code de départ pour cette question est `q2lock/q2lock.c` du fichier `.zip` fourni.

2.1 Implémentation d'un mauvais verrou : `BadLock`

Créez un fichier `q2lock1.c`, à partir du fichier fourni.

Créez deux fonctions `BadLock()` et `BadUnlock()` qui utilisent une variable entière lock comme verrou pivotant. La fonction `BadLock()` doit tester sans cesse la libération du verrou, puis le saisir dès qu'il est libre (une valeur de 0 indique un verrou libre, et une valeur non nulle indique qu'il est occupé).

2.2 Implémentation d'un verrou atomique : `AtomicLock`

Créez un fichier `q2lock2.c`, à partir du fichier fourni.

Créez deux fonctions `AtomicLock()` et `AtomicUnlock()` qui utilisent une variable entière `atomicLock` comme verrou pivotant. La fonction `AtomicLock()` doit utiliser la fonction `xchg()` du fichier `q2lock.c` qui appelle l'instruction assembleur `xchg`, pour tester et saisir le verrou. Une valeur de 0 indique un verrou libre, et une valeur non nulle indique qu'il est occupé.

Astuce Un pseudo-code montrant une utilisation possible de `xchg` est disponible sur Wikipédia: https://fr.wikipedia.org/wiki/Spinlock. Dans cette page votre fonction `xchg`s'appelle TestAndSet.

2.3 Utilisation de `pthread_mutex`

Créez un fichier `q2lock3.c`, à partir du fichier fourni.

Utilisez un `pthread_mutex` pour vous assurer que le code fonctionne de la bonne manière, i.e. complète avec une valeur finale de 20000000 pour `count`. Notez que vous ne pouvez pas empêcher un thread de s'exécuter au complet, i.e. la section critique doit être seulement `count++`.

2.4 Analyses

Pour chaque cas :

1. Que vous n'utilisiez aucun mécanisme d'exclusion mutuelle,
2. Que vous utilisiez le `BadLock`,
3. `AtomicLock`
4. `pthread_mutex`.

Calculez le temps moyen par itération et la proportion d'itérations où on observe une condition de concurrence (un échec de l'incrémentation du compte, en d'autres mots). Indiquez aussi si chaque algorithme est *correct* (son taux d'échec sera toujours strictement 0). Faites l'analyse avec un et deux processeurs. Inscrivez ces résultats dans le rapport.
Remplissez le tableau du rapport.

Notes :
Pour le test d'échec, rapportez le temps moyen par lock/unlock. Donc, vous pouvez réduire le nombre de boucles à 100.

Notez que vous ne pouvez pas empêcher un thread de s'exécuter au complet, i.e. la section critique doit être seulement `count++`.

Pour certains essais, vous n'avez pas toujours besoin de faire toutes les itérations. Par exemple, si vous observez que le taux d'échec est de 10+%, vous pouvez interrompre les tests. Pour d'autres, les taux d'échec sont plus proches de 0,1 % : il vous faut donc faire tourner le programme pendant plusieurs heures. Prévoyez donc un temps suffisant pour ces tests.

3. Producteurs-Consommateurs en utilisant une file FIFO (60 pts)

Cette question est en C++. 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++
```

Vous devez terminer un programme qui contient des producteurs et des consommateurs. Chaque item produit par un thread producteur est un objet contenant trois attributs. Le premier attribut représente le numéro de série unique du produit. Ce numéro de série augmente de 1 à chaque création de nouveau produit, peu importe quel thread producteur produit l'objet. Le deuxième et le troisième attribut représentent respectivement le nom et le prix du produit en question. Ces informations sont pigées au hasard dans un fichier texte contenant dans chaque ligne un nom de produit ainsi que son prix.

Le code nécessaire est dans le fichier **codeTP2.zip**.

Explication du code de main() :

Dans le code `main.cpp` fourni, les threads producteurs transmettent les items aux consommateurs par l'entremise d'une file d'attente FIFO (*First In First Out*), qui est un objet. Lorsqu'un thread producteur a produit un item, il l'insère dans cette file par la méthode suivante :

```
CodeFile File::Insere(Produit &p)
```

Lorsqu'un consommateur retire un objet de la file, il le fait par la méthode :

```
CodeFile File::Retire(Produit &p)
```

Quand les producteurs ont terminé de produire des items (déterminé par la constante `MAX_ITEMS_APRODUIRE`), ils quittent. Le code dans `main` attend que tous les producteurs terminent :

```
for (i=0; i < N_PRODUCTEURS; i++) {
      pthread_join(threadsProd[i], NULL);
}
```

Puis le code de `main` indique à la file qu'elle doit avertir les consommateurs de terminer, quand cette file sera vide:

```
cout << "Nous devons flusher les consommateurs de la file!" << endl;; int
ItemsRestants = MaFile.FlushConsommateurs();
cout << "Il reste " << ItemsRestants << " items dans la file!" << endl;
```

La fonction `main` ensuite attend que les threads consommateurs terminent :

```
for (i=0; i < N_CONSOMMATEURS; i++) {
      pthread_join(threadsCons[i], NULL);
}
```

C'est donc un coup que tous les threads producteurs et consommateurs ont terminé que le programme va pouvoir quitter. Vous ne pouvez pas modifier cela.

Comment compiler le code du TP :

Pour construire le programme, vous devez utiliser l'utilitaire **`make`**. La manière dont **`make`** doit compiler et linker le code est décrite dans un fichier appelé **`Makefile`** dans le même répertoire. Pour compiler le projet, il faut juste faire : **`make tp2`**

Pour effacer tous les objets sources ainsi que le programme, il faut faire : **`make clean`**

Pour exécuter le code, il suffit de faire : **./tp2**

Pour plus d'explications sur la manière dont **`make`** fonctionne, vous pouvez consulter par exemple ces deux adresses Web : lien 1 et/ou lien2.

Le code compilé fourni avec ce TP sera exécutable, mais ne fera rien de bien utile puisqu'il manque ce qui suit.

3.1 Contrôler les accès à la variable globale `ProchainNumeroSerie` (10 pts)

Le prochain numéro de série, qui est utilisé par les threads producteurs, est stocké dans une variable globale :

```
int ProchainNumeroSerie = 1;
```

Dans le code `main.cpp` distribué, les accès à cette variable globale par les threads ne sont pas protégés, ce qui est incorrect. **Votre première tâche est donc de corriger le code dans `main.cpp` afin de protéger les accès à cette variable, par un mutex.**

Expliquez, en quelques lignes maximum, les changements faits au code de `main.cpp`. Idéalement, mettez en évidence les changements dans le code source inclus dans le rapport, comme cette ligne en gris.

3.2 Implémentation de la classe Produit (10 pts)

Vous devez compléter l'implémentation de la classe `Produit` avec la méthode publique suivante (en plus, bien entendu, du constructeur et des accesseurs) :

```
void Produit::PigerProduit(void);
```

Cette méthode doit sélectionner un produit au hasard parmi ceux qui sont présents dans le fichier texte « Produits.dat ». Libre à vous de choisir la méthode que vous voulez appliquer, mais nous vous conseillons d'utiliser un conteur vector de la STL pour stoker les produits. Il suffit par la suite d'utiliser la fonction `rand()` afin de choisir un produit au hasard. Veuillez consulter ce lien pour plus d'informations.

La fonction PigerProduit() doit modifier les attributs de l'objet afin de donner des nouvelles valeurs à NomProduit et à PrixProduit. Il est à noter que le main contient la ligne suivante permettant d'initialiser le générateur de nombre aléatoire afin d'avoir un tirage différent à chaque exécution du code :

```
srand(time(NULL)); // initialisation de rand
```

3.3 Implémentation de la classe `File` (40 pts)

Vous devez compléter l'implémentation de la classe `File` avec les trois méthodes publiques suivantes (en plus du constructeur et du destructeur) :

```
CodeFile File::Retire(Produit &p);
```

La méthode `Retire()` doit retirer à chaque fois le premier item de la file. Si la file est vide, **l'appel `Retire` doit bloquer**. Cet appel débloque lorsqu'un nouvel item est disponible dans la file. Cette méthode doit imprimer à l'écran l'un des messages suivants :

a) `"--File::Retire() La file termine, nous devons quitter."`
Ceci signifie que la méthode `File::FlushConsommateurs()` a été appelée précédemment, et qu'il n'y a plus d'items dans la file. La fonction retourne alors le code `FILE_TERMINEE` pour indiquer au thread consommateur appelant qu'il doit quitter. Le thread consommateur appelant va alors quitter, puisqu'il vérifie pour ce code de retour (voir `main.cpp`).

b) `"--File::Retire() La file est vide. Allons dormir!"`
Vous devez imprimer à l'écran cette ligne quand la file est vide, juste avant de bloquer en utilisant une variable de condition (p.ex. CondCons).

c) `"--File::Retire()produit avec numero de serie x. Nombre de produits dans la file = y"`
Lorsque vous retirez un item de la file, vous imprimez cette information à l'écran.

Cette méthode retourne `FILE_ITEM_VALIDE` si un produit a été retiré avec succès, ou `FILE_TERMINEE` dans le premier cas mentionné auparavant (a).

```
CodeFile File::Insere(Produit &p);
```

La méthode `Insere()` sert à insérer un produit dans la file. Si la file est pleine, l'appel `File::Insere` **doit bloquer**. Il débloquera si de la place est faite dans la file, suite à un retrait d'un produit par `Retire()`.

Cette méthode doit imprimer à l'écran l'un des messages suivants :

a) `"++File::Insere() File pleine! Allons dormir!"`
Vous devez imprimer à l'écran cette ligne quand la file est pleine, juste avant de bloquer en utilisant une variable de condition (p.ex. CondProd).

b) `"++File::Insere()produit avec numero de serie x. Nombre d'items dans la file = y"`
Lorsque vous insérez un produit dans la file quand il y a de la place, vous imprimez cette information à l'écran.

La méthode retourne simplement `FILE_ITEM_VALIDE`.

```
int File::FlushConsommateurs(void);
```

La méthode `FlushConsommateurs()` est appelée pour indiquer à la file qu'aucun nouvel item ne sera inséré dans la file à partir de ce moment. Après cet appel, la file continue de fonctionner normalement, jusqu'au moment où la file est vide. À ce moment-là, la file doit indiquer aux appelants de `File::Retire()` que la file va se terminer, et que les appelants doivent donc quitter. Voir le cas a) de la méthode `File::Retire()`. Nous vous conseillons d'utiliser la fonction `pthread_cond_broadcast()` et un booléen (p.ex. `FileTourne`) permettant de contrôler la file.

Autres détails :

Prenez garde de ne pas avoir de threads consommateurs qui restent bloqués sur l'appel `File::Retire()` quand la file est vide, après l'appel `File::FlushConsommateurs()`. Le symptôme de ce problème sera que le programme bloque éternellement sur :

```
pthread_join(threadsCons[i], NULL);.
```

Tous les mécanismes de synchronisation et de contrôle d'accès relatifs à la file (variables de conditions, mutex, etc.) doivent être pris en charge dans la classe `File`, donc par vous. Il ne vous est pas permis d'avoir un mutex visible de `main.cpp` qui contrôle l'accès à cette file. Le seul mutex visible de `main.cpp`, celui pour le numéro de série `ProchainNumeroSerie`, ne doit pas être utilisé pour contrôler l'accès à la file.

La file doit avoir de la place pour contenir exactement 10 items au maximum. De plus, nous vous demandons d'utiliser le conteneur *queue* de la librairie STL pour stocker et manipuler les items de votre file.

Votre classe `File` doit être 100% compatible avec le `main.cpp` fourni. Nous devons donc être capables de prendre les fichiers `File.cpp` et `File.h` que vous avez créés, et les compiler avec le reste du code du TP.

Vous ne pouvez pas modifier le code de `main.cpp`, **sauf pour l'ajout du code relatif au mutex pour le contrôle d'accès à la variable** `ProchainNumeroSerie`. Pour votre file, inspirez-vous du code présent dans ce lien qui montre un exemple d'utilisation des mutex et des variables de conditions.

Notez que tous les items produits doivent être consommés! Il faut donc faire attention de ne pas faire terminer tous les threads consommateurs avant que tous les items dans la file ne soient consommés, quand la méthode `File::FlushConsommateurs()` est appelée. Nous pouvons vérifier ce cas en regardant la sortie de votre programme. Une sortie du programme est incluse (`ExempleSortie.txt`), dans le cas où nous avons 3 producteurs, 3 consommateurs et 50 items produits.