TP Noté - LIFPCA Programmation Concurrente et Administration Système

1h30 (2h tiers temps)

23 Février 2023

Consignes générales

Le programme qui vous est demandé doit être écrit en `C++`. Vous n'êtes pas autorisés à échanger d'information (ni par le réseau, ni via vos téléphones, ni aucun autre moyen de communication). Vous n'êtes autorisés qu'à utiliser le système C5, qui doit être mis en plein écran pendant toute la durée de l'épreuve. Vous disposez normalement de suffisamment de connaissance avec ce qui a été fait en TP et TD (un rappel de C++ est fourni avec ce sujet).

Le fichier rendu à la fin du TP doit impérativement compiler et s'exécuter sans lever d'erreur dans la plupart des cas. Un code ne compilant pas aura un malus de 10 points. Le code écrit doit être clair et commenté. Il est préférable de traiter convenablement peu de questions plutôt que de fournir des codes non fonctionnels pour chaque partie.

Lisez bien tout le sujet avant de commencer. Les différentes parties du sujet doivent être traitées dans l'ordre et leur difficulté est croissante.

Prenez également le temps de lire le code du fichier fourni. Vous ne devez en aucun cas modifier le code des fonctions auxiliaires, ni celui de la fonction `main()` qui initialise les différents scénarios considérés. Toute modification de ces fonctions sera sanctionnée.

I Présentation du sujet

On s'intéresse à une plateforme de rendus de TP où des étudiants peuvent déposer leur rendu et le mettre à jour (le redéposer). Un enseignant s'occupe de la gestion de la plateforme.

On modélise ce système avec un thread `enseignant` et un nombre quelconque de threads `etudiant` qui peuvent appeler les fonctions suivantes :

- `deposerRendu()` : fonction appelée par un thread `etudiant` pour tenter de déposer son rendu.
- `archiverRendus()` : fonction appelée par le thread `enseignant` pour analyser les rendus et les archiver.
- `finaliserRendus()` : fonction appelée pour indiquer qu'il n'est plus possible de faire des rendus. Appelée uniquement dans `lancement_multithread()` sauf dans le cas de l'enseignant qui arrête de traiter les rendus après un certain laps de temps.

Le système a une capacité de stockage limitée et ne peut gérer qu'au maximum `MAX_RENDUS` non-sauvegardés. Une erreur se produira si plus de `MAX_RENDUS` appels à `deposerRendu()` sont faits par les étudiants avant l'appel à `archiverRendus()` par l'enseignant. L'objectif de ce TP est de fournir un mécanisme pour garantir que cette erreur ne se produise jamais. Autrement dit, il faudra que l'enseignant appelle `archiverRendus()` assez souvent pour libérer de l'espace de stockage.

II Prise en main du code et lancement des threads

Pour l'instant, le programme ne fait rien et n'utilise même pas les threads.

Exercice 1 *Modifiez la fonction* `lancement_multithread()` *pour qu'elle lance* `nb_etudiants` *threads exécutant la fonction* `etudiant()` *avec les bons paramètres (identifiant de l'étudiant, nombre de rendus que l'étudiant réalisera) et un thread exécutant la fonction* `enseignant()`*. La fonction doit faire en sorte que tous les threads s'exécutent en parallèle et attendre qu'ils terminent. Bien sûr, vous devez supprimer l'appel à* `A_FAIRE`*. Attention, il faut bien appeler* `gestion_plateforme->finaliserRendus()` *au bon endroit : quand les threads exécutant* `etudiant()` *ont tous terminé et avant que le thread exécutant* `enseignant()` *ne se termine.*

Relancez votre programme. Lorsqu'il vous pose la question « Entrez la version du code », répondez « `0` ». Pour l'instant, vous devriez obtenir une erreur. C'est normal, car pour l'instant les appels à `gestion_plateforme->...()` utilisent la classe `Plateforme_Sans_Protection`, qui comme son nom l'indique, ne garantit rien.

Exercice 2 *Expliquez le problème avec* `Plateforme_Sans_Protection`*. Pour cela, répondez dans le commentaire* `À FAIRE` *prévu à cet effet dans la classe* `Plateforme_Sans_Protection`*. Quel message d'erreur obtenez-vous ? Pourquoi ?*

III Une première solution naïve

Ré exécutez maintenant votre programme avec la version `1` (sur), qui va utiliser la classe `Plateforme_Sur` au lieu de `Plateforme_Sans_Protection`. Vous devriez obtenir une autre erreur « Cette partie n'est pas encore faite » (qui vous indique le numéro de ligne).

Exercice 3 *Implémentez la classe* `Plateforme_Sur`*. Cette classe doit agir comme un moniteur de Hoare. Elle doit par ailleurs s'assurer qu'un étudiant ne puisse pas déposer (dans* `deposerRendu()`*) s'il y a déjà* `MAX_RENDUS` *non traités (si le cas se produit, la fonction renvoie* `false` *sans faire le rendu). Un étudiant ne réussit donc pas nécessairement à faire tous ses rendus.*

Les appels aux fonctions `depotEffectif()` *et* `archiverRendusEffectif()` *s'occupent de vérifier que les choses sont faites correctement. Il est interdit de supprimer ces appels de fonctions mais vous ne devez appeler* `depotEffectif()` *que lorsque le dépôt est effectivement réalisé, c'est-à-dire quand la fonction*`deposerRendu()` *renvoie* `true`*.*

IV Version avec attente

Dans l'approche naïve, étudiant ne peut effectuer son dépôt que si le nombre courant de dépôts est inférieur à `MAX_RENDUS`. On veut maintenant mettre en place un mécanisme pour que `deposerRendu()` attende qu'il soit disponible de déposer.

Exercice 4 *Ré exécutez maintenant votre programme avec la version* `2` *(avec attente), qui va utiliser la classe* `Plateforme_Avec_Attente` *pour vos tests. Implémentez la classe* `Plateforme_Avec_Attente`*. Elle doit s'assurer qu'un étudiant attende tant qu'il ne peut pas déposer. Dans cette question, les étudiants peuvent toujours faire leurs rendus, mais devront peut-être attendre pour le faire : l'enseignant appelle* `archiverRendus()` *autant de fois que nécessaire (toutes les 400 microsecondes en pratique) et ne s'arrête que quand tous les threads exécutant* `etudiant()` *ont terminé.*

V Gestion de la fin de l'examen

Jusqu'ici le thread `enseignant` attend que tous les étudiants aient terminé leurs rendus avant de se terminer. Dans un cas plus réaliste, on peut imaginer que la plateforme soit fermée par l'enseignant au bout d'un certain laps de temps (et donc que certains étudiants ne pourront pas effectuer tous les rendus). C'est la version `3` du programme.

Exercice 5 *Écrivez une fonction* `enseignant_limite()` *qui sera exécutée à la place de la fonction* `enseignant()`*, et qui ne fera qu'un nombre fixé de fois (à vous de décider) appel à* `gestion_plateforme->archiverRendus()` *avant de se terminer. Attention, il devra certainement appeler* `gestion_plateforme->finaliserRendus()` *afin de positionner la variable* `encore_des_rendus` *à* `false`*.*

Pour vous aider : l'instanciation du thread enseignant peut maintenant être faite comme ceci :

```
thread t_enseignant;
if (version <= 2) {
  t_enseignant = thread(enseignant);
} else {
  t_enseignant = thread(enseignant_limite);
}
```

Exercice 6 *Dans ce cas, les threads exécutant* `etudiant()` *ont à nouveau à gérer un cas où un rendu ne peut pas être réalisé. Modifiez votre méthode* `Plateforme_Avec_Attente::deposerRendu` *pour gérer ce cas : comme pour la version naïve, la méthode devra toujours terminer, mais pourra parfois renvoyer* `false` *pour signaler que le dépôt n'a pas pu être réalisé.*

VI Version avec équité

La version précédente n'est pas forcément équitable : il est possible qu'un étudiant fasse tous ses rendus très rapidement et fasse tous les rendus permis par l'enseignant, empêchant ainsi les autres étudiants de faire leurs rendus.

Exercice 7 *Mettez en place un mécanisme pour éviter les cas de non-équité (un étudiant qui n'arrive pas à déposer alors que les autres ont déjà effectué plus de dépôts que lui et que les autres continuent à faire des rendus) dans la classe* `Plateforme_Equitable`*, instanciée par la version* `4` *du programme. Vous pouvez ajouter des variables globales et des fonctions auxiliaires pour gérer ce cas.*

Montrez que votre approche ne présente plus de cas de non-équité dans le commentaire `À FAIRE` *prévu dans la classe* `Plateforme_Equitable`*. Si vous n'avez pas suffisamment de temps pour implémenter votre approche, vous pouvez également la décrire à ce niveau et expliquer ses avantages et ses limites. Si vous avez des pistes pour améliorer votre solution, vous pouvez également les présenter ici.*

VII Le mot de la fin

Exercice 8 *Dans notre approche les étudiants déposent tous leurs rendus au même endroit. Que faudrait-il faire si chaque étudiant avait sa propre plateforme de dépôt ? Répondez à cette question dans le commentaire prévu à la fin du fichier C++.*

Voici enfin un barème indicatif pour la notation.

- **Clarté du code (code clair et commenté) :** 1 point
- **Création des threads :** 2 points
- **Explication du problème :** 1 point
- **Classe** `Plateforme_Sur` **:** 2 points
- **Classe** `Plateforme_Avec_Attente` **:** 4 points
- **Code Enseignant Limite :** 1 point
- **Gestion fin dans** `Plateforme_Avec_Attente` **:** 2 points
- **Classe** `Plateforme_Equitable` **:** 3 points
- **Justification équitabilité :** 2 points
- **Le mot de la fin :** 2 points
- **Un code ne compilant pas aura un malus de 10 points.**

Rappels sur C++11 et les threads

Pour vous aider, voici un rappel de la syntaxe C++11 pour les threads :

```
// Création et attente de terminaison d'un thread :
int main () {
  // ...
  std::thread t(f, 42, std::ref(x));
  // ...
  t.join();
}

// Déclaration d'un mutex
std::mutex m;

// Verrouillage/déverrouillage d'un mutex :
m.lock();
// ...
m.unlock();

// Instantiation d'un verrou :
{
  std::unique_lock<std::mutex> l(m);
  // ...
}

// Opérations sur une variable de condition :
std::condition_variable c;
c.wait(l);  // l de type verrou (std::unique_lock par exemple)
c.notify_one(); // ou c.notify_all();

// Opérations sur une variable de condition :
std::condition_variable_any c;
c.wait(m); // m de type mutex
c.notify_all(); // ou c.notify_all();
```