TP Noté - ASR7 Programmation Concurrente

1h30

1er Mars 2022

Consignes générales

Le programme qui vous est demandé doit être écrit en `C++` sur les machines de la salle. Vous n'êtes pas autorisés à échanger d'information (ni par le réseau, ni via vos téléphones, ni aucun autre moyen de communication), et vous n'avez pas accès à internet. Vous disposez normalement de suffisamment de connaissance avec ce qui a été fait en TP et TD (un rappel de C++ est fourni en fin de sujet).

Les machines de la salle vous permettent de vous connecter localement avec le login `tpr` et le mot de passe `tpr`. Vous sauvegarderez votre travail sur ce compte, et **vous le déposerez à la fin sous la forme d'un** *unique* **fichier** `spectacle.cpp` **sur le serveur** `http://tpnote.tpr.univ-lyon1.fr`. Vous n'oublierez pas de mettre votre **NOM, PRÉNOM et numéro étudiant** en commentaire sur les premières lignes du fichier rendu. Attention, bien rendre le fichier `.cpp` et pas l'exécutable !
Sans cela, votre travail ne pourra pas être considéré.

Le fichier rendu à la fin du TP doit impérativement compiler et s'exécuter sans lever d'erreur dans la plupart des cas. 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 peuvent être traitées indépendamment les unes des autres, sauf la partie II qui doit être traitée en premier. Vous pouvez donc commencer par la question avec laquelle vous vous sentez le plus à l'aise. Notez toutefois que leur difficulté est également croissante.

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

Le fichier se compile avec la ligne de commande suivante :

```
g++ -g -Wall -pthread -std=c++11 spectacle.cpp -o spectacle
```

I Présentation du sujet

On s'intéresse à un site de réservation de places pour un spectacle. Un vendeur met en place des ventes à chaque tour et des spectateurs veulent acheter une place à chaque tour. Malheureusement, le nombre de places disponibles est limité, et toujours inférieur au nombre de potentiels spectateurs.

On modélise ce système avec un thread `vendeur` et un nombre quelconque de threads `spectateur` qui peuvent appeler 4 fonctions :

- `achatSpectacle()` : fonction appelée par un thread `spectateur` pour tenter d'acheter une place
- `ajoutPlace()` : fonction appelée par le thread `vendeur` pour ajouter un place à la vente
- `plusDePlace()` : fonction appelée par le thread `vendeur` quand il a mis à disposition toutes les places du tour courant pour indiquer qu'aucune place ne sera ajoutée au tour courant
- `nouvellesPlaces()` : fonction appelée par le thread `vendeur` pour indiquer que de nouvelles places vont être ajoutées (appelée en début de chaque tour).

II Prise en main du code et lancement des threads

Compilez et exécutez le programme avec `./spectacle`. Vous devriez obtenir ceci :

```
$ ./spectacle
On lance les threads du tour 0
On achète une place alors que c'est déjà complet !
Abandon (core dumped)
```

Vous pouvez lancer avec l'option `--debug` pour avoir plus d'informations affichées. En ajoutant l'option `--no-check`, aucune vérification n'est réalisée. Vous devriez obtenir ceci :

```
$ ./spectacle --no-check --debug
On lance les threads du tour 0
Achat Spectacle.
Le spectateur 0 a obtenu une place au tour 0
2 places sont disponibles au tour 0
Ajout Place.
Ajout Place.
spectacle.cpp:227: Cette partie n'est pas encore faite, à vous !
Abandon (core dumped)
```

Pour l'instant, le programme ne fait pas grand chose : il n'utilise même pas les threads, mais lance juste la fonction `spectateur()` puis la fonction `vendeur()`. Le spectateur arrive à obtenir une place avant quelle ne soit mise à la vente, on a un problème !

Vous pouvez modifier le nombre de spectateurs et le nombre de tours avec les options `--nb-spectateurs` et `--nb-rounds`. Cela sera utile pour vos tests !

Exercice 1 *Modifiez la fonction* `lancement_multithread()` *pour qu'à chaque tour, elle lance* `nb_spectateurs` *threads exécutant la fonction* `spectateur()` *avec les bons paramètres (identifiant du thread, latence et numéro de tour) et un thread exécutant la fonction* `vendeur()` *(avec le numéro de tour en paramètre), puis attend que tous les threads terminent. Tous les threads doivent s'exécuter en parallèle. Bien sûr, vous devez supprimer l'appel à* `A_FAIRE` *ainsi que l'appel original aux deux fonctions* `spectateur()` *et* `vendeur()`*. Notez bien qu'à chaque tour, les threads doivent être créés puis attendus.*

Relancez votre programme (sans l'option `--no-check`). Pour l'instant, vous devriez obtenir une erreur. C'est normal, car pour l'instant les appels à `gestion_spectacle->...()` utilisent la classe `Spectacle_Sans_Protection`, qui comme son nom l'indique ne garantie rien.

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

III Une première solution naïve

Ré-exécutez maintenant votre programme avec l'option `--sur` (exécutez `./spectacle --sur`), qui va utiliser la classe `Spectacle_Sur` au lieu de `Spectacle_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* `Spectacle_Sur`*. Cette classe doit agir comme un moniteur de Hoare. Elle doit assurer que 1) deux personnes n'achètent pas la même place et 2) le vendeur ne modifie pas le nombre de places disponibles pendant que quelqu'un achète une place (mais le vendeur peut ajouter des places entre deux achats). La fonction* `ajoutPlace()` *doit incrémenter un compteur indiquant le nombre de places disponibles. La fonction* `achatSpectacle()` *doit prendre une place s'il en existe une immédiatement disponible et retourner* `false` *dans le cas contraire. Les fonctions* `plusDePlace()` *et* `nouvellesPlaces()` *ne sont pas utiles dans ce moniteur. Les appels aux fonctions* `achatEffectif()`*,* `ajoutEffectif()`*, ... s'occupent de vérifier que les choses sont faites correctement. Il est interdit de supprimer ou de déplacer ces appels de fonctions.*

IV Affichage des logs

Les threads exécutant `spectateur` et `vendeur` font de l'affichage en simultané, il peut arriver que des messages se mélangent à l'écran.

Exercice 4 *Modifiez le code de ces fonctions pour que l'affichage de chaque thread soit complet sur une ligne.*

V Version plus réaliste

Dans l'approche naïve, un spectateur n'achète une place que si elle est immédiatement disponible. On veut maintenant mettre en place un mécanisme pour que `achatSpectacle()` attende qu'une place soit disponible ou qu'il n'y ait plus de disponibilités au tour courant (cela sera indiqué par un appel à la fonction `plusDePlace()`).

Exercice 5 *Exécutez maintenant la commande* `./spectacle --attente`*. Implémentez la classe* `Spectacle_Avec_Attente`*. Pour simplifier vos tests, vous pouvez utiliser l'option* `--nb-rounds 1` *dans un premier temps afin de vérifier le bon comportement avec un seul tour.*

VI Version sans famine

Les spectateurs qui se connectent au site de réservation ont une latence qui ne change pas au cours du temps. Si plusieurs tours sont effectués, les spectateurs qui ont une plus grande latence sont susceptibles de ne jamais avoir de place, alors que les autres vont réussir à acheter un grand nombre de places (on limite à un achat maximum par tour, mais on suppose que chaque spectateur cherche à acheter une place à chaque tour).

Exercice 6 *Assurez-vous qu'un spectateur qui n'a pas encore été servi (au tour précédent ou à un tour encore plus ancien) soit prioritaire par rapport aux spectateurs qui ont déjà été servis. Pour cela, vous pouvez empêcher aux threads ayant déjà réussi à acheter une place d'acheter une autre place tant qu'il existe des threads qui n'ont toujours pas eu la moindre place. Implémentez la classe* `Spectacle_Sans_Famine` *pour résoudre ce problème. Vous pouvez ajouter des structures de données en variable globale, des fonctions annexes, et/ou modifier les prototypes de fonction (à faire dans toutes les classes pour que le code continue de compiler...)*

Y a-t-il encore une possibilité de famine avec votre approche ? Si oui, donnez un exemple et proposez une amélioration de l'algorithme. Si non, justifiez. Pour cela, répondez dans le commentaire `À FAIRE` *prévu à cet effet dans la classe* `Spectacle_Sans_Famine`*. 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 7 *Dans notre approche, à chaque tours,* `nb_spectateurs+1` *threads sont créés puis détruit avant le tour suivant. Quelle est la limite de l'approche ? Quel mécanisme aurait pu être utilisé pour améliorer le code ? S'il vous reste du temps, vous pouvez tenter de l'implémenter (bonus).*

Voici enfin le barème prévu pour la notation. Il pourra être adapté en fonction de la réussite du sujet.

- **Clarté du code (code clair et commenté) :** 1,5 points
- **Création des threads :** 2 points
- **Explication du problème :** 1 point
- **Classe `Spectacle_Sur` :** 3 points
- **Accès à une variable partagée :** 1,5 points
- **Classe `Spectacle_Avec_Attente` :** 4 points

- **Classe `Spectacle_Sans_Famine` :** 3 points
- **Justification absence famine :** 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();
```