UGA - IUT2 - Département informatique
BUT Semestre 6 - Programmation Concurrente et Répartie - 2024/2025

TP02 : Mutex

0. Préambule

Vérifiez que vous avez bien installés les paquets suivants, en exécutant la commande (en étant root):

```
apt update && apt install -y build-essential clang libboost-all-dev
```

N'oubliez pas de faire les upgrade ce soir chez vous avec `apt upgrade` **(mais pas en début de TP !)**.

Les fichiers pour le TP sont sur chamilo, dans le dossier Documents, dans un fichier compressé pour le tp01 et le tp02. Les deux tps sont en lien, sur deux séances et partagent des fichiers de l'un à l'autre. **Si vous avez déjà chargé les fichiers lors du TP01, il n'y a donc rien de plus à charger**.

Pour le premier exercice, afin de rendre visibles les effets attendus, il convient de désactiver toute optimisation de compilation. Pour cela, vous compilerez vos programme avec la commande :

```
make OPT=n
```

au lieu de

```
make
```

Cela revient à compiler avec l'option **`-O0`** au lieu de **`-Os`**. Vous pourrez récompiler vos programmes avec les optimisations (commande `make` habituelle après avoir exécuté `make menage`) à l'issue de ce premier exercice.

1. Incrémentations concurrentes

Nous cherchons dans cet exercice à écrire dans `somme.cpp` un programme qui crée **`nb_threads`** tâches, chacune incrémentant **`RessourceProtegee::NB_INCR`** fois une même variable partagée, avant que la tâche principale n'affiche la valeur finale de celle-ci. Les tâches devront se synchroniser afin que cette valeur finale égale **`nb_threads*RessourceProtegee::NB_INCR`** tout en minimisant le surcoût temporel d'une telle synchronisation.

Nous avons vu en exercice de cours une modélisation différente d'un problème similaire :

- la variable partagée était une variable locale de la procédure principale;
- elle était manipulée directement dans les méthodes de la classe-fonction exécutée par les *threads*;
- le mutex était donc un attribut **`static`** de cette classe-fonction.

Dans cet exercice, nous suivons **presque** la modélisation recommandée en cours :

- la variable partagée est un attribut d'une classe **`RessourceProtegee`**;
- elle n'est manipulée que par les méthodes de cette classe, en sections critiques;
- le mutex est un attribut de cette classe (il n'est donc pas nécessaire de le construire indépendamment des instances de la classe).

Nous dévions de cette modélisation recommandée par le fait que :

- le *thread* principal manipule une copie de la variable partagée par l'affichage final;
- il utilise pour cela une méthode de la classe **`RessourceProtegee`** qui ne manipule pas la variable en section critique.

Nous discuterons de la validité de notre modélisation en fin d'exercice.

Instructions et Questions :

1. Se positionner dans le répertoire `tp02/somme/`.
2. Compléter le programme principal dans le fichier `somme.cpp` pour créer **`nb_threads`** tâches exécutant chacune l'opérateur **`()`** d'une instance de la classe **`MonThread`**. Ne pas oublier d'attendre la fin de leurs exécutions avant la fin du programme. Compiler (avec `make OPT=n`), puis exécuter plusieurs fois `bin-gcc/somme pls 10 30`.
3. Lors des exécutions précédentes, vous devez constater que le total en fin d'exécution n'est pas celui attendu. Pour remédier à ce problème, une première solution est d'utiliser la bibliothèque **`atomic`** qui permet les manipulations atomiques de variables. Vérifiez après cette modification que le total est désormais correct.
4. Relancer désormais le programme avec un seul thread pour constater comment l'affichage de chaque itération se fait quand il n'y a pas de concurrence entre threads.
5. L'affichage effectué au sein de chaque itération doit donc également être protégé pour ne pas avoir de mélange avec les itérations des autres threads et retrouver pour chaque itération un affichage des trois caractères `" 6,"` en un seul bloc. Compléter les fichiers `RessourceProtegee.hpp` et `RessourceProtegee.cpp` pour que les deux méthodes suivantes fassent l'affichage et manipulent l'attribut **`total`** en sections critiques :
   - **`incrementer_avec_une_sc()`** : la boucle **`for`** doit être incluse dans une section critique, incluant le **`endl`** final;
   - **`incrementer_avec_pls_sc()`** : seuls l'affichage et l'incrémentation de total doivent être inclus dans une section critique.

   Pour cela il faut ajouter un mutex comme attribut de la classe, et appeler ses méthodes **`lock()`** et **`unlock()`** dans les deux méthodes précédentes. Compiler (toujours avec `make OPT=n`), puis exécuter plusieurs fois la commande `bin-gcc/somme pls 10 30` afin de vous assurer que la synchronisation semble fonctionner lorsque chaque tâche effectue plusieurs sections critiques.

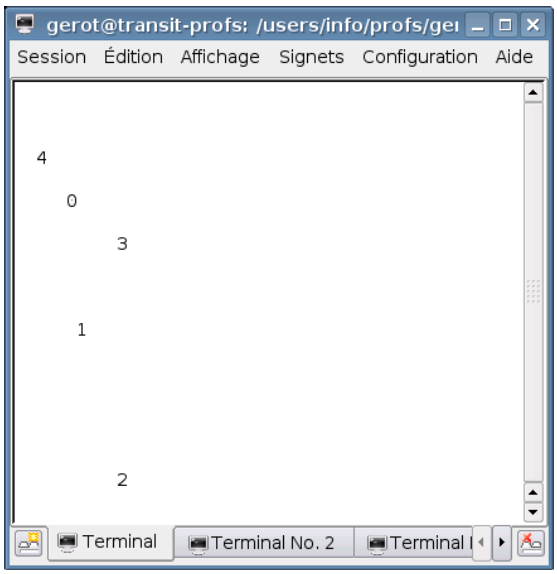
6. Executer à présent plusieurs fois la commande `bin-gcc/somme une 10 30` pour tester la version de votre programme où chaque tâche n'effectue qu'une seule section critique pour toutes ses sommes. Qu'observez-vous? Expliquer.

7. Exécuter `time bin-gcc/somme pls 30 100000 > /dev/null` puis `time bin-gcc/somme une 30 100000 > /dev/null` Qu'en déduisez-vous quant à l'utilisation des méthodes **`lock()`** et **`unlock()`**?

8. Comme précisé en introduction de cet exercice, l'affichage final exécuté par le *thread* principal est une manipulation non protégée de la variable partagée. Pourquoi est-elle sûre malgré tout ?

9. Annuler ce qui a été fait dans la question 3, **`total`** n'étant alors plus **`atomic`**. Exécuter le programme avec une ou plusieurs sections critiques. Pourquoi la somme totale reste conforme?

2. Androïdes avec une initialisation dynamique du numéro



Nous reprenons l'exercice du TP précédent qui consistait à modéliser, à l'aide de *threads*, l'exploration d'une nouvelle planète par des androïdes. Mais cette fois-ci, nous modélisons des androïdes explorateurs qui se déplacent tous à même vitesse. Par ailleurs, si dans le TP précédent chaque *thread* démarrait avec son numéro affecté par la tâche principale, dans ce TP, chaque androïde doit s'identifier lui-même avant de se mettre en marche.

La stratégie adoptée par votre équipe pour l'identification des *threads* est de définir une variable partagée **`numero`**, attribut de la classe **`SequenceEntiere`** initialisée à `0` et accessible par une méthode qui, après chaque accès, l'incrémente pour que les autres *threads* récupèrent d'autres valeurs pour s'identifier.

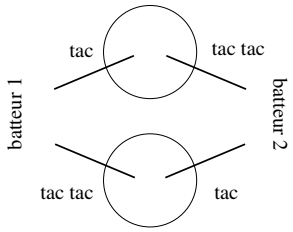
Instructions et Questions :

1. Se positionner dans le répertoire `tp02/androides/`.
2. Compiler `androides.cpp` et lancer l'exécutable. Vous devez constater un problème lié à une mauvaise identification (numérotation) des androïdes. Quel est-il ?

3. Afin de corriger ce problème, compléter dans les fichiers `SequenceEntiere.hpp` et `SequenceEntiere.cpp` la définition de la classe **`SequenceEntiere`** pour que chaque *thread* récupère un numéro identifiant différent en exécutant sa méthode **`nouveau_numero()`**.
   Consigne : le mutex sera utilisé avec l'emballage (ou *wrapper*) **`lock_guard<mutex>`** vu en cours.
4. Pourquoi l'ensemble de la méthode **`nouveau_numero()`** doit-elle être en section critique ?

5. Compiler et tester à nouveau votre programme pour vous assurer que **`nb_threads`** *threads* ont été créés et qu'ils se sont correctement identifiés.

3. Une batterie à deux toms pour deux batteurs

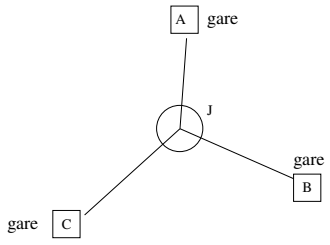


Cet exercice simule une batterie à deux toms utilisée par deux batteurs jazzmen positionnés de part et d'autre de l'instrument. Chaque batteur frappe un coup sur le tom qui se trouve à sa gauche puis deux coups sur le tom qui est à sa droite, puis répète le mouvement **`Batteur::NB_FRAPPES`** fois. Par ailleurs, les deux batteurs se coordonnent pour ne pas frapper sur un même tom en même temps.

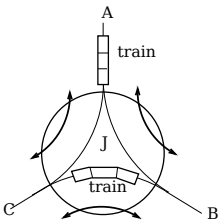
Instructions et Questions :

1. Si les deux batteurs se coordonnent correctement, combien de coups doit finalement recevoir chaque tom de la batterie ?
2. Se positionner dans le répertoire `tp02/batterie-partagee/`.
3. Compiler puis lancer l'exécutable. Que constatez-vous ?
4. Lorsque chaque batteur effectue chaque mouvement (un coup à gauche, deux coups à droite), a-t-il besoin de réserver les deux toms de la batterie en même temps ?
5. Corriger le programme pour résoudre ce problème, sans ajouter ni supprimer de lignes, mais en modifiant simplement leur ordre.
6. On souhaite à présent synchroniser non plus deux batteurs jazzmen, mais deux batteurs rustres, chacun répétant le mouvement suivant : taper un coup simultanément sur les deux toms de la batterie. Pour cela, modifier `batterie_partagee.cpp` pour créer des threads avec des instances de la classe **`BatteurRustre`**, puis modifier la méthode **`frapper_ensemble()`** pour réaliser la synchronisation voulue. *Attention : cette fois-ci, chaque batteur a besoin de réserver les deux toms en même temps; pour cela la fonction **`std::lock()`** vue en cours devra être utilisée.*

4. Trains



Trois villes, nommées A, B, C, sont reliées par des voies ferrées, comme indiqué sur le schéma ci-contre. Chaque voie est à double sens, mais à voie unique, donc les trains ne peuvent pas se croiser.



Une jonction J permet de relier les voies (avec des aiguillages complexes, voir la figure) et permet aux trains de parcourir les segments de voies sans s'arrêter.

On cherche à éviter les collisions entre trains, donc avant de partir d'une gare, un train doit s'assurer que les 2 segments de voies qu'il va parcourir sont libres. De plus, un seul train est autorisé à parcourir une voie donnée (même si ce serait possible dans le cas où les trains circulent dans le même sens).

1. Se positionner dans le répertoire `tp02/trains/`.
2. On modélise le problème par des *threads* représentant les trains (classe **`Train`**) et 3 mutex permettant de verrouiller les 3 segments de voies (classe **`VoieVille`**). Par ailleurs, afin de tester les collisions et de permettre les affichages, les positions de chacun des trains sont enregistrées dans un tableau protégé (classe **`Positions`**). Chaque train possède son trajet qui lui indique en particulier les deux segments qu'il devra réserver (classe **`Trajet`**). Enfin un *thread* supplémentaire est chargé d'afficher régulièrement la position des différents train (classe **`Temps`**). Parcourir les différents fichiers fournis pour comprendre la modélisation proposée.
3. Expliquer pourquoi la classe **`Positions`** utilise un mutex.
4. Exécuter le programme. Quel problème constate-t-on ? Quel phénomène s'est produit entre les *threads* ?

5. Analyser la trace affichée et justifier votre réponse précédente en complétant le graphe d'allocation des ressources ci-dessous.
6. Pour corriger ce problème, modifier le fichier `Train.cpp` pour que la réservation des ressources se fasse selon un ordre global. *Remarque : pour cela, vous n'utiliserez pas la fonction* ***`std::lock()`*** *comme pour l'exercice précédent, mais vous réserverez les voies selon l'ordre global défini grâce à la méthode* ***`departAvantArrivee()`*** *de la classe* ***`Trajet`***. *En effet, cette solution n'est qu'une étape intermédiaire vers une solution plus évoluée (question 10) où tous les trains ne réserveront pas simultanément les deux voies de leur trajet.*
7. Exécuter de nouveau et vérifier que le programme se termine normalement. Grâce à la commande `time`, noter la durée d'exécution de cette version.

8. Sur la trace affichée, arrive-t-il que 2 trains circulent en même temps ? Expliquer.

9. Modifier les **libérations** de verrou pour permettre à 2 trains de circuler en même temps, dans les cas où c'est possible, sans introduire de collisions. Tester et noter la durée d'exécution de cette version. Comparer cette durée avec celle de la version précédente.

10. Modifier les **réservations** de verrou pour qu'en plus, un train puisse parfois commencer à circuler dès la réservation de son premier segment. L'idée est illustrée par la figure de la jonction J : pour ne pas entrer en collision avec un train circulant en sens inverse, un train n'a besoin d'avoir réservé ensemble les deux segments de son trajet que lorsqu'il est sur la jonction; lorsqu'il circule sur un des deux segments, la réservation de celui-ci est suffisante. Attention néanmoins à combiner ce fait avec l'exigence d'absence d'interblocage et donc de réservation des segments selon un ordre global. Tester et noter la durée d'exécution de cette version. Comparer cette durée avec celles des versions précédentes.

Graphe d'allocation des ressources