HEIG-VD HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

Chargés de cours : Florian Vaussard, Rick Wertenbroek

Laboratoire de Programmation Concurrente

semestre automne 2022

Gestion d'accès concurrents

Temps à disposition : 4 périodes (deux séances de laboratoire)

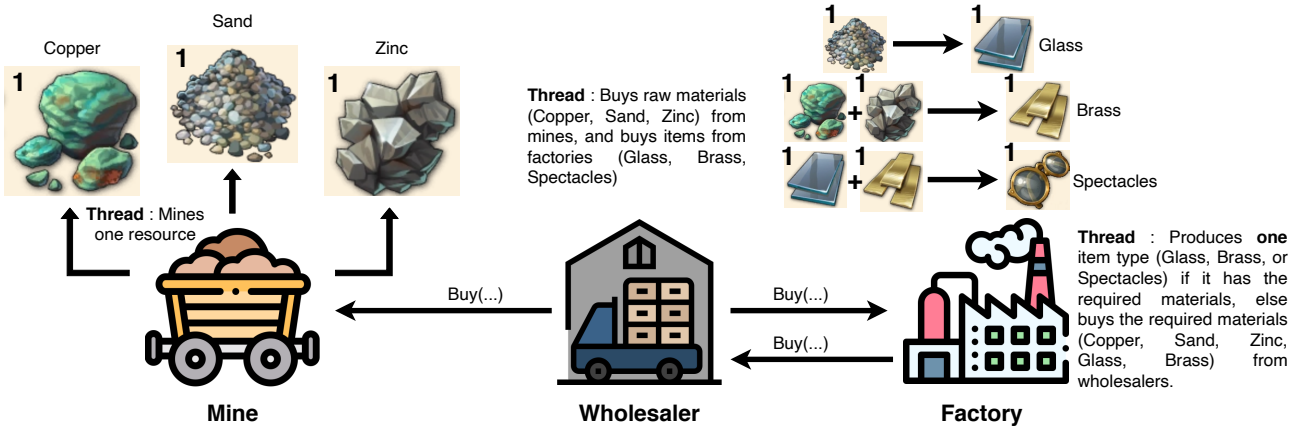
Récupération du laboratoire : `retrieve_lab pco22b lab3`

1 Objectifs pédagogiques

— Mettre en évidence les problèmes liés aux accès concurrents.
— Se familiariser avec les *mutex*.
— Protéger les accès concurrents à l'aide de *mutex*.

2 Cahier des charges

Nous souhaitons réaliser un logiciel simulant les ventes entre des mineurs, des grossistes et des usines, dans l'objectif de simuler des chaînes de production de lunettes (depuis la matière première jusqu'au produit). Le laboratoire portera sur la gestion des transactions entre les différents acteurs et donc la bonne gestion des fonds et des inventaires de chaque intervenant dans le marché.



Dans la simulation il y aura 3 mines, chacune produisant une ressource : le cuivre (copper), le sable (sand) ou le zinc, tout en payant un salaire au mineur qui mine la ressource. Il y aura aussi des usines spécialisées dans le production d'un seul objet : le verre (glass), l'étain (brass) ou des lunettes (spectacles). Pour produire ces objets elles paient un employé. Entre les mines et les usines il y a des grossistes (wholesalers), qui achètent les matières premières aux mines ainsi que les objets produits aux usines.

Dans ce laboratoire il s'agira d'implémenter la logique régissant les interactions entre les différents acteurs tout en faisant attention aux sections critiques.

3 Travail à faire

Il vous sera demandé de compléter les méthodes fournies par les mines, usines et grossistes selon

le comportement décrit ci-dessous, tout en gérant la concurrence. Le code associé se trouve dans les fichiers suivants :

— mine.cpp

— factory.cpp

— wholesaler.cpp

Interface vendeur (Seller)

L'interface vendeur (Seller), dont héritent tous les acteurs de la simulation, propose les méthodes et attributs suivants :

— `std::map<ItemType, int> get_items_for_sale()`
Cette méthode retourne une *map* des objets à vendre qui associe aux type(s) d'objet(s) vendus une quantité disponible.

— `int buy(ItemType what, int qty)`
Cette méthode représente un achat. Elle prend en entrée le type de ressource qu'on désire acheter ainsi que la quantité. La valeur retournée est le prix de la transaction si la transaction peut avoir lieu et sinon retourne 0 si la transaction ne peut pas avoir lieu (manque de stock, objet pas vendu, quantité négative passée en argument, etc.). Dans tous les cas où la méthode ne retourne pas 0, il faudra mettre à jour les stocks et les fonds du vendeur (la gestion des fonds et stocks de l'acheteur se fera hors de cette méthode par l'appelant).

— `std::map<ItemType, int> stocks;`
Les stocks de l'instance (type, quantité), ne correspond pas forcément aux objets en vente, par exemple une usine ne va pas vendre les matières premières (p.ex. cuivre) qu'elle utilise, seulement les objets qu'elle produit.

— `int money`
La quantité d'argent (les fonds) de l'instance.

Notons finalement que la méthode `int get_cost_per_unit(ItemType item)` retourne le prix de chaque ressource.

3.1 Les mines

Une mine est simulée par un thread dont le comportement est décrit ci-dessous. Il vous faudra implémenter la méthode `buy(...)` qui permet à un autre thread d'essayer d'effectuer un achat.

— La méthode `void run()` est la fonction du thread des mines, elle est contient déjà un minimum de code fonctionnel. La mine va, si elle a l'argent disponible, payer un mineur pour miner la ressource associée à la mine et attendre qu'il finisse son travail.

3.2 Les usines

Une usine est simulée par un thread dont le comportement est décrit ci-dessous. Il vous faudra implémenter la méthode `buy(...)` qui permet aux grossistes d'acheter des objets aux usines ainsi que le reste des méthodes décrites ci-dessous.

— La méthode `void run()` est la fonction du thread des usines, elle contient déjà un minimum de code fonctionnel. L'usine va vérifier si elle possède les ressources pour la production du type d'objet qu'elle produit. Si oui elle va le construire, si non elle va essayer d'acheter les ressources chez un grossiste.

— La méthode `build_item()` va permettre à l'usine de construire un objet selon le type qu'elle produit (**const** `ItemType built_item`). Les recettes sont indiquées dans le diagramme en page 1, la liste des "*ingrédients*" est donnée dans l'attribut `ressources_needed` (présenté ci-dessous). Pour construire un objet, il faudra payer un employé, comme dans les mines où l'on paie des mineurs. Pour ce faire des fonctions sont données dans `seller.h`.
  — `EmployeeType get_employee_that_produces(ItemType item)` permet de savoir quel type d'employé peut construire l'objet.
  — **int** `get_employee_salary(EmployeeType employee)` permet de savoir combien on doit payer l'employé pour produire une unité des objets qu'il peut produire (voir l'exemple dans `mines.cpp`).
— la fonction **void** `Factory::order_ressources()` permet aux usines d'acheter des ressources aux grossistes. Il faut donc pouvoir, pour chaque ressource dont une usine a besoin, demander aux grossistes s'ils peuvent la fournir. Le cas échéant incrémenter le stock de la ressource, et décrémenter les fonds selon la facture retournée par le grossiste. La demande aux grossistes se fait via la fonction `buy(...)`, une usine ne va acheter que les objets dont elle a besoin pour sa production et aucun autre. Lorsqu'une usine a besoin de plusieurs objets, il faut en priorité acheter ceux que l'usine n'a pas en stock.
— **const** `std::vector<ItemType> ressources_needed` est la liste des ressources qu'une usine utilise dans sa production.

3.3 Les grossistes

Un grossiste est simulé par un thread dont le comportement est décrit ci-dessous. Il vous faudra implémenter la méthode `buy(...)` qui permet aux usines d'acheter des objets aux grossistes ainsi que le reste des méthodes décrites ci-dessous.

— La méthode **void** `run()` est la fonction du thread des grossistes, elle contient déjà un minimum de code fonctionnel. Le thread va simplement acheter des ressources selon son humeur et attendre un délai aléatoire.
— La méthode **void** `Wholesale::buy_ressources()` est la routine d'achat de ressources mises à disposition par les mines et les usines. Le grossiste va vouloir en tout temps se procurer des ressources afin de pouvoir les revendre. Les envies du grossistes sont aléatoires, il va choisir un vendeur parmi ceux dans sa liste de revendeurs et va leur demander ce qu'ils ont à vendre, il va ensuite choisir une quantité entre 1 et 5 et vouloir l'acheter à ce revendeur (il se peut donc que le revendeur n'aie pas le stock disponible). Il faudra ici implémenter l'achat, en utilisant la méthode `buy(...)`. Attention il faudra regarder que le grossiste ait assez d'argent. S'il a assez d'argent pour s'affranchir de l'achat, il faudra alors déduire des fonds le montant total et augmenter les stocks.

Une partie du code vous est déjà donnée pour cette méthode, l'intention d'acheter des objets est déjà émise par le grossiste et affichée dans une console, cela vous permet aussi de voir comment on peut afficher des messages dans les consoles des différents acteurs. Il faudra implémenter le reste de la méthode.

Ne pas hésitez a observer les fichiers .h de chacune des classes afin de pouvoir voir les méthodes, attributs mis à disposition. Il peut être intéressant aussi de regarder les autres fichiers.

Gestion de la concurrence

Dans les différentes méthodes il y aura des sections critiques liées à la gestion de stocks, les fonds (money), ou la production d'objets, par exemple les stocks sont mis à jour lors d'achats ou de productions. Il faudra exploiter les *mutex* fournis par la librairie `<pcosynchro/pcomutex.h>` afin de protéger les sections critiques et d'assurer que tout se passe bien et que la simulation ne génère pas de l'argent ou des objets de "nulle-part", ni n'en fasse disparaitre et éviter les *dead-locks*.

Gestion de la fin de la simulation

Il s'agira ici d'implémenter la fin de la simulation, qui va permettre d'arrêter de manière propre tous les threads. Il faut donc que les threads puissent s'arrêter afin qu'ils soient "*join*" par le thread principal. Une fois les threads arrêtés le thread principal va faire un calcul basé sur les ressources afin de regarder si les totaux sont cohérents et l'afficher.

L'arrêt de la simulation est initiée par la fonction `void Utils::endService()` dans `utils.cpp` qui est vide et qu'il faudra implémenter. Cette fonction est appelée lorsqu'on appuie sur *enter* dans le terminal associé à l'application (n'oubliez pas de cocher "Run in terminal" dans *Projects/Run*). Le résultat du calcul du total des fonds de la simulation n'est affiché que lorsque tous les threads ont été *join*.

4 Travail à rendre

Note : Dans le code source se trouvent des mots clés `TODO` en commentaires qui peuvent vous aiguiller sur où il faut ajouter du code. Vous pouvez ajouter des méthodes (fonctions), attributs membres (variables) aux classes etc. mais ne modifiez pas les signatures de méthodes données ni l'architecture proposée, **ne pas non plus créer de nouveaux fichiers**, merci.

Remplir les en-têtes des fichiers modifiés avec vos prénoms, noms.

- Les modalités du rendu se trouvent dans les consignes qui vous ont été distribuées.
- **Utilisez le script de rendu fourni** pour préparer votre rendu à remettre sur Cyberlearn.
- La description de l'implémentation, ses différentes étapes, la manière dont vous avez vérifié son fonctionnement et toute autre information pertinente doivent figurer dans votre **mini rapport**.
- Inspirez-vous du barème de correction pour connaître là où il faut mettre votre effort.
- Vous pouvez travailler en équipe de deux personnes au plus.

5 Barème de correction

Conception (rapport)	40%
Exécution (fonctionnement, gestion correcte de la concurrence)	40%
Documentation, commentaires, bonnes pratiques (code)	20%