HEIG-VD | PCO - Laboratoire 3

Gestion d'accès concurrents

Auteurs : Harun Ouweis & Christen Anthony
Date : 08.11.2023

Introduction

Le logiciel qui nous est fourni dans le cadre de ce laboratoire simule des interactions entre différentes entitiés. Nous retrouvons des extracteurs (`Extractor`) de matières premières, des grossisites (`Wholesaler`) et des usines (`Factory`). Chaque entité possède un rôle à jouer au sein de l'écosystème ainsi que certaines interactions possibles avec d'autres entités. Les instances des entités listées ci-dessus possèdent des ressources qui peuvent s'avérer être critiques lors de l'utilisation du multi-threading. Lorsque des interactions ont lieu entre différentes instances, leurs ressources peuvent être impactées.

Fonctionnalités clés du logiciel :

- Multi-threading :
  Chaque instance d'une entité est représentée par un thread. Les entités travillent de manière automne pour certaines opérations (extraction/fabrication) et de manière dépendante des autres instances pour d'autres (achat/vente). Ainsi, pour un extracteur (`Extractor`), des opérations telles que l'extraction et la vente de ressources peuvent s'exécuter simultanément.
- Gestion des ressources :
  La simulation tient compte des ressources disponibles et des fonds de l'entreprise pour la prise de décision concerant les opérations à effectuer. Comme expliqué préceddememnt, certaines de ces ressources peuvent être critiques lors d'accès concurrents. Il est nécessaire de les gérer de manière sécurisée afin de conserver un écosystème cohérent.
- Interface de suivi :
  Fournit une vue d'ensemble des indicateurs économiques de l'entreprise (trésorerie et stocks) en temps réel.
- Arrêt propre et calculs de vérification :
  Les threads sont stoppés de manière propre afin d'arrêtrer la simulation dans un état cohérent. Une fois cette dernière stoppée, un calcul du total des fonds qui ont transités dans le système est effectué afin de s'assurer de l'intégrité des données. Il est nécessaire qu'il y ait le même

montant total d'argent au début de la simulation qu'à la fin (pas de création de valeur ou de bénéfices durant la simulation).

Objectif

Nous avons pour objectif d'implémenter la logique "métier" de chaque entité permettant à la simulation de fonctionner ainsi que de protéger les ressources critiques des différentes instances des entités afin de conserver un état cohérent de l'écosystème durant toute la durée de la simulation. Afin de parvenir à cela, nous devons protéger les accès concurrents effectués. Il nous est également demandé de terminer le programme de manière "propore" en stoppant la simulation dans un état cohérent.

Afin de répondre au cahier des charges qui nous a été fourrni, nous avons identifié les potentiels problèmes suivants lors de l'utilisation du multi-threading dans notre simulation :

- Perte ou création d'argent
- Stocks se trouvant dans un état incohérent

Implémentation de notre solution

Nous avons commencé ce laboratoire en analysant le code fournit afin de se l'approprier et d'identifier les modifications et ajouts de code à effectuer. Cette première phase d'analyse nous a également permis d'identifier les variables à considérer lors de l'utilisation du multi-threading

Les variables relatives aux stocks et à la trésorerie (`money` et `stock`) sont les variables partagées que nous avons identifées comme étant à protéger. Les variables relatives au nombre d'employés payés ou d'éléments créés (`nbExtracted` et `nbBuilt`) sont les variables que nous avons identifiées comme n'ayant pas besoin d'être protégées car elles ne sont pas accédées (lues ou écrites) de manière concurrente.

Seller

Il s'agit d'une classe abstraite dont hérite chacune des classes représentant les différentes entitiés (`Extractor` , `WholeSaler` , `Factory`) de la simulation. C'est dans cette classe que les ressources partagées à protéger que nous avons identifiées se trouvent (`money` et `stock`). Nous avons donc décidé d'y ajouter un unique mutex `resourcesProtector` . En procédant ainsi, chaque instance d'une classe héritant de `Seller` possède son propre mutex permettant de protéger ses ressources critiques lors d'accès concurrents.

Extractor

Classe héritant de `Seller`. Son rôle est d'extraire des matière premières et de les vendre aux grossistes. Les ressources critiques d'une instance de cette classe (`money` et `stock`) sont potentiellement impactées lors des opérations suivantes :

- Extraction (`run()`) : Afin d'être en mesure d'extraire des matières premières, l'extracteur doit avoir les fonds nécessaires pour payer un employé. Il faut donc consulter la ressource critique `money`. Si l'extracteur dispose de fonds suffisants, il doit mettre à jour ses ressources critiques (`money` diminue et `stock` augmente).
- Vente (`trade()`) : Pour que la vente puisse se faire, l'extacteur doit posséder suffisament de stock pour répondre à la demande. Si c'est le cas, ses ressources critiques doivent être mises à jour (`money` augmente et `stock diminue`).

Etant donné que la méthode `run()` est exécutée par le thread représentant l'extracteur et que la méthode `trade()` peut être appelée à tout moment par le thread d'une instance de `WholeSaler`, les variables `money` et `stock` peuvent être accédées simultanément et doivent donc être protégées par le mutex `resourcesProtector` provenant de la classe `Seller`. Vous trouverez dans notre code des commentaires indiquant le début et la fin des sections critiques protégées.

```
resourcesProtector.lock(); // Début S.C.
if (money < minerCost) {
    resourcesProtector.unlock(); // Fin S.C.
    /* Pas assez d'argent */
    /* Attend des jours meilleurs */
    PcoThread::usleep(1000U);
    continue;
}
```

La portion de code ci-dessus est issue de la méthode `run()`. Elle permet de vérifier que l'extracteur dispose des fonds nécessaires afin de payer un employé. En l'état actuel des choses, bien que la variable `money` puisse être accédée de manière concurrente, il n'est pas obligatoire de la protéger car seule la méthode `trade()` pourrait la modifier après que la condition ait été vérifiée. Comme `trade()` ne fait qu'augmenter `money` si elle est amenée à la modifier, nous ne risquons pas de générer un accès concurrent problématique. Nous avons tout de même choisi de protéger cette vérification pour conserver une certaine cohérence avec le reste de notre code et afin de prévenir tout potentiel problème futur si du code venait à être ajouté afin de permettre aux extracteurs d'acheter des objets ou d'extraire des ressources de manière concurrente (plusieurs employés travaillant simultanément).

La variable `nbExtracted` n'est pas protégée car il ne s'agit pas d'une ressource accédée de manière concurrente. En effet, lors de la simulation, cette variable est modifiée et consultée uniquement par la méthode `run()` qui est elle-même exécutée uniquement par le thread représentant l'extracteur.

Factory

Classe héritant de `Seller`. Son rôle est de construire des objets après avoir acheté des ressources aux grossistes (`WholeSaler`) et de vendre les objets créés aux grossistes intéressés. Les ressources critiques d'une instance de cette classe (`money` et `stock`) sont potentiellement impactées lors des opérations suivantes :

- Achat des ressources nécessaires (`orderResources()`) : Pour pouvoir construire des objets, l'usine doit posséder les ressources nécessaires en quantités suffisantes (`verifyResources()`). Elle se fournit auprès de différents grossistes (`WholeSaler`) en appelant leur méthode (`trade()`) à condition qu'elle dispose des fonds (`money`) nécessaires. Lors d'un achat, l'usine doit mettre à jour ses ressources critiques (`money` diminue et `stock` augmente).
- Construction (`buildItem()`) : Afin d'être en mesure de construire un objet, l'usine doit disposer des fonds nécessaires pour payer un employé. Il faut donc consulter la ressource critique `money`. Si l'usine dispose de fonds suffisants, elle doit mettre à jour ses ressources critiques (`money` diminue, `stock` augmente por l'objet créé et diminue pour les ressources utilisées).
- Vente des objets construits (`trade()`) : Pour que la vente puisse se faire, l'usine doit posséder suffisament de stock pour répondre à la demande. Si c'est le cas, ses ressources critiques doivent être mises à jour (`money` augmente et `stock diminue`).

```
resourcesProtector.lock(); // Début S.C.
if (money < getCostPerUnit(resource) * qty) {
    resourcesProtector.unlock(); // Fin S.C.
    continue;
}
resourcesProtector.unlock(); // Fin S.C.

int bill = wholesaler->trade(resource, qty);

if (bill == 0) {
    continue;
}

resourcesProtector.lock(); // Début S.C.
money -= bill;
++stocks[resource];
resourcesProtector.unlock(); // Fin S.C.
```

Le code ci-dessus est issu de la fonction `orderResources()`. Avant de vérifier si l'usine dispose des fonds suffisants pour un achat de matières premières, le mutex (`resourcesProtector`) est pris (`lock()`) pour éviter toute modification de la variable `money`. Si les fonds sont insuffisants, le mutex est libéré (`unlock()`). En cas de fonds suffisants, le mutex est également libéré pour éviter un interblocage lors de l'appel à la méthode `trade()` du grossiste (voir figure 1). Bien qu'il puisse y avoir un laps de temps important avant que l'usine récupère le mutex pour mettre à jour ses ressources, cela ne pose pas de problème de fonctionnement. En effet, en cas de blocage lors du deuxième `lock()`, cela signifie qu'un achat (`trade()`) auprès de l'usine est en cours et que la variable `money` pourrait être modifiée. Cependant, cela n'est pas problématique, car si la variable `money` est modifiée, c'est UNIQUEMENT pour l'augmenter. L'usine aurait donc plus d'argent qu'avant d'avoir vérifié si elle disposait des fonds nécessaires pour acheter des matières premières au grossiste. Ainsi, il n'y a aucun risque de manquer d'argent et donc de générer un état incohérent.

Bien que cette implémentation puisse potentiellement différer la diminution de la trésorerie de l'usine après avoir augmenter celle du grossiste, nous avons décidé de choisir cette solution car elle ne nuit pas au fonctionnement du programme, ne pose pas de problème d'accès concurrent et évite un risque d'interblocage (voir figure 1).

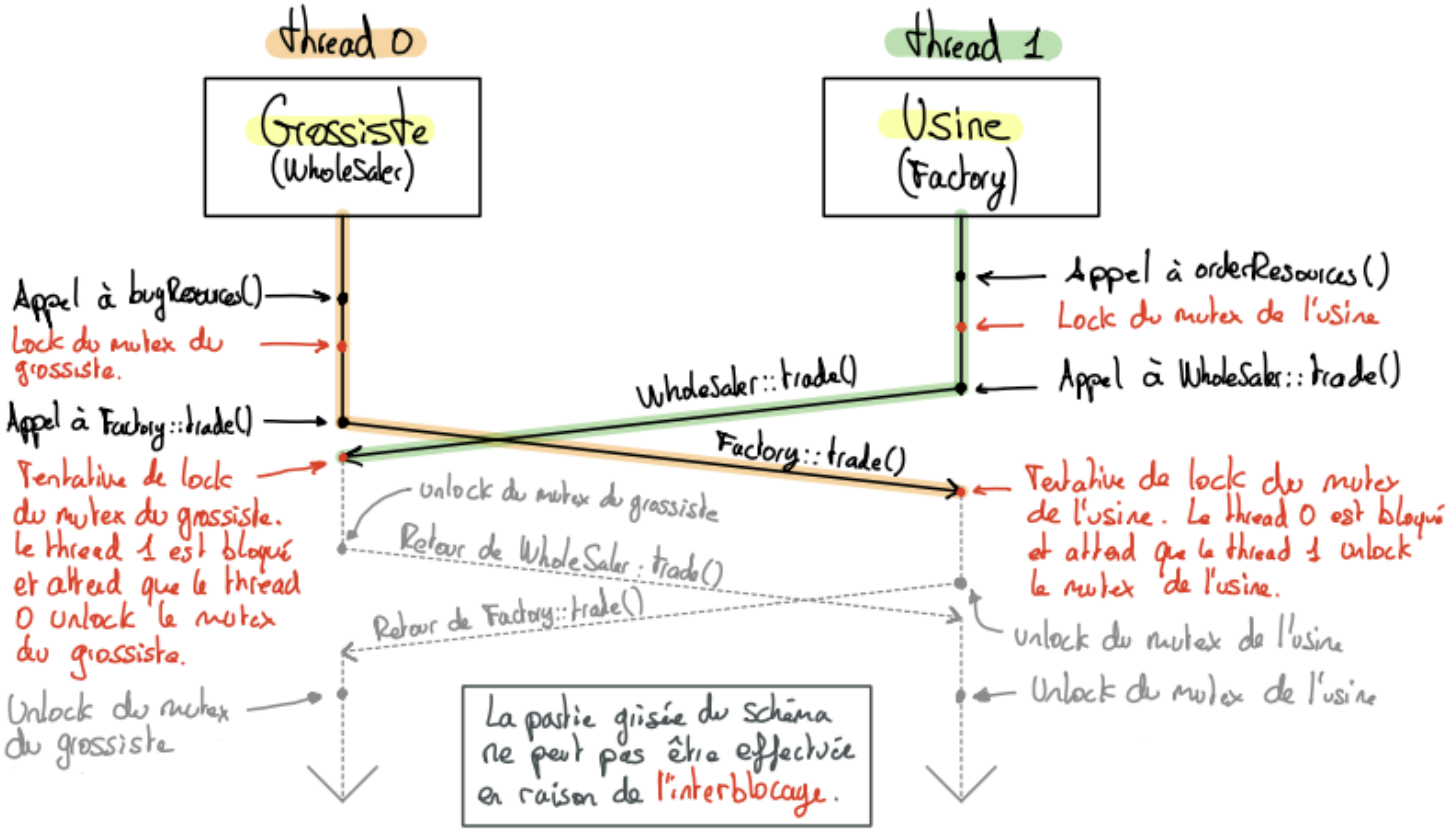

Figure 1 : Schéma du risque d'interblocage

Il s'agit d'une implémation ou le mutex n'est pas libéré avant d'appeler la méthode `WholeSaler::Trade()` depuis la méthode `Factory::OrderResources()`. Comme expliqué précédemment, nous avons pris en compte ce risque lors de notre implémentation et avons fait en sorte qu'un interblocage ne soit pas possible.

La variable `nbBuild` n'est pas protégée car il ne s'agit pas d'une ressource accédée de manière concurrente. En effet, lors de la simulation, cette variable est modifiée et consultée uniquement par la méthode `buildItem()` qui est elle-même exécutée uniquement par le thread représentant l'usine.

La méthode `verifyResources()` qui est appelée afin de déterminer si l'usine doit commander des ressources (`orderResources()`) ou construire un objet (`buildItem()`) ne nécessite pas d'être protégée car seules les ressources nécessaires à la construction (`resourcesNeeded`) sont accédées. Comme ces ressources ne peuvent pas être achetées, nous ne risquons pas d'accès concurrent.

Wholesaler

Classe héritant de `Seller` . Les grossistes jouent un rôle central. Ils font office d'intermédiaires entre les extracteurs de matières premières (`Extractor`) et les usines (`Factory`). Les ressources critiques d'une instance de cette classe (`money` et `stock`) sont potentiellement impactées lors des opérations suivantes :

- Achat de ressources (matières premières ou objets créés par les usines) (`buyResources()`) : Pour acheter des ressources, le grossiste doit disposer des fonds nécessaires (`money`). Une fois l'achat effectué, le grossiste doit mettre à jour ses ressources critiques (`money` diminue et `stock` augmente).
- Vente de ressources (matières premières ou objets créés par les usines) (`trade()`) : Pour que la vente puisse se faire, le grossiste doit posséder suffisament de stock pour répondre à la demande. Si c'est le cas, ses ressources critiques doivent être mises à jour (`money` augmente et `stock diminue`).

Concernant la méthode `buyResources()` la même structure/logique que celle de la méthode `Factory::orderResources()` a été implémentée. Nous avons également géré le risque d'interblocage de la même manière. Vous pouvez vous référer à la figure 1 se trouvant ci-dessus.

Utils

Afin d'arrêter la simulation dans un état cohérent lorsque l'utilisateur souhaite quitter le programme, nous avons implémenté la fonction `endService()` en utilisant les fonctionnalités offertes par la classe `PcoThread` . En effet, nous appeleons la méthode `requestStop()` sur chacun de nos threads, dont un pointeur a été stocké dans le vecteur `threads` , afin des les avertir que nous

souhaitons qu'ils s'arrêtent. Le thread peut alors vérifier si un tel signal a été émis en appeleant la fonction `stopRequested()`. Dans notre programme, lorsqu'un appel à la fonction `requestStop()` est effectué, le thread concerné termine son travail avant de s'arrêter. Il fini l'itération de la boucle `while(!PcoThread::thisThread()->stopRequested())` dans laquelle il se trouve. En d'autres termes, l'entité s'arrête de travailler une fois que sa tâche courante est terminée. Etant donné que chacun de nos threads représente une instance d'une entitié de la simulation, nous garantissons que chaque instance termine entièrement son travail afin de se trouver dans un état cohérent avant de quitter le programme.

Remarques

Comme expliqué dans certaines sections de ce rapport, nous avons protégé les accès en lecture de la variable `money` afin de se prémunir de tout risque de problème d'accès concurrent. Nous avons choisi de nous conformer aux précautions à prendre mises en avant durant les cours. Comme il s'agit de "simples" vérifications (portions de code très petites) nous avons estimé que cela n'engendrerait pas de problèmes significatifs de performances.

Tests effectués

Pour ce laboratoire, il est passablement difficile d'effectuer des tests nous permettant de garantir que notre solution est correcte. Nous avons utiliser le rapport généré une fois la simulation terminée afin de nous assurer que l'argent total à la fin de la simulation correspond à l'argent total mis à disposition au début de celle-ci. Nous avons lancé la simulation en la faisant tourner durant des laps de temps courts, moyens et longs afin de voir comment elle réagissait. Tous nos tests sont passés avec succès !

De plus, nous avons décidé tester la simulation en retirant tous nos mutex ainsi que tous les `PcoThread::usleep()`. En faisant cela, nous avons pu constater des problèmes d'accès concurrents qui ont générés des incohérences dans la simulation. Le rapport de contrôle généré en fin de simulation a mis en évidence des problèmes de trésorerie. En re-testant notre code avec nos mutex et les `PcoThread::usleep()`, les problèmes décrits précédemment n'ont plus été constatés.

Conclusion

Ce laboratoire nous a permis de mettre en pratique la gestion d'accès concurrents. Nous avons pu nous rendre compte de l'importance de la protection des variables critiques que ce soit en lecture ou en écriture afin de ne pas générer de données/états incohérents pouvant avoir des répercussions catastrophiques.