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

Yann Thoma
Olivier Cuisenaire
Florian Vaussard

Laboratoire de Programmation Concurrente

semestre automne 2024

Gestion d'accès concurrents

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

Récupération du laboratoire : `retrieve_lab pco24 lab03`

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

Une épidémie s'est abattue sur une petite ville, entraînant une crise sanitaire majeure. Face à cette situation, l'État a mandaté le système hospitalier local pour prendre en charge un maximum de patients et leur a garanti le remboursement intégral de toutes leurs dépenses. Le but est de gérer efficacement les flux de patients et les besoins médicaux.

Le système de santé de cette ville repose sur l'interaction de quatre acteurs principaux :

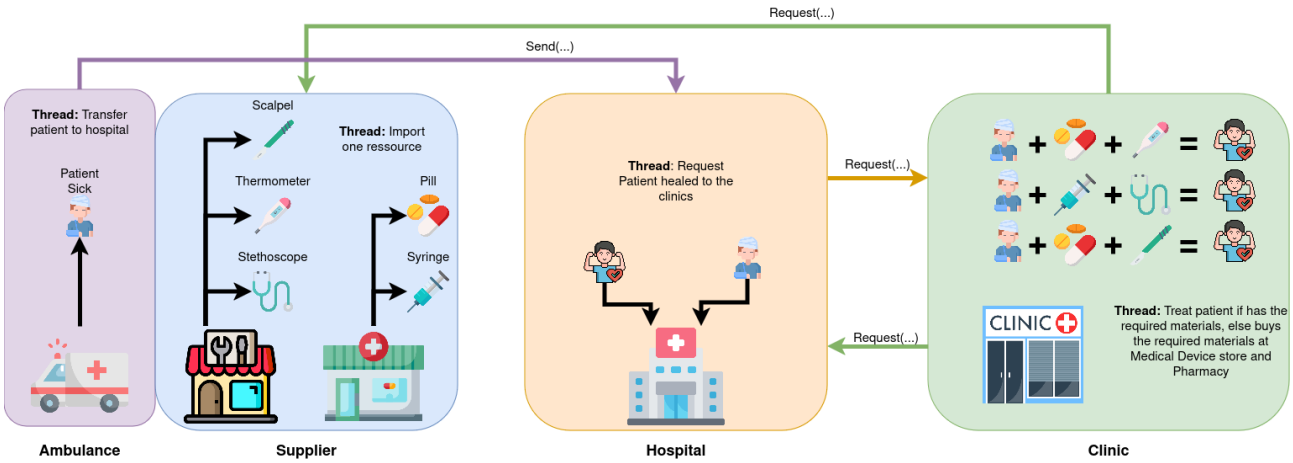
1. **Le système ambulancier** : Son rôle est d'acheminer le plus de patients possible vers les hôpitaux de la ville. Les ambulances transportent uniquement des patients malades et leur objectif est d'assurer une prise en charge rapide en fonction des places disponibles dans les hôpitaux.

2. **Les fournisseurs de médicaments et d'outils médicaux** : Ils importent de l'étranger des médicaments et des équipements médicaux indispensables au traitement des patients. Ils assurent l'approvisionnement en ressources critiques, telles que des pilules, des seringues, ainsi que des outils comme des stéthoscopes, des thermomètres et des scalpels.

3. **Les hôpitaux** : Ils jouent un rôle central en accueillant et en soignant les patients transportés par les ambulances. Les hôpitaux gèrent deux types de *stocks* : des patients malades en attente de soins et des patients soignés qui ont été traités et stabilisés.

4. **Les cliniques spécialisées** : Ces établissements se concentrent sur des traitements spécifiques nécessitant des outils médicaux particuliers. Leur stock est plus spécialisé et dépend de la nature des traitements qu'elles offrent. Elles ont besoin de ressources spécifiques pour traiter leurs patients de manière optimale.

L'objectif est de simuler ces interactions et de gérer les ressources et les stocks. Chaque acteur doit collaborer efficacement pour répondre aux besoins médicaux de la ville tout en évitant des erreurs et des pertes dans leur logistique ou comptabilité.



3 Travail à faire

Dans ce laboratoire, vous devrez implémenter la logique régissant les interactions entre les différents acteurs tout en prenant en compte les sections critiques. Il vous sera demandé de compléter les méthodes fournies pour les ambulances, fournisseurs, hôpitaux et cliniques, selon le comportement décrit ci-dessous, tout en gérant la concurrence. Le code associé se trouve dans les fichiers suivants :

- ambulance.cpp/h
- hospital.cpp/h
- clinic.cpp/h
- supplier.cpp/h

Dans le code source se trouvent des mots clés `TODO` en commentaires qui peuvent vous aiguiller sur les emplacements où il faut ajouter du code. **N'hésitez pas à observer les fichiers .h de chacune des classes afin de pouvoir voir les méthodes et attributs mis à disposition.** Il peut être intéressant aussi de regarder les autres fichiers.

Interface vendeur (Seller)

L'interface vendeur (Seller) est une classe abstraite dont héritent tous les acteurs de la simulation. Il est utile de comprendre son fonctionnement car certaines méthodes devront être implémentées dans les classes dérivées. Elle propose les méthodes et attributs suivants :

- `std::map<ItemType, int> getItemsForSale()` Cette méthode retourne une *map* des objets à vendre, associant le(s) type(s) d'objet(s) vendu(s) à une quantité disponible.
- `int request(ItemType what, int qty)` Cette méthode représente un achat. Elle prend en entrée le type de ressource que l'on souhaite acheter ainsi que la quantité. La valeur retournée est le prix de la transaction si celle-ci peut avoir lieu, ou 0 si ce n'est pas possible (manque de stock, objet non vendu, quantité négative, etc.). Si la méthode ne retourne pas 0, il faut mettre à jour les stocks et les fonds du vendeur (la gestion des fonds et stocks de l'acheteur se fait en dehors de cette méthode, par l'appelant).
- `int send(ItemType what, int qty, int bill)` Cette méthode représente une offre. Elle prend en entrée le type de ressource, sa quantité et le prix de ce que l'on souhaite envoyer. Si la transaction n'est pas possible (manque de place, fonds insuffisants, etc.), elle retourne 0. Sinon, un autre nombre est renvoyé pour confirmer que la transaction a été acceptée. Il faudra alors mettre à jour les stocks et les fonds de la cible (la gestion des fonds et stocks de celui qui propose se fait par l'appelant, en dehors de cette méthode).

— `std::map<ItemType, int> stocks;` Les stocks de l'instance (type, quantité), ne correspondent pas forcément aux objets en vente. Par exemple, une clinique ne vendra pas les outils (p.ex. scalpel) qu'elle utilise, mais uniquement les objets qu'elle "produit".

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

— `Seller* chooseRandomSeller(std::vector<Seller*>& sellers)` Cette méthode permet de choisir un vendeur (Seller) aléatoirement parmi une liste. Elle prend en paramètre une liste de vendeurs.

— `ItemType getRandomItemFromStock()` Cette méthode permet de choisir au hasard une ressource parmi les stocks de l'appelant.

— `ItemType chooseRandomItem(std::map<ItemType, int>& itemsForSale)` Cette méthode choisit aléatoirement un ItemType parmi une map donnée en paramètre.

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

3.1 Les Ambulances

Une ambulance est simulée par un thread qui exécute la méthode `run()`. Cette méthode contient déjà un minimum de code fonctionnel. L'ambulance va tenter d'envoyer ses patients au plus vite dans les hôpitaux. Les ambulanciers sont payés à chaque fois qu'un envoi est réussi.

Vous devrez principalement implémenter :

— La méthode `run()` qui exécute la routine d'un fournisseur.

— La méthode `sendPatient()` qui doit permettre d'envoyer des patients dans les hôpitaux. Dans cette méthode, vous devez également tenir à jour vos stocks et fonds si les hôpitaux peuvent prendre en charge les patients envoyés par l'ambulance.

3.2 Les Fournisseurs de ressources (Suppliers)

Un fournisseur est simulé par un thread qui exécute la méthode `run()`. Cette méthode contient déjà un minimum de code fonctionnel. Le fournisseur, s'il a assez d'argent, paiera un employé pour gérer l'import de la ressource et attendra que le travail soit terminé.

Vous devrez principalement implémenter :

— La méthode `run()` qui exécute la routine d'un fournisseur.

— La méthode `request(...)` qui permet à un autre thread d'essayer d'effectuer un achat.

3.3 Les Hôpitaux

Un hôpital est simulé par un thread qui exécute la méthode `run()`. Cette méthode contient déjà un minimum de code fonctionnel. Le thread va uniquement demander aux cliniques si des patients ont été soignés pour les récupérer et leur permettre de retourner à leur vie normale. Pour qu'un patient puisse être libéré, il devra occuper un lit pendant 5 itérations de la routine de l'hôpital (similaire à 5 jours de repos) afin de se remettre de son opération. Les hôpitaux disposent d'un nombre limité de lits.

Vous devrez principalement implémenter :

— La méthode `run()` qui exécute la routine d'un hôpital.

— La méthode `send(...)` qui permet à un autre thread de lui proposer des patients si l'hôpital peut les prendre et payer les frais.

— La méthode `request(...)` qui permet à un autre thread de demander des patients malades.

— La méthode `transferPatientFromClinic()` qui doit permettre aux hôpitaux de récupérer les patients soignés des cliniques, si possible.

— Les hôpitaux paient leurs infirmiers à chaque fois qu'une transaction est effectuée.

3.4 Les Cliniques spécialisées

Une clinique est simulée par un thread qui exécute la méthode `run()`. Cette méthode contient déjà un minimum de code fonctionnel. La clinique vérifie si elle possède les ressources nécessaires pour traiter un patient malade. Si oui, elle traite le patient ; sinon, elle essaie d'acheter ou de transférer les ressources depuis les fournisseurs et les hôpitaux.

Le vecteur `const std::vector<ItemType> resourcesNeeded` contient la liste des ressources qu'une clinique utilise pour le traitement.

Vous devrez principalement implémenter :

— La méthode `run()` qui exécute la routine d'une clinique.

— La méthode `request(...)` qui permet à un autre thread de lui demander des patients soignés.

— La méthode `orderResources()` qui permet aux cliniques d'acheter les ressources auprès des fournisseurs et de demander des patients malades aux hôpitaux. La liste des ressources est indiquée dans le diagramme de la page 2.

— La méthode `treatPatient()` qui permet aux cliniques de traiter un patient malade. Pour traiter un patient, il faudra payer un employé, comme pour les fournisseurs. Les fonctions suivantes sont disponibles dans seller.h :

  — `EmployeeType getEmployeeThatProduces(ItemType item)` permet de savoir quel type d'employé peut produire l'objet.

  — `int getEmployeeSalary(EmployeeType employee)` permet de savoir combien il faut payer l'employé pour produire une unité de l'objet.

  Il faudra également garder un historique du nombre de patients traités par les cliniques.

Note : Pour toutes les entités des valeurs de contrôle doivent être incrémentées pour savoir à chaque fois que l'entité à payé un salaire.

3.5 Gestion de la concurrence

Dans les différentes méthodes il y aura des sections critiques liées à la gestion des stocks, des fonds (money) ou de 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*.

3.6 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 (trésorerie, stocks et salaires payés) afin de regarder si les totaux sont cohérents et les 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 lorsque l'on ferme la fenêtre de l'application. Le résultat du calcul du total des fonds de la simulation est alors affiché dans une fenêtre de dialogue que lorsque tous les threads ont été *join*.

3.7 Contrôle

Pour vous aider à suivre l'exécution de votre programme en mode graphique, des attributs de classe sont mis à votre disposition dans chaque classe. Vous devrez les incrémenter au moment approprié.

3.7.1 Ambulance

L'attribut `nbTransfer` doit être incrémenté chaque fois qu'un ambulancier est payé, c'est-à-dire chaque fois qu'un hôpital effectue un transfert.

3.7.2 Supplier

L'attribut `nbSupplied` doit être incrémenté chaque fois qu'un fournisseur est payé, c'est-à-dire à chaque fois qu'une nouvelle ressource est importée.

3.7.3 Clinic

L'attribut `nbTreated` doit être incrémenté chaque fois qu'un médecin est payé, c'est-à-dire chaque fois qu'un patient est soigné.

3.7.4 Hospital

Deux attributs sont à prendre en compte : `nbHospitalised` et `nbFree`.

— `nbHospitalised` : cet attribut doit être incrémenté chaque fois qu'une infirmière est payée, c'est-à-dire à chaque fois qu'un patient est admis à l'hôpital (que ce soit par le biais d'un transfert ou par le retour de cliniques).

— `nbFree` : cet attribut doit être incrémenté chaque fois qu'un patient est libéré de l'hôpital, c'est-à-dire dès qu'un patient a passé 5 jours de convalescence. Il est important de conserver son historique de passage dans l'hôpital afin de ne pas perdre le compte.

4 Test

Tester la concurrence dans votre programme via l'interface graphique peut s'avérer complexe. Pour faciliter le test de votre programme, un fichier `test_main.cpp` vous est fourni. Ce fichier utilise Google Test (gtest) pour écrire et exécuter des tests unitaires. Un exemple y est inclus, qui teste les hôpitaux de manière simple.

Ces tests contournent l'interface graphique, permettant ainsi une exécution plus rapide et facilitant la génération de conditions d'accès concurrentiel aux ressources partagées.

Voici un example d'une exécution sans protection :

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from SellerTest
[ RUN      ] SellerTest.TestHospitals
/home/bruno/dev/PCO/pco-2018/labs/24-25/labo3_hospital/ressources/solution/src/tests_main.cpp:66: Failure
Expected equality of these values:
  endFund
    Which is: 23084
  initialFund
    Which is: 20000
/home/bruno/dev/PCO/pco-2018/labs/24-25/labo3_hospital/ressources/solution/src/tests_main.cpp:67: Failure
Expected: (hospital.getNumberPatients()) >= (0), actual: -1 vs 0
/home/bruno/dev/PCO/pco-2018/labs/24-25/labo3_hospital/ressources/solution/src/tests_main.cpp:68: Failure
Expected: (hospital.getNumberPatients()) <= (maxBeds), actual: -1 vs 35
[  FAILED  ] SellerTest.TestHospitals (3 ms)
[----------] 1 test from SellerTest (3 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (4 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] SellerTest.TestHospitals

 1 FAILED TEST
```

5 CMake

Si vous préférez ne pas utiliser un IDE pour compiler votre projet et souhaitez garder votre répertoire de travail propre, vous pouvez utiliser CMake en ligne de commande. Voici les étapes à suivre pour compiler votre projet sans "polluer" votre dossier de travail principal :

1. Créez un dossier build
   Dans le même répertoire où se trouve votre fichier CMakeLists.txt, créez un nouveau dossier appelé build qui contiendra tous les fichiers générés lors de la compilation.
   `mkdir build`
2. Générez les fichiers de build avec CMake
   Ensuite, déplacez-vous dans le dossier build et exécutez CMake en pointant vers le répertoire parent (où se trouve CMakeLists.txt).
   `cd build`
   `cmake ..`
3. Compilez le projet avec Make
   Une fois les fichiers générés, il suffit de lancer make pour compiler le projet.
   `make`

Voilà ! Votre projet est maintenant compilé, et tous les fichiers générés (fichiers objets, exécutables, etc.) restent confinés dans le dossier build, gardant ainsi votre répertoire de travail propre.

6 Travail à rendre

Note : Vous pouvez ajouter des méthodes (fonctions), attributs membres (variables) aux classes 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**.
- Vous pouvez travailler en équipe de deux personnes au plus.

7 Barème de correction

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