# Hopital concurrent

Auteurs : Christophe Künzli, Léonard Jouve

## Introduction au problème

On souhaite au travers d'une application de gestion d'inventaire pour des hopitaux gérer des accès concurrents à des ressources partagées entre threads.
Pour celà, nous allons utilier des mutex afin de faire de l'exclusion mutuelle.

## Choix d'implémentation

### Seller

Nous avons ajouté le mutex responsable de limiter les accès concurrents aux ressources critiques de chaque instance des objet héritants de Seller.

### Ambulance

- Dans la méthode `sendPatient`, nous avons choisi d'essayer d'envoyer le nombre maximum possible de patients aux hopitaux.
  La méthode `send` de Hopital nous renvoie le nombre de patients acceptés.
- Il n'y a pas de problème de concurrence dans Ambulance, car les ressources internes et méthodes d'une instance d'Ambulance sont accédées uniquement par cette même instance.

### Clinic

- La méthode `request` doit uniquement gérer des items de type PatientHealed demandés par les hopitaux.
  On limite la quantité demandée à la quantité de patients guéris dans la clinique.
  Cette méthode a une section critique car elle accède et modifie les stocks et l'argent de la clinique et qu'elle est appelée par les hopitaux qui sont executés par d'autres threads.
- Dans la méthode `orderRessources`, la clinic commande les outils nécéssaires au soin d'un patient incluant le patient lui même si elle n'en possède pas déja un.
  Cette méthode a également une section critique lors des accès à money et stocks. Ses accès sont donc protégés par son mutex.
- La méthode `treatPatient` consomme les ressources nécéssaires au soin d'un patient, met à jour les stocks et l'argent en payant un docteur.
  Les accès concurrents sont contrôlés par le mutex.

### Hospital

- La méthode `send` permet à une ambulance d'envoyer des patients à l'hopital. Elle vérifie que l'hopital a de la place pour les patients et renvoie le nombre de patients acceptés.
  Cette méthode a une section critique lors des accès aux stocks de patients, nombre de lits occupés(currentBeds), nombre total de patient hospitalisé(nbHospitalised) et à money. L'accès à ces ressouces doit être protégé par le mutex, car elles sont accédées et modifiées par d'autres threads via les autre méthodes de Hospital.
- La méthode `transferPatientsFromClinic` permet de transférer des patients guéris de la clinique à l'hopital. Elle demande à la clinique de lui donner des patients guéris et les ajoute à ses stocks de patients.
  Cette méthode a une section critique lors des accès aux stocks de patients, nombre de lits occupés(currentBeds), nombre total de patient hospitalisé(nbHospitalised) et à money. L'accès à ces ressouces doit être protégé par le mutex.
- La méthode `request` permet aux cliniques d'acheter des patients malades. Elle retourne le nombre de patients achetés.
  Elle a une section critique lors des accès aux stocks de patients, nombre de lits occupés(currentBeds) et à money. L'accès à ces ressouces doit être protégé par le mutex.
- La méthode `freeHealedPatient` permet de libérer tous les patients guéris qui ont fini leurs jours de convalescence. Elle a une section critique lors des accès aux stocks de patients et nombre de lits occupés(currentBeds).

- Nous avons ajouté l'attribut `patientsRecovering` afin de tenir le compte des jours restants pour que les patients guéris puissent être libérés. C'est un tableau de 5 int dont la valeur représente un nombre de patients et l'index représente le nombre de transaction avant leur libération.
  La méthode `freeHealedPatient` s'occupe de décaler les valeurs du tableau afin de diminuer le nombre transations avant la libération des patients guéris.
  La méthode `transferPatientsFromClinic` incrémente la dernière valeur du tableau du nombre de patients envoyés par la clinique.

#### Supplier

- La méthode `request` permet aux autres objets de tenter d'acheter des outils. Elle vérifie que l'objet demandé fait bien partie des objets vendus par le fournisseur, limite la quantité demandée à la quantité disponible et renvoie le nombre d'objets acheté. Elle met à jour les stocks et l'argent en conséquence.
  Comme pour Clinic et Hospital, cette méthode a une section critique lors des accès à money et stocks. Ses accès sont donc protégés par son mutex.
- La routine `run` contient le code permettant de générer des objets à vendre aléatoirement.
  Comme elle accède et modifie les stocks et l'argent, elle doit être protégée par le mutex.

#### Fin d'execution

Pour terminer la simulation proprement, nous avons implémenté le corps de la méthode `Utils::endService()` qui parcours les threads et leur envoie un signal d'arrêt.

Dans les routines de chaque thread, nous avons ajouté dans la boucle une vérification si le signal d'arrêt a été envoyé. Si c'est le cas, on sort de la boucle et on termine le thread.

## Tests effectués

Afin de tester notre implémentation, nous avons essayé différents scénarios et vérifié que chacun possédait en fin d'exécution le bon nombre de patients et d'argent: - attendre que les hopitaux et cliniques n'aient plus d'argent et que plus aucun intervenant ne puisse faire de transactionle - arrêter le programme avant que toutes les transactions possibles ne soient effectuées

Nous avons également vérifié que les différents intervenants ne dépensent pas plus d'argent qu'ils n'en possèdent ou encore que les stocks des suppliers ne descendent pas en dessous de 0.