Laboratoire PCO - Gestion de ressources par moniteurs

Auteurs : Perret Jonatan et Marcuard Adrien

Date : 8 décembre 2025

Cours : Programmation Concurrente - Semestre automne 2025

Introduction au problème

Ce laboratoire simule un système de partage de vélos dans une ville où des citadins se déplacent entre plusieurs sites équipés de bornes. Le système doit gérer de manière synchronisée les actions de plusieurs acteurs concurrents en utilisant des **moniteurs de Mesa**.

Acteurs du système

Les habitants (Person) : Chaque habitant suit un cycle répétitif :

- Attend qu'un vélo de son type préféré devienne disponible au site courant et le prend
- Se déplace en vélo vers un autre site choisi aléatoirement
- Attend qu'une borne devienne libre au site de destination et y dépose son vélo
- Se déplace à pied vers un autre site
- Recommence le cycle

Chaque habitant a une préférence pour un type de vélo spécifique (VTT, Route ou Gravel) définie à sa construction et n'utilise que ce type.

L'équipe de maintenance (Van) : Une camionnette effectue des tournées régulières pour rééquilibrer les vélos entre les sites et le dépôt. Elle peut transporter jusqu'à 4 vélos simultanément et priorise les types manquants lors des dépôts.

Les stations (BikeStation) : Chaque site possède un nombre fixe de bornes (B = 6 par défaut). Les stations doivent gérer l'accès concurrent des habitants et de la camionnette de manière thread-safe.

Contraintes et paramètres

- **Nombre de sites** : S = 8 sites + 1 dépôt (NB*SITES*TOTAL = 9)
- **Bornes par site** : B = 6 (BORNES)
- **Nombre d'habitants** : 10 personnes (NBPEOPLE)
- **Vélos totaux** : V = 35, vérifiant V ≥ S(B-2) + 3 = 8×4 + 3 = 35
- **Capacité camionnette** : 4 vélos (VAN_CAPACITY)
- **Types de vélos** : 3 types (VTT, Route, Gravel)

Initialement, chaque site contient B-2 = 4 vélos, et le reste (3 vélos) est au dépôt.

Problèmes de concurrence à résoudre

- **Accès concurrent aux stations** : Plusieurs habitants et la camionnette peuvent vouloir accéder simultanément à une même station
- **Gestion de la capacité** : Respecter le nombre maximum de bornes sans dépassement
- **Ordre d'attente (FIFO)** : Les habitants doivent être servis selon leur ordre d'arrivée pour un type de vélo donné
- **Éviter les interblocages** : Garantir qu'aucune situation de deadlock ne peut survenir
- **Terminaison propre** : Pouvoir arrêter la simulation sans laisser de threads bloqués

Choix d'implémentation

Modélisation par moniteur de Mesa

La classe **BikeStation** implémente un moniteur de Mesa qui centralise toute la synchronisation. Chaque station protège ses ressources (vélos et bornes) par un mutex unique et utilise des variables de condition pour gérer les attentes.

Séparation des vélos par type - Optimisation clé

Choix structurel : L'utilisation d'un tableau de deques (`std::array, Bike::nbBikeTypes>`) plutôt qu'une unique file commune est un choix d'implémentation fondamental.

Justification :

- **Ordre FIFO par type garanti** : Conformément au cahier des charges, lorsque plusieurs habitants attendent le même type de vélo, ils doivent être servis selon leur ordre d'arrivée. En séparant les vélos par type et en utilisant des deques (FIFO), on garantit naturellement cette propriété.
- **Notifications ciblées avec `notifyOne()`** : En associant une variable de condition par type (`bikeAdded[type]` , `bikeRemoved[type]`), on peut utiliser `notifyOne()` au lieu de `notifyAll()` . Lorsqu'un vélo de type VTT est ajouté, seul un habitant attendant un VTT est réveillé, et non tous les habitants.
- **Éviter les réveils inutiles** : Sans cette séparation, avec un `notifyAll()` global, chaque ajout de vélo réveillerait potentiellement 10 threads dont 9 se rendormiraient immédiatement après avoir constaté que ce n'est pas le bon type. Cela génèrerait des changements de contexte coûteux et dégraderait les performances.
- **Respect du moniteur de Mesa** : La structure avec variables de condition par type permet d'implémenter correctement le pattern de Mesa avec des boucles `while` et des attentes ciblées.

Mécanisme de protection des ressources critiques

Mutex unique : Un seul `PcoMutex` protège l'ensemble des données d'une station (vélos stockés, compteurs). Toute opération de lecture ou modification acquiert ce mutex, garantissant l'exclusion mutuelle.

Sections critiques : Les méthodes `putBike()` et `getBike()` suivent le schéma classique du moniteur de Mesa :

```
mutex.lock()

while (condition non satisfaite && !shouldEnd)

    conditionVariable.wait(&mutex)

if (shouldEnd)
```

```
    return


// section critique


conditionVariable.notifyOne()


mutex.unlock()
```

Ce schéma évite les attentes actives (busy-wait) : les threads se bloquent dans `wait()` sans consommer de CPU.

Variables de condition pour l'attente bloquante

Le système utilise deux vecteurs de variables de condition (`std::vector`) **par type de vélo** pour permettre aux threads de s'endormir (attente passive) plutôt que de tourner en boucle (attente active) :

- **bikeAdded[type]** : Signale l'ajout d'un vélo d'un type donné. Seuls les usagers en attente de ce type spécifique s'y bloquent.
- **bikeRemoved[type]** : Signale le retrait d'un vélo d'un type donné, libérant potentiellement une borne. Les usagers souhaitant déposer un vélo de ce type s'y bloquent lorsque la station est pleine.

Chaque vecteur est initialisé avec `Bike::nbBikeTypes` éléments (3 types), offrant ainsi une granularité fine (3 types × 2 conditions = 6 variables de condition par station). Cette structure permet une synchronisation efficace avec des notifications ciblées plutôt que des réveils massifs. Les threads bloqués ne consomment aucun CPU et sont réveillés uniquement lorsque leur condition spécifique change.

Gestion de la capacité et des attentes

putBike() : Lorsqu'un habitant veut déposer un vélo sur un site plein (nbBikes() ≥ BORNES), il se bloque sur `bikeRemoved[bikeType]` jusqu'à ce qu'une borne se libère. La notification est faite par `getBike()` ou `getBikes()` lors du retrait d'un vélo.

getBike() : Lorsqu'un habitant attend un vélo d'un type spécifique non disponible, il se bloque sur `bikeAdded[bikeType]` jusqu'à ce qu'un vélo de ce type soit déposé. La notification est faite par `putBike()` ou `addBikes()` .

Ordre FIFO : Les variables de condition PcoCOnditionVariable garantissent que les threads bloqués sont réveillés dans leur ordre d'arrivée sur chaque variable, respectant ainsi l'exigence de l'énoncé.

Éviter les interblocages :

- Chaque thread n'acquiert qu'un seul mutex à la fois (celui de la station courante)
- Les habitants se déplacent séquentiellement : prendre vélo → déposer vélo → marcher
- La camionnette ne bloque jamais (getBikes/addBikes non-bloquants)
- Pas de cycles de dépendances possibles

Algorithme de la camionnette

La camionnette suit l'algorithme suivant pour équilibrer les vélos entre les sites :

Étape 1 : Chargement initial au dépôt

- Charge `a = min(2, D)` vélos, où `D` est le nombre de vélos disponibles au dépôt
- Cela garantit un chargement minimal sans vider le dépôt

Étape 2 : Visite de chaque site (i = 0 à 7)

Pour chaque site `i` , soit `Vi` le nombre de vélos présents et `B = 6` le nombre de bornes :

- **Cas 2a : Site surchargé (`Vi > B-2` , donc > 4 vélos)**
- Prendre `c = min(Vi - (B-2), 4 - a)` vélos du site
- Limite : ne prend que si la camionnette a de la place (capacité max = 4)
- Met à jour : `a ← a + c`
- **Cas 2b : Site sous-chargé (`Vi < B-2` , donc < 4 vélos)**
- Calculer `c = min((B-2) - Vi, a)` vélos à déposer

- **Priorité aux types manquants** : pour chaque type de vélo `t`
- Si le site n'a aucun vélo de type `t` ET la camionnette en a au moins un
- Déposer un vélo de type `t` au site
- Incrémenter le compteur de dépôts
- **Compléter avec n'importe quel type** : tant que moins de `c` vélos ont été déposés et que la camionnette n'est pas vide
- Déposer un vélo quelconque pour atteindre le seuil

Étape 3 : Retour au dépôt

- Vider complètement la camionnette : `D ← D + a, a ← 0`
- Tous les vélos restants sont déposés au dépôt

Étape 4 : Pause au dépôt

- Temps d'attente au dépôt : 1 000 000 microsecondes (1 seconde) défini par `VAN*DEPOT*WAITIME`

Objectif de l'algorithme : Maintenir chaque site autour du seuil de 4 vélos (B-2) tout en assurant une diversité de types, grâce à une capacité de camionnette limitée (4 vélos) et un chargement minimal initial (2 vélos).

Opérations par lot pour la camionnette

Pour éviter que la camionnette ne se bloque indéfiniment, les méthodes **getBikes()** et **addBikes()** sont non-bloquantes :

getBikes(n) :

- Retire jusqu'à `n` vélos disponibles immédiatement
- Parcourt les types dans l'ordre (0, 1, 2) en appliquant FIFO par type
- Retourne ce qui est disponible, même si moins de `n` vélos
- Notifie `bikeRemoved[type]` pour chaque vélo retiré

addBikes(vector) :

- Tente d'ajouter chaque vélo du vecteur
- Si la station est pleine, le vélo est retourné dans le vecteur résultat
- Notifie `bikeAdded[type]` pour chaque vélo ajouté avec succès
- Ne bloque jamais, respectant le comportement attendu de la camionnette

Cette approche permet à la camionnette de continuer sa tournée même si un site refuse temporairement des vélos (plein) ou n'a pas assez de vélos à retirer.

Terminaison propre du système

Conformément au cahier des charges, la méthode **ending()** permet d'arrêter proprement la simulation :

Mécanisme :

- Positionne le flag `shouldEnd = true` sous protection du mutex
- Notifie (`notifyAll()`) toutes les variables de condition pour tous les types
- Les threads bloqués dans `getBike()` ou `putBike()` se réveillent
- Détectent le flag `shouldEnd` et sortent proprement sans bloquer

Dans Person::run() : La boucle vérifie `PcoThread::thisThread()->stopRequested()` et `getBike()` retourne `nullptr` si la simulation est arrêtée.

Dans Van::run() : Même vérification du flag d'arrêt.

Résultat : Aucun thread ne reste bloqué indéfiniment dans les variables de condition lors de l'arrêt. La simulation se termine proprement en quelques secondes, même si des habitants ou la camionnette étaient en attente.

Comportement des habitants (Person)

Chaque habitant exécute la boucle spécifiée dans l'énoncé :

- **Attente et prise d'un vélo** : Appel à `getBike(preferredType)` qui bloque si aucun vélo du type préféré n'est disponible
- **Déplacement en vélo** : `bikeTo(destination)` avec temps aléatoire, vers un site j ≠ i choisi aléatoirement
- **Dépôt du vélo** : Appel à `putBike(bike)` qui bloque si le site de destination est saturé (6 vélos)

- **Déplacement à pied** : `walkTo(destination)` vers un autre site k avec temps aléatoire plus long
- **Mise à jour** : Le site courant devient k, la boucle recommence

Préférence de type : Définie aléatoirement dans le constructeur (VTT, Route ou Gravel), chaque habitant n'utilise que son type préféré.

Interface graphique : Chaque action met à jour l'affichage via `binkingInterface->setBikes()` et envoie des logs à la console.

Tests effectués

La simulation, étant une application Qt avec interface graphique, a été testée à la main en observant le comportement des habitants et de la camionnette. Cette methode n'est pas très rigoureuse mais permet de vérifier visuellement que :

- Les habitants prennent et déposent des vélos correctement
- La camionnette rééquilibre les sites comme prévu
- Aucun thread ne reste bloqué indéfiniment
- La terminaison propre fonctionne sans deadlocks

Utilisation de l'IA

Nous avons utilisé l'IA pour nous aider à structurer notre rapport et à clarifier certains concepts liés aux moniteurs de Mesa et à la gestion de la concurrence. L'IA nous a aidés à formuler des explications plus claires et à organiser nos idées de manière cohérente. Cependant, tout le code source et les choix d'implémentation sont entièrement notre propre travail.