MPI: Système, Mutex & Sémaphore

Cours

- Fil d'exécution, non déterminisme, opération atomique.
- Mutex, Sémaphore.
- L'algorithme de Peterson
- L'algorithme de la boulangerie de Lamport
- Problème du rendez-vous, producteur-consommateur, du diner des philosophes.

Lecteurs-Rédacteurs

On se place dans le cadre d'une base de donné avec 2 types d'utilisateurs: les lecteurs et les rédacteurs. La base de donné a ces deux contraintes:

- Plusieurs lecteurs doivent pouvoir lire la base de données en même temps ;
- Si un rédacteur accède à la base de donné alors personne d'autres ne peut l'utiliser.

On cherchera dans toutes les questions à donner le code des fonctions:

- `void demander_lecture()` et `void demander_redaction()` quand un fil demande accès à la base
- `void terminer_lecture()` et `void terminer_redaction()` quand un fil à terminer d'utiliser la base

Question 1 Proposer une solution avec une variable `nb_lect` protégé par un sémaphore initialisé à 1 et un deuxième sémaphore de sorte à ce sorte que le rédacteur soit mis en attente tant qu'il y a encore des lecteurs. Aurait-on pu utiliser 2 mutex ?

Question 2 Montrer qu'il y une pottentielle famine des rédacteurs avec cette solution.

Question 3 Proposer une solution avec un compteur protégé par un mutex, et deux sémaphores qui résoud ce problème de famine.

Votes en parralèlles

On se proposer d'implémenter un système de vote entre plusieurs fils. Chacun des `k` fils d'exécution executera la fonction `void vote(int valeur)` que l'on se propose d'implémenter. Une fois que tous les fils ont voté, **tous les fils** devront executer la fonction `resultat(int result)` avec `result` la valeur la plus voté par les fils (en cas d'égalité, prendre la valeur la plus petite dans tous les gagnant).

Par exemple, pour $k = 5$, si les 5 fils d'exécution executent `vote(2), vote(4), vote(3), vote(4), vote(3)` en parralèlle, alors on doit s'attendre à ce que `resultat(3)` soit appelé par les 5 fils.

Question 1 Dans le cas ou les votes sont soit 0 soit 1, proposer une implémentation de `void vote(int valeur);` à base d'un mutex, un sémaphore initialisé à 0 et de deux variables entières partagés.

Question 2 On considère le cas ou les votes sont dans $[\![0; p]\!]$. Proposer une solution avec un tableau de mutex de longueur p, un compteur protégé par un mutex et un sémaphore initialisé à 0.

Question 3 Proposer une solution avec un tableau de longueur k, le nombre de fils, n'utilisant qu'un compteur protégé par un mutex et un sémaphore.

Question 4 En s'inspirant de l'algorithme de la boulangerie de Lamport, proposer un algorithme utilisant un tableaux d'entier partagé, mais aucun mutex nrei sémaphore. On suppose le tableau initialisé à 0.

Salle de spectacle

On considère une salle de spectacle qui peut être utilisée par deux groupes de musiciens: les violonistes et les guitaristes. Chaque musicien se prépare au spectacle en un temps aléatoire. Dès que tous les musiciens d'un groupe sont prêts, alors ils devront tous jouer ensemble (tous les violonistes ou bien tous les guitaristes jouent). Une fois leur performance faite, ils retournent en préparation.

On suppose qu'il y a N violonistes et M guitariste.

1. Proposer une modélisation de ce problème où chaque musicien est représenté par un fil d'exécution exécutant soit la fonction `violon` soit la fonction `guitare`, sans ce soucier des problèmes de synchronisation. On représentera le fait que tous les musiciens d'un groupe partent jouer par un **unique** appel à une fonction `void spectacle(bool est_violon);`
   On écrit les fonctions `violon` et `guitare`.
2. Quels peuvent être les problèmes rencontrés ?
3. Proposer une solution utilisant 2 sémaphores et 3 mutexs.
4. Et s'il y avait k groupes de musiciens ?

Sémaphore pondéré

On cherche à implémenter une structure de données qui est une généralisation pondérée des sémaphores. Elle aura trois opérations:

- `init(int capacite)` qui initiallisé et crée notre sémaphore pondéré avec `capacite` ressources
- `prendre(int k)` qui bloque le fil jusqu'à ce que l'on puisse alloué un total de k ressources
- `vendre(int k)` qui libère k ressources

Question 1 On propose la suivante première implémentation de notre sémaphore pondéré à l'aide d'un sémaphore normal `sem` :

```
void init(int k) {init(sem,k);}

void prendre(int k) {
  for (int i = 0;i<k;i++) {P(sem);};
}

void vendre(int k) {
  for (int i = 0;i<k;i++) {V(sem);};
}
```

Expliquer pourquoi cette implémentation peut donner à des situations d'interblocages

Question 2 Proposer une implémentation sans interblocage utilisant un compteur protégé par un mutex. Cette solution peut-elle être source de famine ?

Question 3 Proposer une solution garentissant l'abscence de famine utilisant un sémaphore, et une variable protégé par un mutex.

Question 4 Montrer l'abscence d'interblocage et de famine.

Mutex ré-entrant

On cherche à implémenter un mutex réentrant: c'est un mutex sauf que si la ressource est allouée à un fil k, alors le fils k peut se lock autant de fois qu'il veux. Notamment, cela permet d'executer ce code sans situation de blocage:

```
void exemple() {
  lock(mutex_reentrant);
  lock(mutex_reentrant);
  unlock(mutex_reentrant);
}
```

Question 1 Proposer une implémentation d'un tel mutex réentrant en utilisant seulement un mutex classique, un entier partagé et de l'attente active.

On cherche maintenant à faire la même chose pour des sémaphore: si un fil d'exécution execute `p(semaphore)` et obtiens la ressource, alors il peut appeller `p(semaphore)` autant de fois qu'il veux sans être mis en attente ni faire décroire le compteur interne, et ce jusqu'à ce que le fil appelle `v(semaphore)`.

Question 2 Proposer une fonction qui, si elle est executé par deux fils, peut donner lieu a un interblocage des deux fils avec un sémaphore classique, mais qui ne donne pas lieu à un interblocage avec un sémaphore réentrant.

Question 3 Proposer une implémentation d'un tel sémaphore réentrant avec un tableaux de booléens et un sémaphore classique.

Pont à voie unique

On cherche ici a modéliser un pont à voie unique qui ne peut être que emprunter par des voitures allant dans le même sens. On représentera cela par deux types de fils d'exécution: les fils ascendant et les fils descendant. Chaque fil répète en boucle l'exécution de `request_bridge(b);` puis de `exit_bridge(b);` avec `b` un booléen valant `true` iff le fil est ascendant. On veux s'assurer que pour chaque fil, lors de sa présence dans la section critique (entre `request_bridge(b);` et `(b);`), il n'y a que des fils allant dans la même direction.

Question 1 Proposer une solution utilisant un mutex, un compteur partagé et de l'attente active. Expliquer pourquoi il peut avoir une famine.

Question 2 Donner une solution ayant une abscence de famine avec 2 sémaphores initialisé à 1 et un entier. On pourra s'inspirer de la solution avec de l'attente active.

Question 3 On suppose maintenant qu'un maximum de 5 voitures peuvent être présentent en même temps sur le pont. Modifier le code pour correspondre à ce critère.

Question 4 Et s'il le pont possèdait k voies, tel qu'une seule ne peut être empruntée en même temps ?