Algorithmique parallèle et distribuée :
Synchronisation des processus légers concurrents

Julien Rossit

julien.rossit@parisdescartes.fr

IUT Paris Descartes

UNIVERSITÉ PARIS DESCARTES
INSTITUT UNIVERSITAIRE DE TECHNOLOGIE

Un problème classique de traitements concurrents

Un problème de synchronisation classique :

1. des récepteurs reçoivent des demandes, déposent les tâches dans une file ;
2. des ouvriers récupèrent les tâches déposées, les effectuent, et déposent les résultats dans une autre file ;
3. des émetteurs récupèrent les résultats et les envoient aux clients.

Plusieurs problèmes apparaissent :

- plusieurs récepteurs déposent simultanément un travail dans la file ;
- la file des travaux est pleine ;
- plusieurs ouvriers prennent une tâche simultanément ;
- la file des tâches est vide ;
- plusieurs ouvriers déposent simultanément le résultat d'une tâche ;
- la file des résultats est pleine.

Un problème classique de traitements concurrents

Un problème de synchronisation classique :

1. des récepteurs reçoivent des demandes, déposent les tâches dans une file ;
2. des ouvriers récupèrent les tâches déposées, les effectuent, et déposent les résultats dans une autre file ;
3. des émetteurs récupèrent les résultats et les envoient aux clients.

Plusieurs problèmes apparaissent :

- plusieurs récepteurs déposent simultanément un travail dans la file ;
- la file des travaux est pleine ;
- plusieurs ouvriers prennent une tâche simultanément ;
- la file des tâches est vide ;
- plusieurs ouvriers déposent simultanément le résultat d'une tâche ;
- la file des résultats est pleine.

Pour éviter ces problèmes

Il faut considérer :

- **la sûreté** : rien de mauvais ne se produit ;
- **la vivacité** : quelque chose de bon finit par se produire.

Attention au non-déterminisme :

Il faut prendre en compte tous les cas possibles !

L'exclusion mutuelle

Un peu de vocabulaire

A propos des **verrous** :

- plusieurs threads tentent d'accéder à la même ressource ou section, dite *critique* ;
- le problème peut alors être levé par *exclusion mutuelle* ;
- on peut pour cela utiliser des *verrous* :

Principe

Un thread :

1. demande le verrou ;
2. (est bloqué en attente du verrou) ;
3. obtient le verrou, exécute le code critique ;
4. libére le verrou.

- il est théoriquement possible de mettre en place un nombre infini de verrous ;
- une section "verrouillée" est vue comme une opération *atomique* (indivisible).

Exclusion mutuelle

D'un point de vue technique :

- une *section critique* est la suite d'instructions qui utilisent une même ressource ;
- les threads exécutant la même section critique doivent être en

exclusion mutuelle.

Une première solution, l'utilisation du mot clé *synchronized* :

Code

public **synchronized** void *nom* (*paramètres*) { *code* }

Ce qu'il se passe :

1. Si un thread exécute le code de la méthode, les autres restent bloqués à l'entrée ;
2. dès que ce thread termine la méthode, le premier thread resté bloqué est libéré et exécute la méthode, les autres threads bloqués restent bloqués.

Et pour plus de flexibilité ?

Mais si la section critique n'est qu'une infime partie de la méthode ?

Pour restreindre la section critique à une partie du code :

code

```
public void nom(paramètres) {
   ...
   synchronized(objetVerrou) { ... } // section critique
   ...
}
```

l'objet *objetVerrou* :

- est l'objet de référence pour assurer l'exclusion mutuelle (en général *this*) ;
- est un objet partagé entre les threads concernés pour assurer l'exclusion mutuelle ;

le contrôle de concurrence s'effectue au niveau de l'objet.

Contexte d'utilisation

Quelques remarques :

- le mot clef *synchronized* en entête se réfère implicitement à l'objet ***this*** ;
- une méthode *synchronized* est exécutable en concurrence sur des objets distincts ;
- plusieurs méthodes synchronisées sur un même objet ne sont pas exécutables en concurrence (en particulier les méthodes *synchronized* d'un même objet) ;
- les autres méthodes non synchronisées sont exécutables en parallèle ;
- JVM garantit l'atomicité d'accès au byte, char, short, int, float, et référence ;

l'appel d'une méthode synchronized est bien plus long qu'un appel standard !

Les sémaphores

Un sémaphore encapsule un entier positif :

Deux opérations **atomiques** :

- *acquire()* : opération P(), décrémente le compteur si possible, bloque en attente si le compteur est à zéro.
- *release()* : opération V (), incrémente le compteur.

Un sémaphore est vu comme une boite de jetons, avec deux opérations :

- prendre un jeton (attendre qu'il y en ait un) ;
- déposer un jeton.

Un sémaphore permet de resitreindre l'accès à une section critique, où à l'inverse d'implémenter des *barrières de synchronisation*.

Cas particulier

Un sémaphore possédant un unique jeton se comporte comme un verrou.

Un exemple :

```
Semaphore sem =  new  Semaphore(1) ;

try{
  sem.acquire() ;
       //section critique
  sem.release() ;
}catch( InterruptedException e ){
     e.printStackTrace() ;
}
```

La synchronisation coopérative

Un début de synchronisation coopérative

La méthode *join()* permet à un thread d'attendre la fin de l'exécution d'un autre :

Code

```
Thread t.start() ;
...
t.join() ;
```

Cette méthode force le thread courant à attendre la fin de l'exécution du thread t.

Méthodes de la classe *Object*

La *synchronisation coopérative* résoud certains problèmes de concurrence.

Ex : problème des *producteurs/consomateurs*, problème des *lecteurs/écrivains*.

Un objet partagé entre les threads permet cette synchronisation :

Les méthodes utilisés :

- **wait()** : le processus appelant s'endort (sur l'objet) ;
- **notify()** : reveille le premier thread à s'être endormi sur l'objet ;
- **notifyAll()** : reveille tous les thread endormis sur l'objet.

Attention :

L'objet sur lequel on invoque ces méthodes est une ressourse critique !

Le Spin Lock

Considérons le code suivant :

```
if (condition)
{
    o.wait()
}
  ... code ...
```

Un *spurious wakeup* survient lorsqu'un thread sort d'un état de sommeil sans y avoir été invité explicitement.

Pour se protéger, on implémente un mecanisme de *spin lock* :

Code

```
while (condition)
{
    o.wait()
}
  ... code ...
```

La fin des problèmes ?

Inter-blocage

Un inter-blocage (ou **deadlock**) survient lorsque plusieurs threads s'attendent mutuellement :

```
class DeadLock {

  private final Object lock1 = new Object() ;
  private final Object lock2 = new Object() ;

  void methodA() throws InterruptedException{
      lock1.wait() ;
      lock2.notify() ;
  }

  void methodB() throws InterruptedException{
      lock2.wait() ;
      lock1.notify() ;
  }
}
```

... Mais aussi

Encore d'autres problèmes (moins fréquents) :

- **Starvation** (famine) : Un thread n'est pas capable d'obtenir un accès suffisamment régulier pour progresser dans sa tache.
- **Livelock** : survient lorsque les threads sont trop occupés à se synchroniser ; ils ne sont pas bloqués, mais ne progressent plus suffisamment dans leurs tâches.