Mini-projet – Sémaphores Java

Cours Applications Concurrentes, *Polytech-INFO4*

F. Boyer, Université Grenoble Alpes, année 2022-2023

1. Notion de Sémaphore

Comme vu en cours, un sémaphore est un outil de synchronisation qui se compose d'un compteur et d'une file d'attente, et qui fournit deux méthodes essentielles : *P()* et *V()*.

Un sémaphore est souvent utilisé pour gérer l'accès à un ensemble de ressources indifférenciées. La méthode *P()* permet demander une ressource et la méthode *V()* permet de libérer une ressource.

Le compteur du sémaphore représente à tout instant le nombre de ressources disponibles. Lorsqu'un thread demande une ressource et qu'il n'y a plus de ressources disponible, le thread se retrouve bloqué dans la méthode *P()*. Il ne sera débloqué que lorsqu'une ressource sera disponible. La file d'attente du sémaphore est utilisée pour conserver les références des threads qui sont bloqués en attente de ressource.

Nous rappelons ci-après le *schéma de code simplifié* correspondant aux méthodes *P()* et *V()* (ce schéma ne s'applique que pour les sémaphores dont le compteur est initialisé à une valeur positive ou nulle). Par convention, *S*.c représente le compteur du sémaphore *S* et *S.f* représente la file d'attente du sémaphore *S*.

```
P (Semaphore S) :
        S.c--;
        if (S.c < 0) {              // no more resources, suspend current thread
            add(Thread,currentThread(), S.f);
              thread-switch();
             }
}
```

```
V (Semaphore S):
        S.c++;
        if (S.c <= 0) {             // there is at least 1 waiting thread
            Thread t = get(S.f);    // get the oldest waiting thread
            wakeup(t);              // put t in readyqueue
            }
}
```

En général, les sémaphores garantissent une ordonnancement **FIFO** sur l'allocation des ressources, c'est à dire que si un thread *Ta* invoque *S.P()* avant qu'un thread *Tb* invoque *S.P(),* alors *Ta* obtiendra une ressource de *S* avant *Tb*. Pour obtenir cette propriété, l'instruction *get(S.f)* qui est dans la méthode *V(Semaphore S)* doit retourner le plus ancien thread bloqué dans *S.f*.

Question. Dérouler un scenario d'exécution avec un sémaphore *S* initialisé à 1, et 3 threads T1, T2, T3 qui exécutent chacun le code suivant : *S.P() ; System.out.println(..) ; S.V() ;*

2. Sémaphores Java

En Java, la classe *Semaphore* est fournie par la librairie *java.util.concurrent*. Cette classe fournit les méthodes *P()* et *V(),* mais celles-ci s'appellent respectivement *acquire()* et *release()*.

Exercice pratique. Regardez la javadoc associée à la classe *Semaphore*, vous constaterez que de très nombreuses méthodes sont fournies. Notez que la classe *Semaphore* de Java fournit un constructeur *Semaphore (int permits, boolean fair)* qui prend en argument un booléen permettant de préciser si l'on souhaite que le sémaphore garantisse (ou non) un ordonnancement FIFO sur l'acquisition des ressources.

Jouez un peu avec les sémaphores en écrivant des petits programmes multi-threadés, et en observant leur exécution (notamment sous debug). Si vous n'avez pas d'idées, vous pouvez simplement implémenter la classe ***Parking*** vue en cours.

3. Usage élémentaire de sémaphore

Les sémaphores sont souvent utilisées pour gérer des ressources partagées ou pour coordonner des threads dans leur exécution. Ici nous allons considérer un cas de coordination élémentaire.

Exercice. En supposant que l'on ne dispose pas de la méthode *join()* en Java, proposez un moyen à base de sémaphores permettant à un thread d'attendre la terminaison des threads fils qu'il a créé. On supposera que les threads fils peuvent manipuler des objets (notamment des sémaphores) partagés avec le thread père. Préciser les instructions à placer dans le code du thread père et celles à placer dans le code des threads fils.

4. Coordination de threads Bavards

Nous allons considérer l'usage de sémaphores pour mettre en œuvre un autre cas (basique) de coordination de threads, basé sur le programme Bavard.

.

Exercice pratique. Reprenez le programme Bavard. En supposant qu'il y a autant de threads Bavard A créés que de threads BavardB, faites en sorte de respecter la règle ci-après (vous n'utiliserez que des sémaphores, pas de join) :

(a) Un thread Bavard B ne pourra démarrer qu'après qu'un thread Bavard A ait terminé. Deux threads Bavard B ne pourront donc être en exécution qu'après que deux threads Bavard A aient terminés.
(b) Les threads Bavard B ne pourront démarrer leur exécution qu'après la terminaison d'au moins un thread Bavard A.

5. Classe RDV à base de sémaphore

Nous allons considérer un cas de coordination un peu plus élaboré que précédemment : un rendez-vous entre threads.

Exercice pratique. Programmer une classe *Rdv* basée sur l'usage de sémaphore.
Vous reprendrez la même spécification que celle du TD/TP Moniteurs : le constructeur permet de préciser le nombre de threads attendus au rdv. La méthode *come()* permet à un thread d'arriver au rdv. Tout thread arrivant au rendez-vous est mis en attente tant que les autres threads attendus ne sont pas tous arrivés.

Usage (exemple) :

```
…
Rdv rdv = new Rdv(3) ;
MyThread mt1 = new MyThread(rdv).start() ;
MyThread mt2 = new MyThread(rdv).start() ;
MyThread mt3 = new MyThread(rdv).start() ;
…

class MyThread extends Thread {
    Rdv rdv ;
    public MyThread(Rdv rdv) { this.rdv = rdv ;}
    public void run() { … ; rdv.come() ; …}
   }
```

Programmer cette classe sur papier, puis implémentez la sous Eclipse en la testant avec différents cas d'usage. Comme indication, sachez que vous aurez besoin de 2 sémaphores pour programmer cette classe, l'un pour assurer l'exclusion mutuelle au niveau de la manipulation des variables partagées, et l'autre pour bloquer les threads tant que les autres ne sont pas arrivés au rendez-vous.

6. Comparaison en termes de performances

Les performances d'exécution ne sont généralement pas le critère de choix décisif entre l'usage de sémaphores ou de moniteur dans la mise en œuvre d'une section de code concurrent. Les critères de choix s'orientent plutôt vers la simplicité d'écriture et de compréhension du code. Le code concurrent est souvent intrinsèquement complexe à comprendre, il faut privilégier la compréhensibilité pour en assurer l'évolutivité.

Simplement à titre d'exercice, nous vous demandons de comparer les coûts en termes de performances entre un appel de méthode synchronisée et un appel de méthode encadrée par *mutex.P()* et *mutex.V()*.