Programmation concurrente

Sémaphores et Sections critiques conditionnelles

Polytech/INFO 4, 2020-2021

Fabienne Boyer
UFR IM2AG, LIG, Université Grenoble Alpes
Fabienne.Boyer@imag.fr

Sémaphores (Dijkstra, 1965)

Un outil permettant de contrôler l'exécution de threads concurrents, et notamment de gérer des attentes passives conditionnelles

Sémaphore *S* :
- **compteur S.c** → Modélise un nombre de ressources ou une condition de passage
- **file d'attente S.f** → Processus en attente

Fournit 2 opérations atomiques :
- **P** → Prendre une ressource
- **V** → Libérer une ressource

Sémaphores (Dijkstra, 1965)

Solution avec compteur initial positif ou nul

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

```
V (Semaphore S):
        S.c++;
        if (S.c <= 0) do {
          // at least 1 waiting process
            Thread t = get(S.f);
            wakeup(t); // put p in ReadyQueue
}
```

sections critiques

Exemple

```
class Parking{
Semaphore park;

   Parking (int nplaces) {
     park= new Semaphore(nplaces);
   }

        void enter(){
              park.P();
        }

          void leave(){
              park.V();
        }
}
```

Propriété des sémaphores: en général FIFO (absence de vol de cycle)

P (Semaphore S):

```
S.c--;
if (S.c < 0) do { // no more resources
    add(Thread,currentThread(), S.f);
   thread-switch();
```

V (Semaphore S):

```
 S.c++;
 if (S.c <= 0) do {
    // at least 1 waiting process
     Thread t = get(S.f);
     wakeup(t); // put p in readyqueue
}
```

Exemple avec un parking a 1 place

T1: enter ->
cpt = 0
entre dans parking

T2: enter ->
cpt = -1
bloqué dans S.f

T1: leave ->
cpt = 0
T2 remis dans ReadyQueue

T3: enter ->
cpt = -1
bloqué dans S.f

T2: entre ds parking

➔ **T2 ne peut pas se faire doubler par un nouvel arrivant**

Propriétés des sémaphores

- **Compteur S.c == S.c initial + nV − nP**
  - nV est le nombre d'opérations V exécutées sur le sémaphore
  - nP est le nombre d'opérations P exécutées sur le même sémaphore
- **Compteur S.c < 0 : correspond au nombre de threads bloqués**
- **Compteur S.c > 0 : correspond au nombre de ressources disponibles**
- **Compteur S.c == 0 : aucune ressource disponible et aucun thread bloqué**

Semaphores Java

- **Package java.util.concurrent**
  - Class Semaphore
    - void acquire()
    - void release()
    - …
  - Propriété FIFO indiquée au niveau du constructeur
    - Semaphore(int permits, boolean fifo)

Sémaphores vs Moniteurs

- **Le choix dépend..**
  - Plus de sémantique dans les sémaphores, mais moins de liberté de programmation
  - Pas de vol de cycle -> solutions sans famine (FIFO) plus simple avec sémaphores
  - Par contre, la gestion de priorités est souvent plus simple avec des moniteurs
- **Par exemple**
  - Un producteur-consommateur se programme plus simplement avec des sémaphores
  - Un RDV se programme plus simplement avec un moniteur
  - Un allocateur de ressources FIFO se programme plus simplement avec un sémaphore
  - Un allocateur de ressources avec priorités se programme plus simplement avec un moniteur
- **Dans certains cas on utilise les deux conjointement..**

Sémaphores vs Moniteurs

Plus simple

```
class Parking{
 Semaphore park;

  Parking (int nplaces) {
    park= new Semaphore(nplaces);
  }

      void enter(){
            park.P();
      }

       void leave(){
            park.V();
      }
}
```

```
class Parking{
int nfree;
   Parking (int nplaces) {
      nfree = nplaces;
   }

synchronized void enter(){
 if (nfree == 0)  wait();
      nfree--;
}

synchronized void leave(){
   nfree++;
   notifyAll();
}
```

Sémaphores vs Moniteurs

```
class Parking{
Parking (int nplaces) {
   park= new Semaphore(nplaces);
   att = new Semaphore(0);
}

void enter(){
   while (nprio > 0) att.P();
   park.P();
}
void enterPrio(){
    nprio++;
    park.P();
    nprio--;
    att.V();
}

void leave(){
   park.V();
 }
```

Plus simple

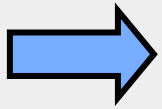
```
class Parking{
int nfree, nprio;
Parking (int nplaces) {
      nfree = nplaces;
   }

synchronized void enter(){
 if (nfree == 0 || nprio>0)  wait();
      nfree--;
}

synchronized void enterPrio(){
    nprio++;
    if (nfree == 0)  wait();
    nprio--; nfree--;
}

synchronized void leave(){
   nfree++;
   notifyAll();
}
```

Les sections critiques conditionnelles

- **Outil très proche des moniteurs, mais avec une gestion explicite du verrou**
  - lock(verrou)
  - wait(condition, verrou)
    - libère le verrou V, bloque le processus courant sur C, puis reprend le verrou V à son réveil
  - signal(condition, verrou)
    - réveille un processus bloqué sur C, s'il y en a un (fugace)
  - unlock(verrou)
- **C'est un moyen pour gérer plusieurs wait-sets dans un objet**

SCC « Moniteur » (Implantation procédurale d'un moniteur avec une SCC)

```
monitor M {
    condition c;
    procedure P (…) {
            …
            wait(c)
            . . .
           notify(c)
    }
}
```



```
SCC M {
    condition c;
    lock l;
    void P (…) {
            lock(l);
            . . .
             wait(c, l);
             . . .
             signal(c,l)
             . . .
             unlock(l);
    }
}
```

SCC Java

- **Package java.util.concurrent**
  - Interface Lock → classes ReentrantLock, ReentrantReadWriteLock
    - lock / tryLock(timeout)
    - unlock
    - newCondition
  - Interface Condition
    - await: le thread courant est bloqué jusqu'à ce qu'il soit signalé ou interrompu
    - signal / signalAll: réveille un thread bloqué

Exemple

```
class Parking{
Lock lock = new ReentrantLock();
freePlaces = lock.newCondition();

  Parking (int nplaces) {
  }

 void enter(){
            lock.lock();
            while (nfree <= 0)  // vol de cycle possible
              freePlaces.wait();
            lock.unlock();
 }

 void leave(){
            lock.lock();
            nfree++;
            freePlaces.signal();
            lock.unlock();
}
}
```

A titre d'exercice:
Implantation des SCC
priorité signalant – avec vol de cycles

```
class Scc {
Lock mutex;
Lock condition;
int nwaiting;

public Scc {
  mutex = new Lock(); // Scc exclusion
  condition  = new Lock();
  condition.lock();
  nwaiting = 0;
}

public void lock () {
        mutex.lock();
}
public void unLock () {
        mutex.unLock();
}
```

```
public void wait() {
 nwaiting++;
 mutex.unLock(); // free the SCC
 condition.lock(); // block the current thread
 mutex.lock();      // re-enter in the SCC
}

public void signal() {
 if (nwaiting > 0) {
  // wake up a waiting process
   nwaiting--;
   condition.unLock();
  }
}

public void signalAll() {
while (nwaiting > 0) {
  nwaiting--;
  condition.unLock();
}
```

A titre d'exercice:
Implantation des SCC
priorité signalant – sans vol de cycles

```
class Scc {
Lock mutex, condition, wakeup;
int nwaiting = 0;
int nwakeup = 0;

public Scc {
 mutex      = new Lock();
 condition  = new Lock();
 wakeup     = new Lock();
 condition.lock();
 wakeup.lock();
}
public void lock () {
  mutex.Lock();
}

public void unLock () {
  if (nwakeup > 0){
     nwakeup --; wakeup.unLock();
  } else mutex.unLock();
}
```

```
public void wait() {
        nwaiting++;
        this.unLock();
        condition.lock();
        wakeup.lock();
}

public void signal() {
         if (nwaiting > 0) {
            nwaiting--;
            condition.unLock();
            nwakeup++;
 }

public void signalAll() {
          while (nwaiting)> 0) {
              nwaiting--;
              condition.unLock();
              nwakeup++;
        }
}
```