Programmation concurrente

Les Moniteurs

Polytech/INFO 4, 2022-2023

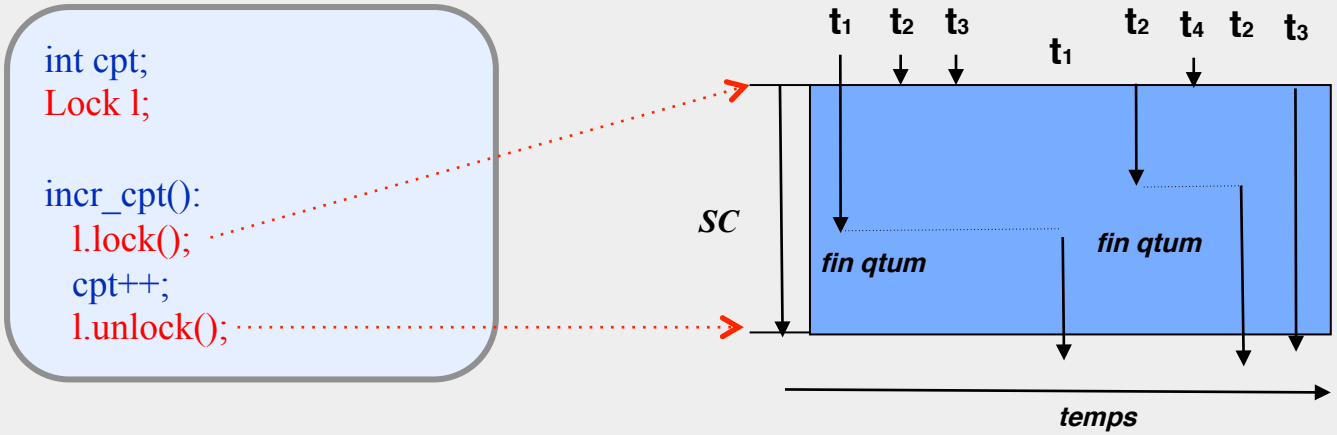
Fabienne Boyer
UFR IM2AG, LIG, Université Grenoble Alpes
Fabienne.Boyer@imag.fr

POLYTECH GRENOBLE

UNIVERSITÉ Grenoble Alpes

Insuffisance des solutions de base pour la synchronisation

- Les verrous permettent de gérer des **sections critiques**



- Ils ne permettent pas de gérer des **sections critiques *conditionnelles***

Exemple du parking

```
int nfree = ..;

enter():
  while (nfree == 0) ;
   nfree--;

leave():
    nfree++;
```

Exemple du parking

```
int nfree = ..;

enter():
  while (nfree == 0) ;
   nfree--;

leave():
    nfree++;
```

Pas de synchro -> nfree peut devenir incohérent

Exemple du parking

```
int nfree = ..;

enter():
  while (nfree == 0) ;
   nfree--;

leave():
    nfree++;
```

Pas de synchro -> nfree peut devenir incohérent

```
int nfree = ..;
Lock l;

 enter():
   l.lock();
   while (nfree == 0) ;
   nfree--;
   l.unlock();

 leave():
    l.lock();
    nfree++;
    l.unlock();
```

Exemple du parking

```
int nfree = ..;

enter():
  while (nfree == 0) ;
   nfree--;

leave():
    nfree++;
```

Pas de synchro -> nfree peut devenir incohérent

```
int nfree = ..;
Lock l;

 enter():
   l.lock();
   while (nfree == 0) ;
   nfree--;
   l.unlock();

 leave():
    l.lock();
    nfree++;
    l.unlock();
```

PB d'attente infinie

Exemple du parking

```
int nfree = ..;

enter():
  while (nfree == 0) ;
   nfree--;

leave():
    nfree++;
```

Pas de synchro -> nfree peut devenir incohérent

```
int nfree = ..;
Lock l;

 enter():
   l.lock();
   while (nfree == 0) ;
   nfree--;
   l.unlock();

 leave():
    l.lock();
    nfree++;
    l.unlock();
```

PB d'attente infinie

```
int nfree = ..;
Lock l;

 enter():
   while (nfree == 0) ;
   l.lock();
   nfree--;
   l.unlock();

 leave():
    l.lock();
    nfree++;
    l.unlock();
```

Exemple du parking

```
int nfree = ..;

enter():
  while (nfree == 0) ;
   nfree--;

leave():
    nfree++;
```

Pas de synchro -> nfree peut devenir incohérent

```
int nfree = ..;
Lock l;

 enter():
   l.lock();
   while (nfree == 0) ;
   nfree--;
   l.unlock();

 leave():
    l.lock();
    nfree++;
    l.unlock();
```

PB d'attente infinie

```
int nfree = ..;
Lock l;

 enter():
   while (nfree == 0) ;
   l.lock();
   nfree--;
   l.unlock();

 leave():
    l.lock();
    nfree++;
    l.unlock();
```

Plusieurs peuvent tester (nfree!=0) alors qu'il n'y a qu'une place libre

Exemple du parking: ce que l'on aimerait exprimer

```
int nfree = ..;
Lock l;

enter():
  l.lock();
  while (nfree == 0)
      <se suspendre en libérant lock>
      <reprendre lock qd on est réveillé>
  nfree--;
  l.unlock();

leave():
    l.lock();
    nfree++;
    <réveiller un suspendu>
    l.unlock();
```

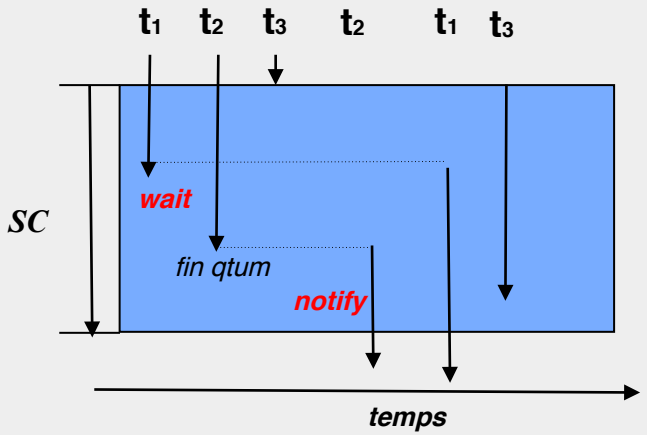
Section critique conditionnelle

(attente passive si nfree vaut 0)

Solutions de plus haut niveau

- **Permettre au programmeur de gérer des sections critiques conditionnelles**
  - Suspendre un processus (avec libération de la SC) lorsqu'il ne peut pas continuer à s'exécuter
  - Le réveiller lorsqu'il peut reprendre son exécution (avec reprise de la SC)

- **Les solutions**
  - Les sémaphores
  - Les moniteurs
  - Les SCC



Les moniteurs

- **Un moniteur est un module comprenant**
  - Des données
  - Des procédures d'accès (P1,..,Pn) exécutées en exclusion mutuelle
  - Une procédure d'initialisation
  - Des conditions

  Une condition est une structure fournissant deux opérations :

  - wait() suspend le processus courant (avec libération/reprise du moniteur)
  - signal() réveille un processus bloqué s'il y en a un. Le signal est fugace.

  En général, les conditions sont gérées de manière FIFO

Schéma de moniteur

```
monitor <monitor-name> {

    <shared variables  + conditions declarations>

    exclusive procedure P1 (…) {
            . . .
        <condition>.wait();
            . . .
    }
    exclusive procedure P2 (…) {
            . . .
        <condition>.notify();
            . . .
    }
    procedure Pn (…) {
            . . .
    }
```

Fonctionnement d'un moniteur

→ **Au maximum un seul thread actif dans un moniteur**

- **Lors d'un notify**
  - Soit le signalant garde le moniteur (priorité signalant)
  - Soit le signalé prend le moniteur (priorité signalé)
- **Libération du moniteur**
  - Lorsque la procédure en cours est terminée
  - Lors d'un wait
- **Lors de la libération du moniteur**
  - Le moniteur peut être alloué en priorité à un thread déjà dans le moniteur (un signalé)
  - Ou bien il n'y a pas de règle (tous les threads sont en concurrence) → **vol de cycle possible**

Moniteurs Java

- **Moniteurs Java**
  - Méthodes (ou blocs) synchronisés = exécutés en exclusion mutuelle
  - Opérations wait et notify/ notifyAll (condition implicite)
- **Objets synchronisés**
  - Instances
  - Classes

```
class Parking{
 int cpt;
 ..
 public void synchronized enter() {
      while (cpt == 0)
            wait();
      cpt--;
}
public void synchronized leave() {
      cpt++;
      notify();
}
```

Principes d'implantation des moniteurs Java

- Tout objet synchronisé possède un verrou
- Ce verrou est transparent pour le programmeur (il est ajouté par le compilateur)
- Le compilateur encadre toute méthode synchronisée par une prise et une libération du verrou

```
class Parking{
 lock l;
 int cpt;
 ..
 public void enter() {
      l.lock();
      while (cpt == 0)
           wait();
      cpt--;
      l.unlock();
}
public void leave() {
      l.lock();
      cpt++;
      notify();
      l.unlock();
}
```

Principes d'implantation des moniteurs Java (schéma de traduction / bloc)

```
class Parking{
int cpt;
..
public void enter() {
      System.out.println(… );
      synchronized(this){
          while (cpt == 0)
               wait();
      cpt--;
   }
}
public void synchronized leave() {
      cpt++;
      notify();
}
```

```
class Parking{
 static lock l;
static int cpt;  // shared data

 public static void enter() {
      System.out.println(…);
      l.lock();
      while (cpt == 0) {
          wait();
      }
      cpt--;
      l.unlock();
}
public static void leave() {
      l.lock();
      cpt++;
      notify();
      l.unlock();
}
```

Principes d'implantation des moniteurs Java

- Tout objet synchronisé possède un verrou et une file de threads bloqués (wait-set)
- **wait()** insert le thread courant dans wait-set, libère le verrou, puis reprend le verrou
- **notify()** retire un thread de wait-set (s'il y en a un) et l'insère dans la ReadyQueue

Principes d'implantation des moniteurs Java (schéma de traduction)

```
class Parking{
 int cpt;
 ..
 public void synchronized enter() {
       while (cpt == 0)
          wait();
       cpt--;
}
public void synchronized leave() {
       cpt++;
       notify();
}
```

```
class Parking{
 lock l;
 List<Thread> wait-set;
 int cpt;  // shared data

 public void enter() {
       l.lock();
       while (cpt == 0) {
          wait-set.add(Thread,currentThread());
          l.unlock(); thread-switch(); l.lock();
       }
       cpt--;
       l.unlock();
}
public void leave() {
       l.lock();
       cpt++;
       if !wait-set.isEmpty() wakeup(wait-set.get(0));
       l.unlock();
}
```

Principes d'implantation des moniteurs Java (schéma de traduction / static)

```
class Parking{
 static int cpt;
 ..
public static void synchronized enter() {
      while (cpt == 0)
         wait();
      cpt--;
}
public static void synchronized leave() {
      cpt++;
      notify();
}
```

```
class Parking{
 static lock sl;
 static List<Thread> swait-set;
 static int cpt;
 ..
 public static void enter() {
      sl.lock();
      while (cpt == 0) {
        swait-set.add(Thread,currentThread());
        sl.unlock(); thread-switch(); sl.lock();
      }
      cpt--;
      sl.unlock();
}
public static void leave() {
      sl.lock();
      cpt++;
      if !swait-set.isEmpty() wakeup(swait-set.get(0));
      sl.unlock();
}
```

Propriétés des moniteurs Java

```
class Parking{
 int cpt;
 ..
 public void synchronized enter() {
      while (cpt == 0)
         wait();
      cpt--;
}
public void synchronized leave() {
      cpt++;
      notify();
}
```

- **Propriétés garanties**
  - Exclusion
  - Réentrance
  - ~~Absence de famine~~
- **Famine possible à cause du vol de cycle**

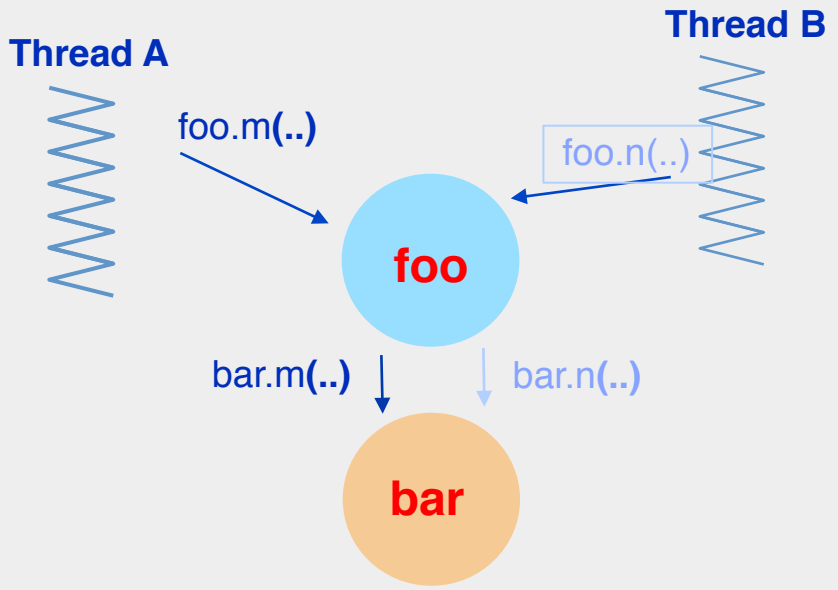
  Exemple avec un parking à 1 place
  T1: enter -> entre dans parking
  T2: enter -> wait
  T1: leave -> T2 remis dans ReadyQueue
  T3: enter -> entre dans parking
  T2: se rebloque dans wait
  ➔ T2 se fait doubler par un nouvel arrivant
- **De plus le notify n'est pas forcément FIFO**

Usage des moniteurs Java : attention à l'imbrication des clauses synchronized

```
Class Foo{
   Bar b;
      ..
   synchronized void m(){
          ..;  b.m(); ..
    }
   synchronized void n(){
          ..; b.N(); ..
    }
 }
```

```
class Bar{

   synchronized void m(){
                          ..; wait();
   }
   synchronized void n(){
                          ..; notify(); ..
   }
   ..
```



blocage infini du à l'imbrication des blocs synchronisés- le thread qui se bloque dans l'objet bar ne relâche pas le verrou pris sur l'objet foo