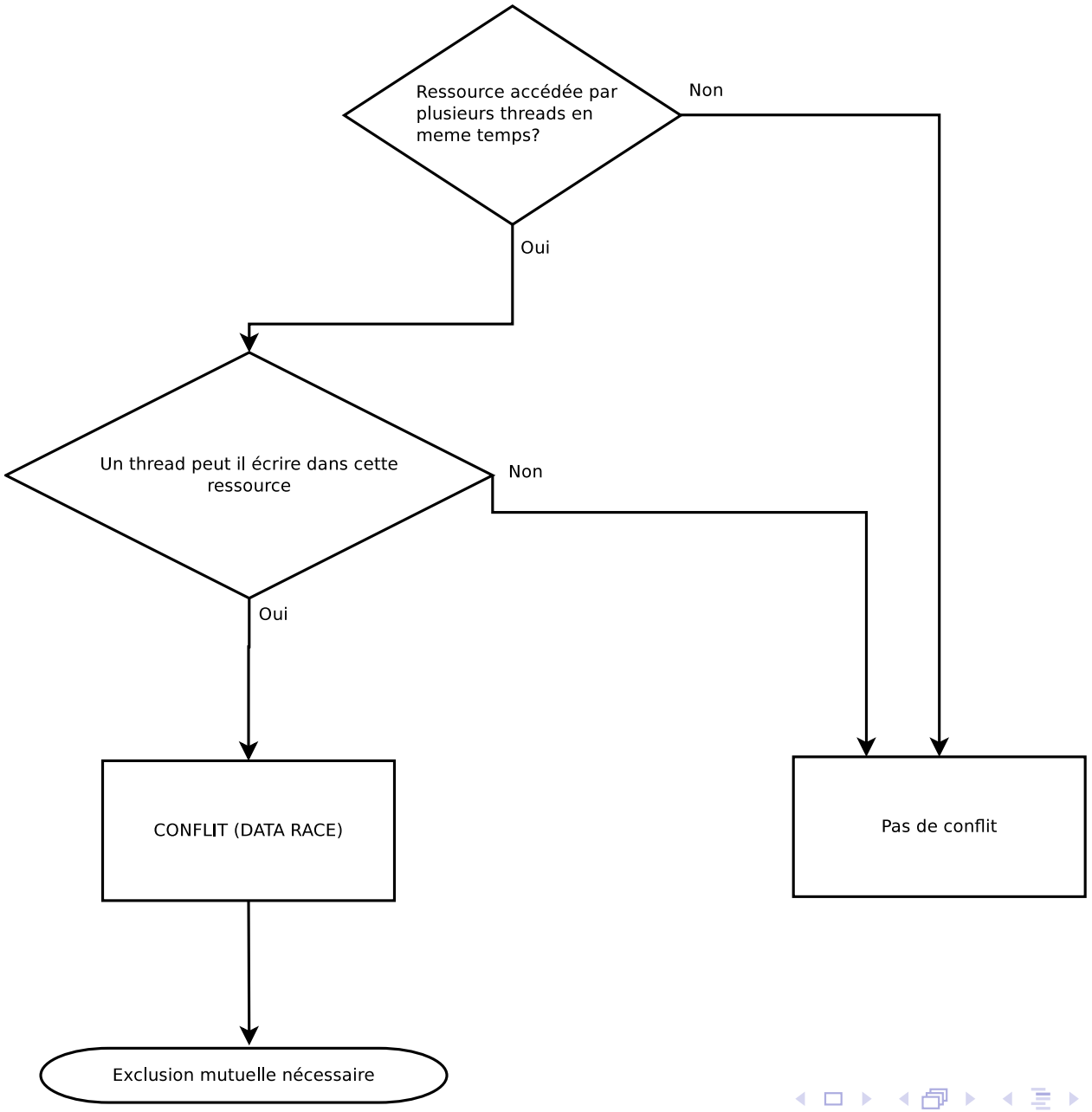
Programmation Répartie 3 : Mécanismes de synchronisation

Thomas Pinsard


Ressource accédée par plusieurs threads en meme temps?
Non
Oui
Un thread peut il écrire dans cette ressource
Non
Oui
CONFLIT (DATA RACE)
Pas de conflit
Exclusion mutuelle nécessaire

Verrou

- Mécanisme de synchronisation plus flexible que les `synchonized`
- Présent dans de nombreux langages (C,C++,C#,python,...)
- Deux opérations atomiques de base : `lock` et `unlock`

```
Lock lock = new ReentrantLock();

lock.lock();

//section critique

lock.unlock();
```

Equivalence par rapport au bloc synchronized

Méthode synchronised

```
Class Compteur{
    int c=0;
 public synchronized
     void inc(){
  c = c+1;
 }
}
```

bloc synchronized

```
class Compteur{
    int c=0;
public void inc(){
   synchronized(this){
   c = c+1;
   }
 }
}
```

verrou

```
class Compteur{
Lock lock =
    new ReentrantLock();
    int c =0;
    void inc(){
        lock.lock();
        c = c+1;
        lock.unlock();
    }
}
```

Contraintes

- Un thread tenant de verrouiller un verrou déja pris est mis en attente sur ce verrou
- Le thread qui verrouille doit être le même que celui qui déverrouille

- Plus flexible, mais plus de responsabilité pour l'utilisateur
- Le déverrouillage n'est plus automatique, car la section atomique n'est plus délimitée syntaxiquement

Pas de déverrouillage ?

Exception

```
Lock lock = new ReentrantLock();

try{

lock.lock();


//code pouvant lance une exception


lock.unlock();

}
catch(Exception e){
}
```

Return

```
Lock lock = new ReentrantLock();

lock.lock();

if (cond1){
    x=x+1;
    return x;
}
else{
    x=x-1;
}

lock.unlock();

return x;
```

Déverrouillage

Exception

```
Lock lock = new ReentrantLock();

try{

lock.lock();


//code pouvant lance une exception



}
catch(Exception e){
}
finally{
lock.unlock();
}
```

Return

```
Lock lock = new ReentrantLock();

lock.lock();

try{
if (cond1){
    x=x+1;
    return x;
}
else{
    x=x-1;
}
}
finally{
lock.unlock();
}

return x;
```

Réentrant

- Que se passe-t-il lorsqu'un thread tente de verrouiller un verrou qu'il possède déja
  - Réentrant :
    - Il le verrouille une nouvelle fois
    - Il devra le déverrouiller autant de fois qu'il l'a verrouillé
  - Non réentrant : cette opération n'est pas autorisée

Différence par rapport au bloc synchronized

- La prise de verrou et la libération peuvent se faire depuis deux blocs d'instructions différents (et donc deux méthodes différentes)

Verrouillage/Déverrouillage dans deux méthodes différentes

```
Lock lock = new ReentrantLock();

void m1(){
    lock.lock();
    // code de m1
}

void m2(){
    //code de m2
    lock.unlock();
}
```

trylock

- `trylock` : acquiert le verrou s'il est libre, sinon retourne faux
- Solution non bloquante
- Permet de différencier les traitements

```
boolean isLock = lock.trylock();
if (isLock){
    //section critique
    lock.unlock();
}
else{
    //code sans exclusion mutuelle necessaire
}
```

Verrous en Java

- Interface `Lock` (package `java.util.concurrent.locks`)
- Depuis Java 1.5
- Implantation de l'interface
  - ReentrantLock
  - ReadWriteLock
  - ...

Sémaphore

Définition

- Maintiens un ensemble de permissions
- Demande de permission avant d'entrer dans une section à protéger
- Si plus de permission, la section ne peut pas être exécutée
- Développé en 1962 par Edsger Dijkstra pour un nouveau système d'exploitation

Opérations de base

- P : demande une permission
- V : rend une permission
- Init : initialise le sémaphore avec un nombre de permissions

Ces opérations doivent être atomiques

- Permet de limiter les accès à une ressource

Sémaphore

```
Semaphore semaphore = Init(10);

useRessource(){
        P(semaphore)
        //do something
]


stopUsingRessource(){
    //do something
    V(semaphore)
}
```

Présent dans la bibliothèque java

- P : `acquire()` ou `acquire(nombre de permissions)`. Bloque si le nombre de permission demandé n'est pas disponible
- V : `release()` ou `release(nombre de permissions)`
- Init : constructeur

Sémaphore

```
Semaphore semaphore = new Semaphore(10,true);

public void useRessource(){
    semaphore.acquire();
    //do something
}

public void stopUsingRessource(){
    //do something
    semaphore.release();
}
```

Différences avec un verrou

- Gestion d'un ensemble d'autorisations
- Un sémaphore avec une autorisation (sémaphore binaire) est-il équivalent à un verrou ?
  - Non, car pas de notion de propriété dans un sémaphore
  - Un thread qui acquiert un verrou devient le propriétaire
  - Un thread peut prendre une autorisation a un sémaphore, et un thread différent la rendre

Moniteur

- Structure de synchronisation
- Constitué de
  - Verrou
  - Ressources
  - Ensemble de procédure permettant d'interagir avec les ressources
  - Mécanisme d'attente

A tout moment un seul thread peut exécuter une méthode d'un moniteur

- Mécanisme voulu plus facile à manipuler que les verrous ou les sémaphores
- Implanté pour la première fois dans Concurrent Pascal
- Mécanisme de base de la synchronisation de Java

Transaction

- Chaque section atomique est éxécutée en supposant qu'il n'y a pas d'interférence
- Si une interférence est détectée, les effets de la section doivent être annulés
- La section est ensuite ré-éxécutée

Transaction

Avantages

- Facile pour l'utilisateur :
  - Délimite uniquement les sections atomiques
  - Pas de déclarations de mutex, ...
- Si peu d'interférence, gain

Désavantages

- Difficile à implanter :
  - Nécessite de maintenir un journal des modifications
  - Les effets de la transactions doivent être annulables (donc pas d'entrée/sortie)
- Mauvaise performance

- Inspirées des bases de données
- Présent sous forme de librairies Java (DSTM2), et dans certains nouveaux langages
- Peu utilisées

Conclusion

- Verrous
- Sémaphores
- Moniteurs
- Transactions