Exclusion mutuelle: problèmes derivés

- Producteurs/consommateurs
- Lecteurs/redacteurs
- Diner des philosophes

Producteur consommateur

- Partage entre threads d’une file
- Les threads productrices produisent, la production est mise a dans la file
- Les threads consommatrices retirent les éléments de la file pour les consommer

Producteurs consommateurs

- Exclusion mutuelle entre threads productrices pour écrire dans la file
- Exclusion mutuelle entre threads consommatrices pour consommer les éléments de la file
- Attente des threads consommatrices quand la file est vide
- Attente des threads productrices quand la file est pleine

Interface Lock

`void lock()`
Acquires the lock.

`void lockInterruptibly()`
Acquires the lock unless the current thread is **interrupted**.

`Condition newCondition()`
Returns a new **`Condition`** instance that is bound to this `Lock` instance.

`boolean tryLock()`
Acquires the lock only if it is free at the time of invocation.

`boolean tryLock(long time, TimeUnit unit)`
Acquires the lock if it is free within the given waiting time and the current thread has not been **interrupted**.

`void unlock()`
Releases the lock.

Interface Condition

`void await()`
Causes the current thread to wait until it is signalled or **interrupted**.

`boolean await(long time, TimeUnit unit)`
Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.

`long awaitNanos(long nanosTimeout)`
Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.

`void awaitUninterruptibly()`
Causes the current thread to wait until it is signalled.

`boolean awaitUntil(Date deadline)`
Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses.

`void signal()`
Wakes up one waiting thread.

`void signalAll()`
Wakes up all waiting threads.

Class ReentrantLock

- Implémente **`Lock`**
- Il peut être acquis plusieurs fois par la même thread
- Le constructeur de cette classe peut avoir un paramètre d'équité. Positionner à **`true`** il indique qu'en cas d'attente sur le verrou c'est la thread qui est la première arrivée qui est servie.
- Méthodes pour connaitre les threads en attente sur le lock ou sur la condition associée

En java

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  =
lock.newCondition();
   final Condition notEmpty =
lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws
InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signalAll();
     } finally {
       lock.unlock();
     }
   }
```

```
 public Object take() throws InterruptedException
{
     lock.lock();
     try {
       while (count == 0)
         notEmpty.await();
       Object x = items[takeptr];
       if (++takeptr == items.length) takeptr = 0;
       --count;
       notFull.signalAll();
       return x;
     } finally {
       lock.unlock();
     }
   }
 }
```

Lecteurs Rédacteurs

- Partage entre threads d’une base de données
- Des threads écrivains écrivent dans la base de données
- Des threads lectrices écrivent dans la base de données

Lecteurs rédacteurs

- Exclusion mutuelle entre threads écrivains et lectrices lors de l’écriture

- Exclusion mutuelle entre threads écrivains

- On veut que plusieurs threads lectrices puissent lire concurremment

Lecteurs-écrivains

- les lecteurs (readers) retournent des valeurs lues (sans les modifier)
- les écrivains (writers) modifient les valeurs
  - les lecteurs n'ont pas besoin de se synchroniser pour lire
  - un écrivain doit faire sa modification en exclusion mutuelle

```
public interface ReadWriteLock {
  Lock readLock();
  Lock writeLock();
}
```

- *2 verrous: readLock() et writeLock()*

Lecteurs-écrivains

- *condition de sûreté:*
  - une thread ne peut obtenir un write-lock tant que d'autres threads possèdent le write-lock ou le read-lock
  - une thread ne peut obtenir le read-lock si une autre thread possède le write-lock

Implémentation

- Class ReentrantReadWriteLock
- ordre d’acquisition:
  - non-equitable (par défaut)
  - équitable: (approximativement par ordre d’arrivée)
- Une condition est associée au writeLock
- Aucune condition n’est associée au readLock readLock().newCondition() throws UnsupportedOperatio nException.

SimpleReadWriteLock

```
1  public class SimpleReadWriteLock implements ReadWriteLock {
2    int readers;
3    boolean writer;
4    Lock lock;
5    Condition condition;
6    Lock readLock, writeLock;
7    public SimpleReadWriteLock() {
8      writer = false;
9      readers = 0;
10     lock = new ReentrantLock();
11     readLock = new ReadLock();
12     writeLock = new WriteLock();
13     condition = lock.newCondition();
14   }
15   public Lock readLock() {
16     return readLock;
17   }
18   public Lock writeLock() {
19     return writeLock;
20   }
```

Suite…

```
21    class ReadLock implements Lock {
22      public void lock() {
23        lock.lock();
24        try {
25          while (writer) {
26            condition.await();
27          }
28          readers++;
29        } finally {
30          lock.unlock();
31        }
32      }
33      public void unlock() {
34        lock.lock();
35        try {
36          readers--;
37          if (readers == 0)
38            condition.signalAll();
39        } finally {
40          lock.unlock();
41        }
42      }
43    }
```

suite

```
44    protected class WriteLock implements Lock {
45      public void lock() {
46        lock.lock();
47        try {
48          while (readers > 0 || writer) {
49            condition.await();
50          }
51          writer = true;
52        } finally {
53          lock.unlock();
54        }
55      }
56      public void unlock() {
57        lock.lock();
58        try {
59         writer = false;
60         condition.signalAll();
61        } finally {
62          lock.unlock();
63        }
64      }
65    }
66  }
```

FifoReadWriteLock

```
1  public class FifoReadWriteLock implements ReadWriteLock {
2    int readAcquires, readReleases;
3    boolean writer;
4    Lock lock;
5    Condition condition;
6    Lock readLock, writeLock;
7    public FifoReadWriteLock() {
8      readAcquires = readReleases = 0;
9      writer   = false;
10     lock = new ReentrantLock(true);
11     condition = lock.newCondition();
12     readLock = new ReadLock();
13     writeLock = new WriteLock();
14   }
15   public Lock readLock() {
16     return readLock;
17   }
18   public Lock writeLock() {
19     return writeLock;
20   }
21   ...
22 }
```

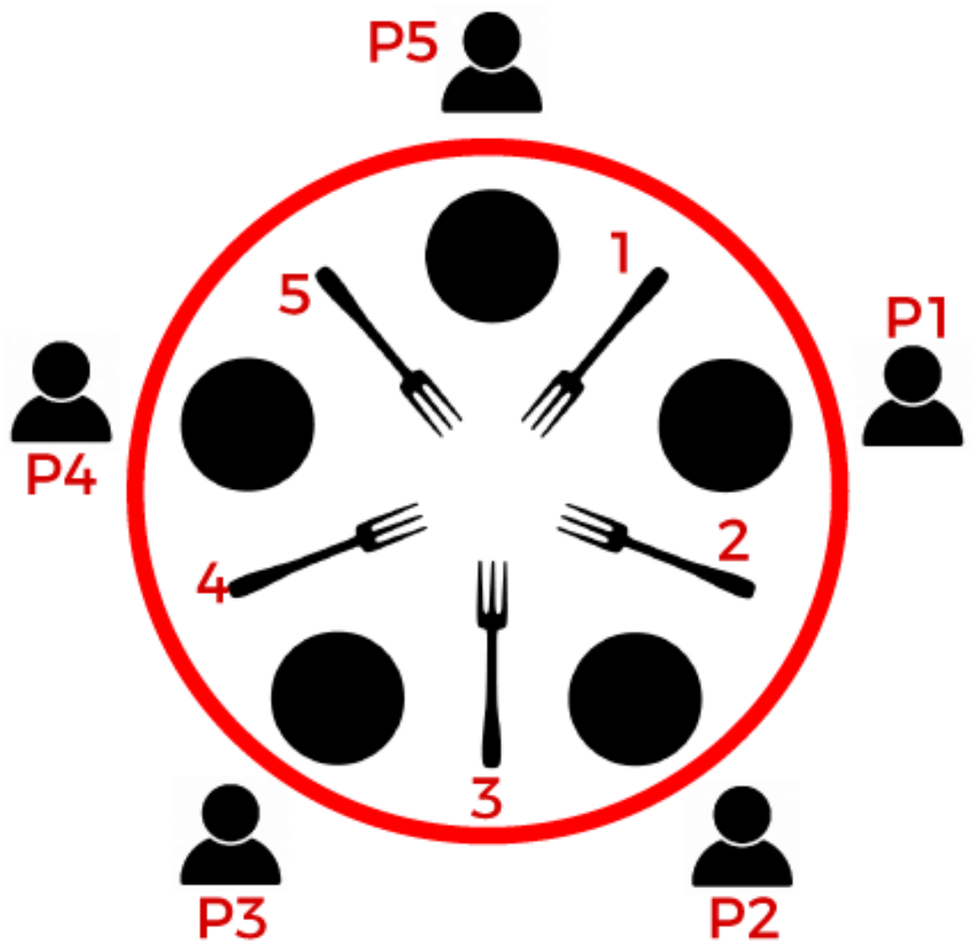
ReadLock

```
23    private class ReadLock implements Lock {
24      public void lock() {
25        lock.lock();
26        try {
27          while (writer) {
28            condition.await();
29          }
30          readAcquires++;
31        } finally {
32          lock.unlock();
33        }
34      }
35      public void unlock() {
36        lock.lock();
37        try {
38          readReleases++;
39          if (readAcquires == readReleases)
40            condition.signalAll();
41        } finally {
42          lock.unlock();
43        }
44      }
45    }
```

WriteLock

```
46    private class WriteLock implements Lock {
47      public void lock() {
48        lock.lock();
49        try {
50          while (writer) {
51            condition.await();
52          }
53          writer = true;
54          while (readAcquires != readReleases) {
55            condition.await();
56          }
57        } finally {
58          lock.unlock();
59        }
60      }
61      public void unlock() {
62        writer = false;
63        condition.signalAll();
64      }
65    }
```

Philosophes



Pour manger un philosophe Pi
a besoin de ses 2 fourchettes i et i+1

i.lock(); (I+1).lock(). -> pb