

Programmation répartie

quelques rappels Java

Rappels Java

- Processus versus thread
 - Processus: propre environnement d' 'exécution, espace mémoire propre, communication par IPC (ProcessBuilder, Process)
 - Thread (processus léger): exécution
 - un process contient au moins une thread, « main » thread
 - interne à la plateforme Java

ProcessBuilder...

```
public class ProcessDemo {
    public static void main(String args[])
        throws InterruptedException, IOException
    {
        ProcessBuilder builder = new ProcessBuilder("ls");
        //l'environnement
        Map<String, String> environ = builder.environment();
        // créer le process
        final Process process = builder.start();
        //récupérer l'inputstream du process crée
        InputStream is = process.getInputStream();
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        String line;
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
        System.out.println("Program terminé");
    }
}
```

quelques rappels java

- Thread, Runnable:
 - méthode run()
 - méthodes start(), join();
 - yield()

```
1  public static void main(String[] args) {
2      Thread[] thread = new Thread[8];
3      for (int i = 0; i < thread.length; i++) {
4          final String message = "Hello world from thread" + i;
5          thread[i] = new Thread(new Runnable() {
6              public void run() {
7                  System.out.println(message);
8              }
9          });
10     }
11     for (int i = 0; i < thread.length; i++) {
12         thread[i].start();
13     }
14     for (int i = 0; i < thread.length; i++) {
15         thread[i].join();
16     }
17 }
```

Quelques rappels Java

```
class Counter {  
    private int c = 0;  
  
    public void increment() {  
        c++;  
    }  
  
    public void decrement() {  
        c--;  
    }  
  
    public int value() {  
        return c;  
    }  
}
```

C++:

- copier la valeur de c
- incrémenter cette valeur
- stocker le résultat dans c

avec synchronized

```
public class SynchronizedCounter {  
    private int c = 0;  
  
    public synchronized void increment() {  
        c++;  
    }  
  
    public synchronized void decrement() {  
        c--;  
    }  
  
    public synchronized int value() {  
        return c;  
    }  
}
```

ThreadId

```
1 public class ThreadID {
2     private static volatile int nextID = 0;
3     private static class ThreadLocalID extends ThreadLocal<Integer> {
4         protected synchronized Integer initialValue() {
5             return nextID++;
6         }
7     }
8     private static ThreadLocalID threadID = new ThreadLocalID();
9     public static int get() {
10         return threadID.get();
11     }
12     public static void set(int index) {
13         threadID.set(index);
14     }
```

ThreadLocal

quelques rappels java

- moniteur
- wait, notify notifyAll

```
public class ThreadA {
    public static void main(String[] args){
        ThreadB b = new ThreadB();
        b.start();

        synchronized(b){
            try{
                System.out.println("Waiting for b to complete...");
                b.wait();
            }catch(InterruptedException e){
                e.printStackTrace();
            }

            System.out.println("Total is: " + b.total);
        }
    }
}

class ThreadB extends Thread{
    int total;
    @Override
    public void run(){
        synchronized(this){
            for(int i=0; i<100 ; i++){
                total += i;
            }
            notify();
        }
    }
}
```

atomicité

- **atomicité:**

- lecture et écriture pour les variables des types primitifs sauf long et double
- lecture et écriture pour les variables déclarées comme volatile

- **happens before** (Spécifié dans le chapitre 17 du [Java Language Specification](#))

- *Each action in a thread happens-before every action in that thread that comes later in the program's order.*
- *An unlock (synchronized block or method exit) of a monitor happens-before every subsequent lock (synchronized block or method entry) of that same monitor. And because the happens-before relation is transitive, all actions of a thread prior to unlocking happen-before all actions subsequent to any thread locking that monitor.*
- *A write to a volatile field happens-before every subsequent read of that same field. Writes and reads of volatile fields have similar memory consistency effects as entering and exiting monitors, but do not entail mutual exclusion locking.*
- *A call to start on a thread happens-before any action in the started thread.*
- *All actions in a thread happen-before any other thread successfully returns from a join on that thread.*
- (les objets **immuables** (immutable) -qui, une fois créés ne peuvent être modifiés permettent aussi d'assurer la cohérence mémoire)

Programmation répartie

locks-moniteurs

lock et condition

(java.util.concurrent.locks)

```
1  public interface Lock {  
2      void lock();  
3      void lockInterruptibly() throws InterruptedException;  
4      boolean tryLock();  
5      boolean tryLock(long time, TimeUnit unit);  
6      Condition newCondition();  
7      void unlock();  
8  }
```

```
Lock l = ...;  
l.lock();  
try {  
    // accès à la ressource  
} finally {  
    l.unlock();  
}
```

Condition

```
1  public interface Condition {  
2      void await() throws InterruptedException;  
3      boolean await(long time, TimeUnit unit)  
4          throws InterruptedException;  
5      boolean awaitUntil(Date deadline)  
6          throws InterruptedException;  
7      long awaitNanos(long nanosTimeout)  
8          throws InterruptedException;  
9      void awaitUninterruptibly();  
10     void signal();           // wake up one waiting thread  
11     void signalAll();       // wake up all waiting threads  
12 }
```

usage

```
1  Condition condition = mutex.newCondition();
2  ...
3  mutex.lock()
4  try {
5      while (!property) { // not happy
6          condition.await(); // wait for property
7      } catch (InterruptedException e) {
8          ... // application-dependent response
9      }
10     ... // happy: property must hold
11 }
```

File avec locks et conditions

```
1  class LockedQueue<T> {
2      final Lock lock = new ReentrantLock();
3      final Condition notFull = lock.newCondition();
4      final Condition notEmpty = lock.newCondition();
5      final T[] items;
6      int tail, head, count;
7      public LockedQueue(int capacity) {
8          items = (T[])new Object[capacity];
9      }
10     public void enq(T x) {
11         lock.lock();
12         try {
13             while (count == items.length)
14                 notFull.await();
15             items[tail] = x;
16             if (++tail == items.length)
17                 tail = 0;
18             ++count;
19             notEmpty.signal();
20         } finally {
21             lock.unlock();
22         }
23     }
24     public T deq() {
25         lock.lock();
26         try {
27             while (count == 0)
28                 notEmpty.await();
29             T x = items[head];
30             if (++head == items.length)
31                 head = 0;
32             --count;
33             notFull.signal();
34             return x;
35         } finally {
36             lock.unlock();
37         }
38     }
39 }
```

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

- *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

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

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     }
```

SimpleReentrantLock

```
1 public class SimpleReentrantLock implements Lock{
2     Lock lock;
3     Condition condition;
4     int owner, holdCount;
5     public SimpleReentrantLock() {
6         lock = new SimpleLock();
7         condition = lock.newCondition();
8         owner = 0;
9         holdCount = 0;
10    }
11    public void lock() {
12        int me = ThreadID.get();
13        lock.lock();
14        try {
15            if (owner == me) {
16                holdCount++;
17                return;
18            }
19            while (holdCount != 0) {
20                condition.await();
21            }
22            owner = me;
23            holdCount = 1;
24        } finally {
25            lock.unlock();
26        }
27    }
28    public void unlock() {
29        lock.lock();
30        try {
31            if (holdCount == 0 || owner != ThreadID.get())
32                throw new IllegalMonitorStateException();
33            holdCount--;
34            if (holdCount == 0) {
35                condition.signal();
36            }
37        } finally {
38            lock.unlock();
39        }
40    }
41
42    public Condition newCondition() {
43        throw new UnsupportedOperationException("Not supported yet.");
44    }
45    ...
46 }
```

sémaphore

```
1 public class Semaphore {
2     final int capacity;
3     int state;
4     Lock lock;
5     Condition condition;
6     public Semaphore(int c) {
7         capacity = c;
8         state = 0;
9         lock = new ReentrantLock();
10        condition = lock.newCondition()
11    }
```

```
12    public void acquire() {
13        lock.lock();
14        try {
15            while (state == capacity) {
16                condition.await();
17            }
18            state++;
19        } finally {
20            lock.unlock();
21        }
22    }
23    public void release() {
24        lock.lock();
25        try {
26            state--;
27            condition.signalAll();
28        } finally {
29            lock.unlock();
30        }
31    }
32 }
```

