ISCTE IUL
Instituto Universitário de Lisboa

Coordination

Programação Concorrente e Distribuída
Parallel and Distributed Programming

2014-2014

After this class you will be able to…

- Understand why we need to coordinate threads.
- Know the basic mechanisms of coordination:
  - wait
  - notify/notifyAll
- Know how to do the general implementation of a Condition Variable.
- Have a deeper understanding of how java implements synchronization and coordination.

Part I

Producer/consumer problems:
Synchronization and coordination between threads

Synchronization and Coordination of Threads

- In the last class, we saw how we can synchronize threads to avoid interference
- We also saw that the order in which variables (shared resources) are accessed is important:
    - Without care, memory/data inconsistencies can happen
- However, safe access to shared resources may need more sophisticated forms of coordination:
    - For example in *producer-consumer* problems

A Producer of numbers

```
public class Producer extends Thread {
    private NumberContainer numberContainer;
    private int id;
    public Producer(NumberContainer nc, int id) {
        numberContainer = nc;
        this.id = id;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            numberContainer.put(i);
            System.out.println("Producer #" +
                this.id + " put: " + i);
        }
    }
}
```

The Consumer

```
public class Consumer extends Thread {
    private NumberContainer numberContainer;
    private int id;
    public Consumer(NumberContainer nc, int id) {
        numberContainer = nc;
        this.id = id;
    }
    public void run() {
        int value = 0;
        for (int i = 0; i != 10; i++) {
          value = numberContainer.get();
          System.out.println("Consumer #" +
              this.id + " got: " + value);
      }
    }
}
```

First Version of the Shared Resource

```
public class NumberContainer {
    private int contents;

    public synchronized int get() {
        return contents;
    }
    public synchronized void put(int value) {
        contents = value;
    }
}
```

Absence of Coordination between the Producer and Consumer

- The producer and consumer share one resource
- If we assume that the producer is slightly faster than the consumer:

```
…
Consumer #1 got: 3
Producer #1 put: 4
Producer #1 put: 5
Consumer #1 got: 5
…
```

- "4" was not consumed

Coordination between Threads

- In a producer-consumer problem the threads share data; the producer produces data and the consumer does something with the data (consumes it somehow)
- The two (or more) threads have to coordinate (and not only synchronize) access to the shared resource.
- The consumer should not try to consume data before a producer has produced data, and a producer should not overwrite data that has not yet been consumed.

Coordination between Threads

- The most common way to coordinate threads is testing a condition *inside a synchronized section*.
- If the condition does not hold, the thread frees the lock and waits. Other threads are thus allowed to acquire the lock and to access a related synchronized section.
- In Java, this mechanism is implemented in the methods *wait* and *notify,* or *wait* and *notifyAll*

New version of the shared resource (1st part)

```
public class NumberContainer {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        while (!available) {
            try { // põe o thread em espera e liberta cadeado
                wait();
            // se thread saiu da espera então ganhou o cadeado
            } catch (InterruptedException e) {
              //TODO: trata excepção
            }
        }
        available = false;
        // notifica os threads em espera
        notifyAll();
        return contents;
    }
```

New version of the shared resource (2nd part)

```
// …
    public synchronized void put(int value) {
        while (available) {
           try {
             // põe o thread em espera e liberta cadeado
              wait();
              // thread saiu da espera e ganhou cadeado
           } catch (InterruptedException e) {
             //TODO: tratar excepção
           }
        }
        contents = value;
        available = true;
        // notifica threads em espera
        notifyAll();
    }
}
```

Methods on the class *Object:* wait

- *wait()* – causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

The current thread must own this object's monitor (intrinsic lock). The thread releases ownership of this monitor and waits until another thread notifies one or all the threads waiting on this object's monitor.

After being notified and before returning the thread has to re-obtain ownership of the monitor.

Methods on the class *Object:* wait

Interrupts and **spurious wakeups** are possible, and this method should always be used in a loop:

```
synchronized (obj) {
    while (<condition does not hold>)
        obj.wait();
    ... // Perform action appropriate to condition
}
```

This method can only be called by a thread that is the owner of this object's monitor (inside a synchronized method or block).

Methods on the class *Object:* wait

- *wait(long timeout)* – waits to be notified. If a notification doesn’t happen in timeout milliseconds the method returns.

  wait(0) is equivalent to wait().

  When the method returns there is no information if the reason was a timeout or a notification. To distinguish both situations the programmer should keep track of time.

wait(int timeout)

```
public synchronized boolean get(int timeout) {
    long startTime = System.currentTimeMillis();
    try {
        while (!condition) {
            long timeLeft = timeout
                    - (System.currentTimeMillis() - startTime);
            wait(timeLeft);
            if(System.currentTimeMillis() –
                    startTime > timeout){
                return false;
            }
        }
        // Change condition ...
    } catch (InterruptedException e) {
        // TODO: deal with exception
    }
    return true;
}
```

Methods on the class *Object:* notify, notifyAll

- *notify* – wakes up one thread that is currently waiting (if any).
- *notifyAll* – wakes up all the threads that are currently waiting (if any).

A thread must own the monitor of the object to call notify and notifyAll.

- For more, see: http://download.oracle.com/javase/6/docs/api/java/lang/Object.html#wait%28long%29

Part II

Semaphores

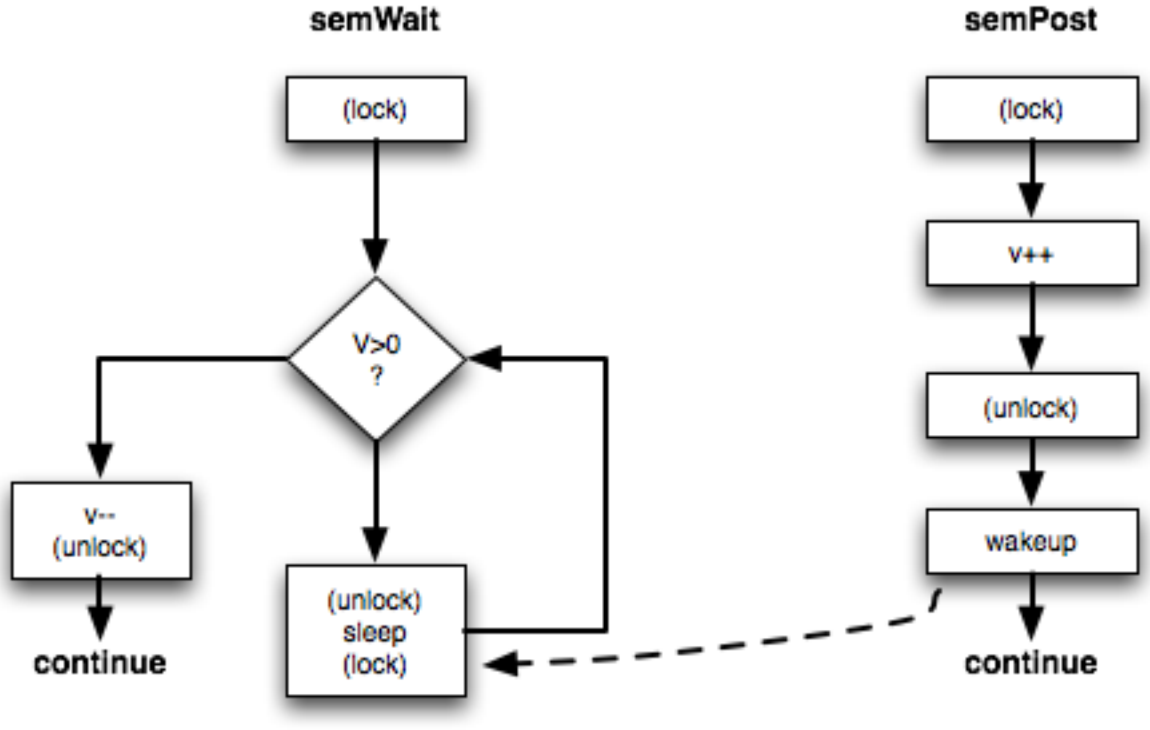
Semaphores

- The wait/notify is a particular case of the general concept of semaphore
- E. Dijkstra, Dutch scientist and world reference in computer science generalized the semaphore concept for computer programming.
- A counting semaphore is a variable that can be arbitrarily incremented but can't be decremented below zero.
- Useful when one wants threads to wait for some event that happens in succession or that can be counted.

semWait and semPost

- ***semWait*** : a thread tries to decrement the count of the semaphore. If successful, the method returns. If the count is zero, the thread will wait until another thread increments the count of the semaphore
- ***semPost***: increments the value of a semaphore and wakes up waiting threads (if any)

Decision diagram of a semaphore



Implementing semaphores in JAVA

```
// Extensions/Semaphore.java
package Extensions; // vide http://www.lambdacs.com/

public class Semaphore {
  int count = 0;

  public Semaphore(int i) {
    count = i;
  }

  public Semaphore() {
    count = 0;
  }

  public synchronized void init(int i) {
    count = i;
  }
  ...
}
```

Implementing semaphores in java (cont.)

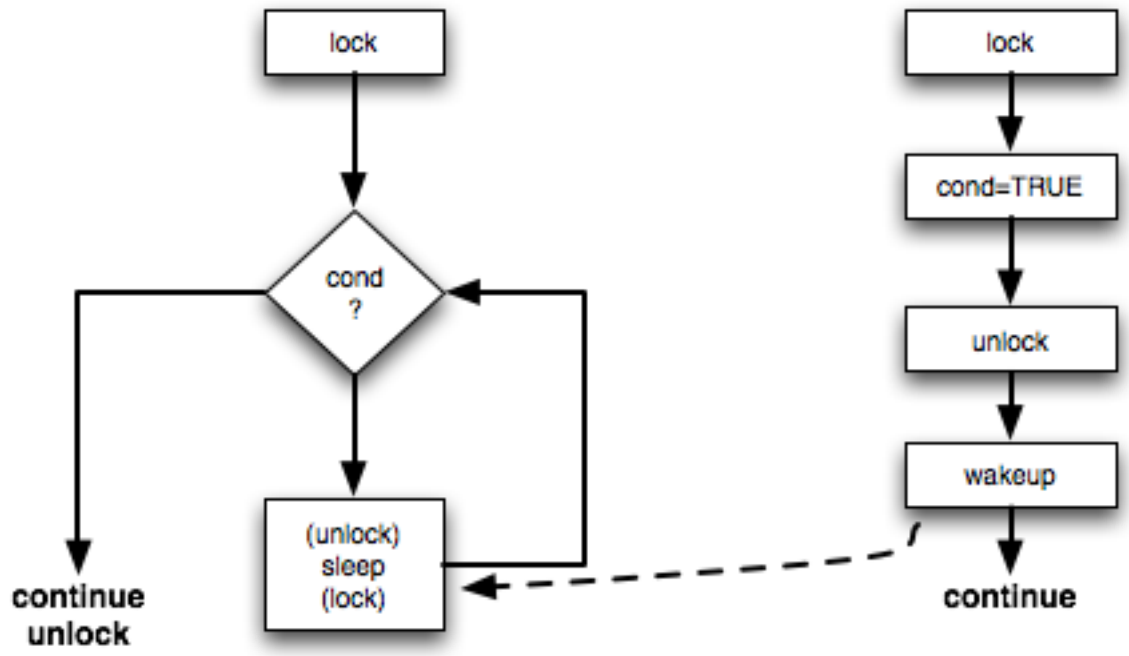
```
public class Semaphore {
  ...
  public synchronized void semWait() {
    boolean interrupted = false;
    while (count == 0) {
      try {wait();}
      catch (InterruptedException ie) {interrupted=true;}
    }
    count--;
    if (interrupted) Thread.currentThread().interrupt();
  }

  public synchronized void semPost() {
    count++;
    notify();
  }
}
```

Condition variables

- In most cases, we want to wait for a (possibly complex) condition to be true.
  - A lock can be associated with a ***condition variable***.
  - Condition variables provide a mechanism for a thread to test a condition, wait if the condition is false and to wake up when the condition may have become true.

Generalization of semaphores to arbitrary conditions



General implementation of Condition Variable

1. A thread acquires the lock associated with the condition variable.
2. Tests the condition while owning the lock.
3. If the condition is true it performs the task and releases the lock when it is done.
4. If the condition is false the lock is released and the thread waits for a change on the condition.
5. When another thread changes the logical value of the condition and releases the lock, one of the waiting threads is notified (woken up) so that it can test the condition again.

Testing condition variables

- When a thread wakes up *it has to test the condition again*. Why?

Advanced issues on synchronization

- Recursive Synchronized methods;
- Exceptions in synchronized methods;
- Static synchronized methods;
- The danger of the public attributes;
- Immutable objects;
- Synchronization vs performance;
- Volatile variable modifier.

Exercise: Explicit Locking - Lock Interface

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

```
Lock l = ...;
l.lock();
try {
   // access the resource protected by
      this lock
} finally {
   l.unlock();
}
```

To study...

- See the API for the class Object
- Read the recommended bibliography
- Exercises

Bibliography

- JAVA tutorial

http://download.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

- JAVA Threads, O'Reilly, chapter 4
- Multithreaded Programming, Lewis & Berg, chapter 6

Summary

- Thread coordination and synchronization
  - Wait and Notify mechanisms
  - The producer/consumer problem
- Semaphores
  - Counting semaphores, how they are implemented in java
  - Semaphore generalization.
  - Condition variables