JAVA Concurrency (SLR 201)

Patrick Bellot & Ada Diaconescu

Télécom ParisTech

Concurrency

- Concurrency occurs when several tasks are simultaneously executed.
- These tasks may interact which each other.
- How does a single processor execute several tasks simultaneously ?
- Nearly all software requires concurrency. Example: applications with GUI.

Concurrent execution

- A sequential program is a series of processor instructions to be executed one after the other.
- A concurrent program consists of several sequential programs, called processes, to be executed in parallel.
- On a single processor machine, concurrency is achieved by randomly interleaving the execution of instructions of the processses.
- Processes may affect each other's behavior through shared data and resources, communication, and synchronization.

Two types of processes : threads

- Two processes may execute on the same computer sharing memory.

- They are called threads or lightweight processes.

- Threads can communicate using the shared memory.

- Synchronization problems: access to data in memory, access to resources, etc.

Two types of processes: processes.

- Two processes may execute without sharing memory. They are called processes or simply programs !
- They may execute on the same computer or on different computers.
- They communicate using Inter Process Communication (IPC) resources such as sockets, pipes, etc.
- In JAVA, they can communicate using JAVA/RMI and Sockets.

Main challenges of concurrent programming

- **Synchronizing access to resources.** For instance, two threads must not write in the same file at the same time.

- **Avoiding deadlocks.** If a first thread is waiting for a second thread to do some job and the second thread is waiting for the first one to do some other job, then a deadlock occurs.

Main advantages of concurrency

- **Increased application throughput.** Parallel execution of concurrent processes allows the number of tasks completed in certain time period to increase.
- **High responsiveness for input/output.** I/O intensive applications mostly wait for input or output operations to complete.
- **More appropriate program structure.** Some problems and problem domains are well-suited to representation as concurrent tasks or processes.

Concurrency in JAVA

- Let us study how JAVA implements threads.

- When running a JAVA program, a thread is initially created for executing the static main of the program:

```
static void main(String[] args)
```

- Then, the programmer is free to create other threads.

Threads in JAVA

- **A thread in JAVA is an object of a class inheriting from the `Thread` class.**

- **The task to be executed is defined by the method**
  `void run()`

- **Practically:**
  - The programmer must create the thread object.
  - Then, it calls its `start()` method to run the thread.
  - When the `run()` method terminates, the thread terminates too.

Example

```
public class MyThread extends Thread
{
  private String threadName ;

  public MyThread(String threadName)
  {
       this.threadName = threadName ;
  }

  public void run()
  {
       for (int i = 0 ; i < 100 ; i++) {
              System.out.println(threadName + “: “ + i) ;
              try { Thread.sleep(10) ; } catch(Exception e) {}
       }
  }
}
```

Example (continued)

- To launch such threads (from the `main` method for instance) :

```
MyThread myThreadA = new MyThread(“Thread A”) ;
MyThread myThreadB = new MyThread(“Thread B”) ;

myThreadB.start() ;
myThreadA.start() ;

// Waiting for the two threads to die
try { myThreadA.join() ; } catch(Exception e) {}
try { myThreadB.join() ; } catch(Exception e) {}
```

An alternative

- An alternative is to create a class implementing the `Runnable` interface, that means having a `void run()` method.

-
  ```
  public class Task implements Runnable
  {
          public void run() { … … … }
  }
  ```

- Then, to run the task in a thread:

```
Thread thread = new Thread(new Task()) ;
thread.start() ;
```

Sleeping

- The static method `void sleep(…)` of class `Thread` causes the thread executing it to suspend execution for a specified period.
- This is an efficient means of making processor time available to the other threads of an application or to other applications that might be running on the same computer.
- Sleep times are not guaranteed to be precise, because they are limited by the facilities provided by the underlying OS.

Sleeping

- Syntax:

```
try {
      Thread.sleep(150) ; // milliseconds
} catch (Exception e) {}
```

- Why a try-catch block ?
- Several `sleep(…)` methods.

Waiting for a thread to terminate

- The method `void join(…)` allows to wait for a thread to terminate:

```
MyThread myThread = new MyThread("…") ;
myThread.start() ;
try {
    myThread.join() ;
} catch(Exception e) {}
```

- Why a try-catch block ?

- Several `join(…)` methods.

Why try-catch block ?

- A thread can be interrupted using the `interrupt()` method.
- If the thread is currently in a waiting state, for instance sleeping or waiting for another thread or waiting for an I/O, then the waiting function raises an exception.

Pause

- A thread may decide to pause herself using the method `yield()`.
- `yield()` causes the currently executing thread object to temporarily pause and allow other threads to execute.

Need for synchronization

- Interferences between threads occcur when two tasks running in different threads but acting on the same data, *interleave*.
- This means that the two tasks consist of multiple steps, and the sequences of steps overlap.
- Interferences on data can be avoided by synhronizing access to this data.

Example

- `int n = 0 ;`
- **Thread A: `n++` ; means:**
  - Get the value of `n` from memory.
  - Increments this value.
  - Stores the new value in memory
- **Thread B: `n--` ; means:**
  - Gets the value of `n` from memory.
  - Decrements this value.
  - Stores the new value in memory
- **Even very simple operations can interleave.**

Synchronized methods

- The methods of an object can be declared as synchronized.
- When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object suspend execution until the first thread is done with the object.
- Constructors cannot be synchronized.
- Call to synchronization methods have a cost.

Synchronized methods

```
public class SynchronizedCounter
{
   private int counter ;

   public synchronized void increment() { counter++ ; }

   public synchronized void decrement() { counter-- ; }

   public synchronized int value()  { return counter ; }
}
```

With synchronized declarations, method executions cannot interleave.

Synchronized methods

- JAVA allows to synchronize a method or several statements on a given object.

- When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object suspend execution until the first thread is done with the object.

Synchronized methods

- Syntax:

```
synchronized (object) {
    …
    instructions ;
    …
}
```

Synchronized methods

- A thread cannot acquire a lock owned by another thread.
- But a thread *can* acquire a lock that it already owns.
- Allowing a thread to acquire the same lock more than once enables reentrant synchronization.

Synchronization with wait and notify

- A thread waiting for a condition **MUST NOT** loop:

```
while (! Condition)   {} ;
```

- Even with a timer:

```
while (!condition) {
   try {
      Thread.sleep(100) ;
   } catch (Exception e) {}
}
```

Synchronization with wait and notify

- In JAVA, all objects have a wait-set that contains the set of threads that have executed the `wait()` method of this object.
- `wait()` must be executed in a synchronized block. However, the synchronized lock on the object is released during the wait.

- These threads are sleeping threads waiting to be awaken by a call to the `notify()` or `notifyAll()` method of the object:
  - `notify()` wakes up one thread.
  - `notifyAll()` wakes up all threads.

Example

```
public class CommandsBuffer
{
   // An array to store the commands to be executed
   private String[] commands = new String[1024] ;

   // Index where to store the next arriving command
   // Condition: (nextStoreIdx + 1) % 1024 != lastTakeIdx
   private int nextStoreIdx  = 0 ;

   // Index where to take the next command to execute
   // Condition: nextTakeIdx != nextStoreIdx
   private int nextTakeIdx  = 0 ;

   …
```

Example continued

```
public synchronized String popCommand()
{
   try {
      while (true) {

         if (nextTakeIdx == nextStoreIdx) {
            wait() ;
         } else {
            String cmd = commands[nextTakeIdx] ;
            nextTakeIdx = (nextTakeIdx + 1) % 1024 ;
            notifyAll() ;
            return cmd ;
         }

      }
   } catch (Exception e) { e.printStackTrace() ;}
   return null ;
}
```

Example continued

```
public synchronized void pushCommand(String cmd)
{
    try {
        while (true) {
            int futureStoreIdx = (nextStoreIdx + 1) % 1024 ;
            if (nextTakeIdx == futureStoreIdx) {
                wait() ;
            } else {
                commands[futureStoreIdx] = cmd ;
                nextStoreIdx = futureStoreIdx ;
                notifyAll() ;
                return ;
            }

        }
    } catch (Exception e) { e.printStackTrace() ;}
}
```

Liveness of threads (SUN Java tutorial)

- Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.

- Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress.

- A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then livelock may result.

Conclusions

- Programming concurrency is tricky.

- Be careful !