Génie logiciel pour la conception d'un Système d'Information

CSC4521

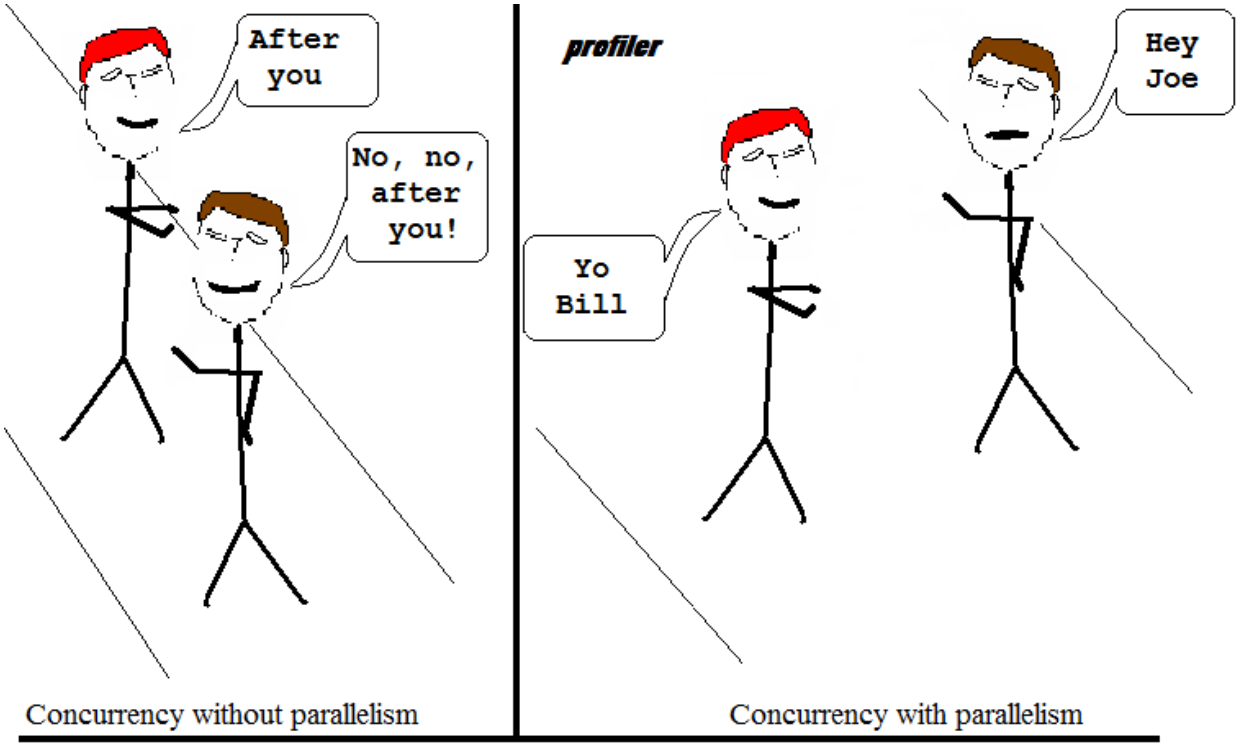
Voie d'Approfondissement
Intégration et Déploiement de Systèmes d'Information
(VAP DSI)

Threads

http://jpaulgibson.synology.me/~jpaulgibson/TSP/Teaching/CSC4521/

…/CSC4521/CSC4521-Threads.pdf

Processes and Threads



Performance tuning technique number 106: Concurrency vs. Parallelism

http://www.fasterj.com/cartoon/cartoon106.shtml

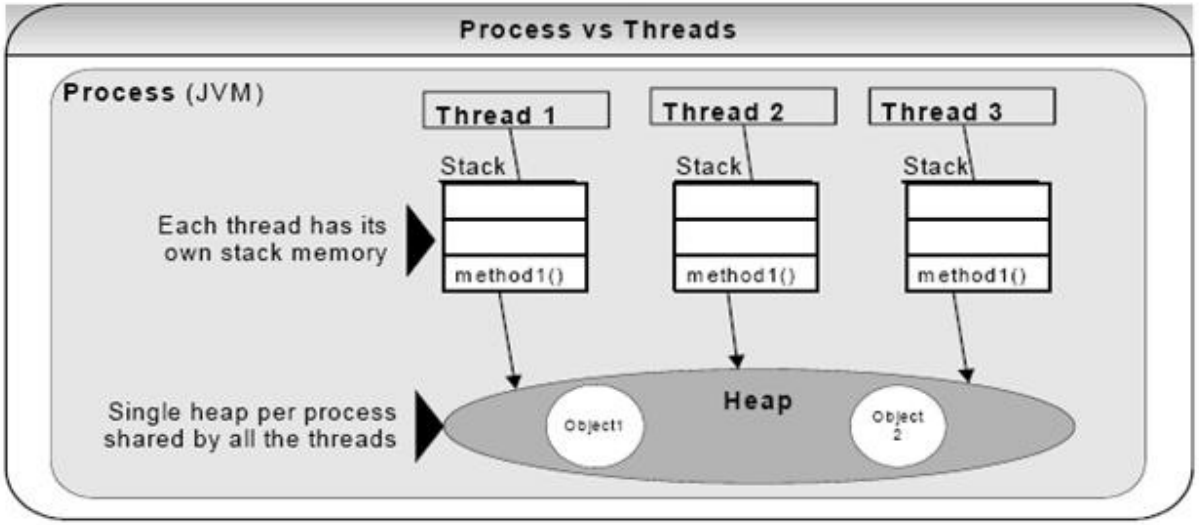
Processes and **Threads** are the two fundamental units of execution in a concurrent program. We will focus on threads as a way of simulating and testing architectures of distributed IoT processes.

Processes and Threads

•In Java, concurrent programming is mostly thread-based.

•Processing time for each core in a system is shared among processes and threads through an OS feature called time slicing.

•Concurrency is possible even on simple systems, without multiple processors or execution cores.



https://i4mk.wordpress.com/2013/03/13/processes-vs-threads/

Processes and Threads

The biggest difference between a process and a thread, is that each process has its own address space, while threads (of the same process) run in a shared memory space.

This means that it's possible to share data amongst threads (e.g. reading and write to the same variables) which should be carefully done, using synchronization on the shared data.

https://i4mk.wordpress.com/2013/03/13/processes-vs-threads/

Processes

Self-contained execution environment.

Independent set of basic run-time resources, such as memory space.

A single application may be implemented by a set of cooperating processes.

Most operating systems support *Inter Process Communication* (IPC) resources.

IPC can also used for communication between processes on different systems.

Most implementations of the JVM run as a single process, with multiple threads executing in parallel.

Threads

Also known as *lightweight processes*.

Creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one.

Threads share the process's resources, including memory and open files.

This has advantages and disadvantages … can you think of them?

Multithreaded execution is essential in Java:

- every application has at least one thread
- "system" threads that do memory management, event/signal handling, etc.

In programming, we start with just one thread, called the *main thread*.

Any thread (including the main thread) can create new threads.

Threads in Java: some additional reading

Fixing The Java Memory Model, William Pugh, 1999.

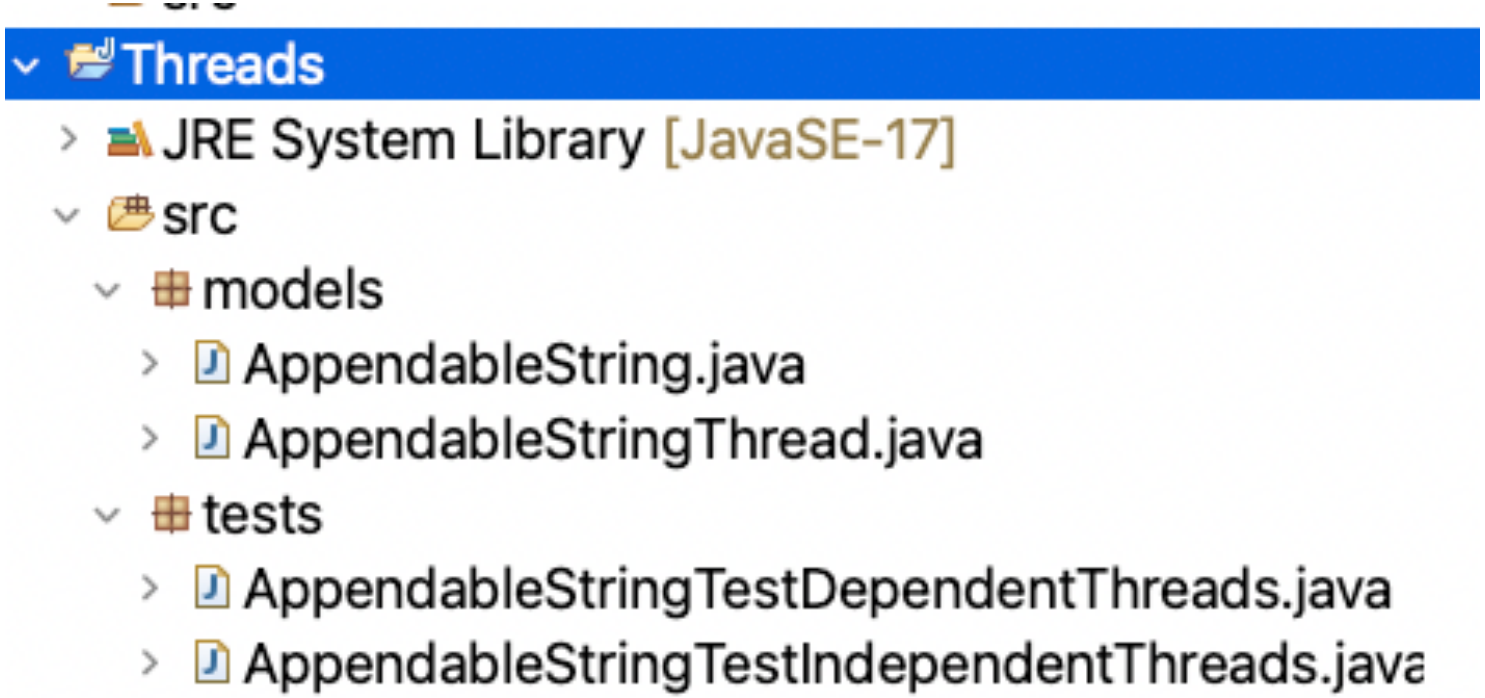
The Problem with Threads, Edward Lee, 2006.

Java Thread Programming, by Paul Hyde ISBN: 0672315858 Sams 1999

Concurrent Programming in Java™: Design, Principles and Patterns, Second Edition, By Doug Lea, ISBN: 0-201-31009-0 Addison Wesley, 2001

Thread Example

Download the code **`Threads.zip`** from the web site and import it into Eclipse



Thread Example

```
package models;

 public class AppendableString {

    public String str;

    public AppendableString(){str ="";}

    public  void append (char c){str = str + c;}

    public String toString () {return str;}
}
```

Thread Example

```
package models;
public class AppendableStringThread extends Thread {

String stringofchars;
final char INCREMENT;
final int MAX_DElAY = 1000;
final int NUMBER_OF_APPENDS=5;
AppendableString appString;

  public AppendableStringThread(String str, AppendableString appString,  char increment) {
        super(str);
        this.appString = appString;
        INCREMENT = increment;
    }
  public void run() {
        for (int appendCount = 0; appendCount < NUMBER_OF_APPENDS; appendCount++) {
            try {
                sleep((int)(Math.random() * MAX_DElAY));
            } catch (InterruptedException e) {}
            System.out.print("Thread " + getName()+ ": string "+appString);
            appString.append(INCREMENT);
            System.out.println(" extended to "+ appString );
        }
        System.out.println("Finishing thread " + getName());
    }
}
```

Thread Example

```
package tests;
import models.AppendableString;
import models.AppendableStringThread;

public class AppendableStringTestIndependentThreads {

    public static void main (String[] args) {
        System.out.println("**Testing Independent Threads**\n");
        AppendableString All1s = new AppendableString();
        AppendableString All2s = new AppendableString();
        System.out.println("Starting Thread main");
        new AppendableStringThread("Add1",All1s, '1').start();
        new AppendableStringThread("Add2",All2s, '2').start();
        System.out.println("Finishing Thread main");
    }
}
```

Thread Example

```
**Testing Independent Threads**

Starting Thread main
Finishing Thread main
Thread Add1: string  extended to 1
Thread Add2: string  extended to 2
Thread Add1: string 1 extended to 11
Thread Add2: string 2 extended to 22
Thread Add1: string 11 extended to 111
Thread Add1: string 111 extended to 1111
Thread Add1: string 1111 extended to 11111
Finishing thread Add1
Thread Add2: string 22 extended to 222
Thread Add2: string 222 extended to 2222
Thread Add2: string 2222 extended to 22222
Finishing thread Add2
```

Thread Example

```
package tests;
import models.AppendableString;
import models.AppendableStringThread;

public class AppendableStringTestDependentThreads {

    public static void main (String[] args) {
        System.out.println("**Testing Dependent Threads**\n");
        AppendableString All1sAnd2s = new AppendableString();
        System.out.println("Starting Thread main");
        new AppendableStringThread("Add1",All1sAnd2s, '1').start();
        new AppendableStringThread("Add2",All1sAnd2s, '2').start();
        System.out.println("Finishing Thread main");
    }
}
```

Thread Example

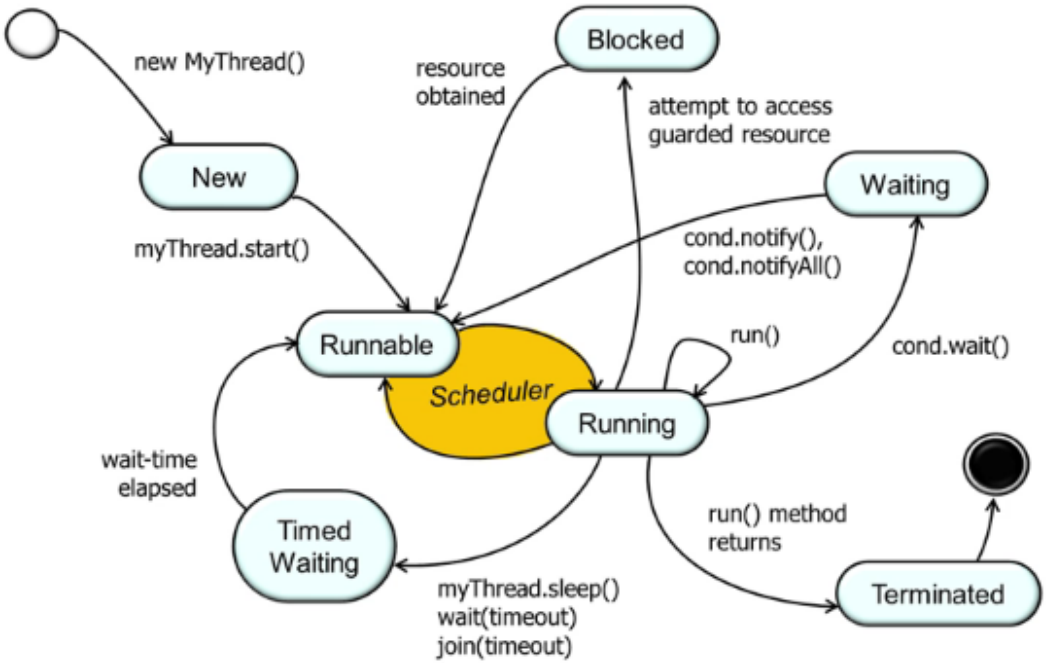
```
**Testing Dependent Threads**

Starting Thread main
Finishing Thread main
Thread Add2: string  extended to 2
Thread Add2: string 2 extended to 22
Thread Add1: string 22 extended to 221
Thread Add1: string 221 extended to 2211
Thread Add1: string 2211 extended to 22111
Thread Add2: string 22111 extended to 221112
Thread Add1: string 221112 extended to 2211121
Thread Add2: string 2211121 extended to 22111212
Thread Add2: string 22111212 extended to 221112122
Finishing thread Add2
Thread Add1: string 221112122 extended to 2211121221
Finishing thread Add1
```

Thread Example - typical output

```
Starting Thread main
Finishing Thread main
String Add2 extended to 2
String Add2 extended to 22
String Add2 extended to 222
String Add1 extended to 1
String Add1 extended to 11
String Add2 extended to 2222
String Add2 extended to 22222
No more increments left for threadAdd2
String Add1 extended to 111
String Add1 extended to 1111
String Add1 extended to 11111
No more increments left for threadAdd1
```

Thread State Machine Model



https://bitstechnotes.wordpress.com/2017/12/16/java-thread-state-diagram/

The `start()` method creates the system resources necessary to run the thread, schedules the thread to run, and calls the thread's `run()` method.

The next state state is "`Runnable`" rather than "`Running`" because the thread might not actually be running when it is in this state.

Threads and Synchronization Issues

Threads can share state (objects)

This is very powerful, and makes for very efficient inter-thread communication

However, it makes two kinds of errors possible:

- *thread interference*, and
- *memory inconsistency*.

Java provides a *synchronization "tool"* in order to avoid these types of errors.

Thread Interference

Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap. Because they are unpredictable, thread interference bugs can be difficult to detect and fix.

Consider a simple class called Counter

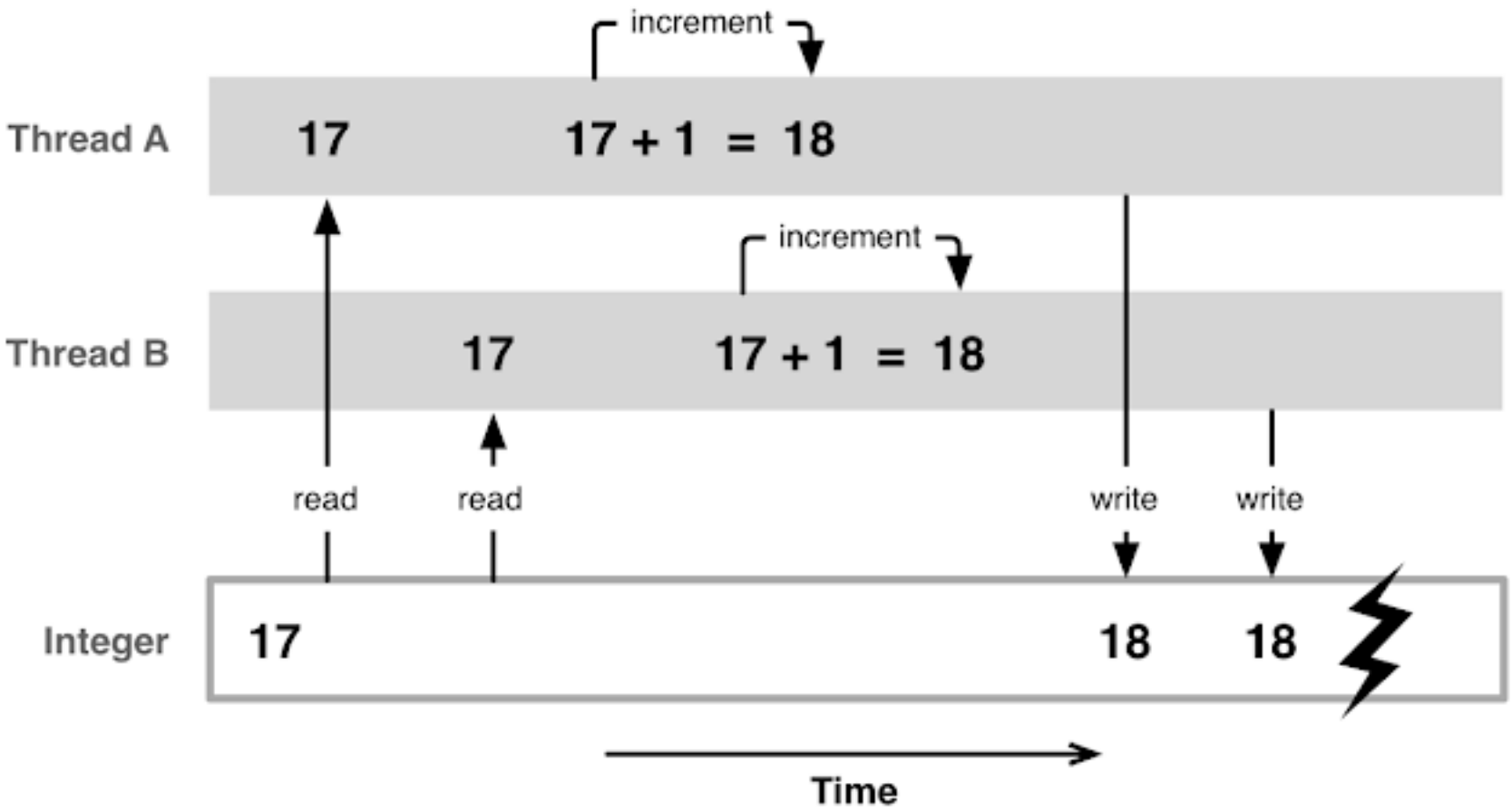
```
    class Counter {
        private int c = 0;
                public void increment() {c++;}

        /*              Multiple steps of c++
1.                    Retrieve the current value of c.
2.                  Increment the retrieved value by 1.
3.                Store the incremented value back in c.
        */


        public int value() {return c;}
    }
```

If a Counter object is referenced from multiple threads, interference between threads may give rise to unexpected behaviour.

Thread Interference



http://opensourceforgeeks.blogspot.co.uk/2014/01/race-condition-synchronization-atomic.html

Memory inconsistency

If the two increment statements had been executed in the same thread, it would be safe to assume that the values written would be “`18`" and “`19`”.

But, in this example, the values printed out might well be “`18`" and “`18`”, because there's no guarantee that thread A's change to counter will be visible to thread B — unless the programmer has established a ***happens-before*** relationship between these two statements.

There are several actions that create ***happens-before*** relationships.

The simplest technique/tool is to use ***`synchronization`***

Synchronized methods, example:

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

Two invocations of synchronized methods on the same object cannot interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

When a synchronized method exits, it automatically establishes a *happens-before* relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Synchronization is effective for keeping systems *safe*, but can present problems with *liveness*

NOTE: Java Constructors cannot be synchronized

http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Note that constructors cannot be synchronized — using the `synchronized` keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

Warning: When constructing an object that will be shared between threads, be very careful that a reference to the object does not "leak" prematurely. For example, suppose you want to maintain a `List` called `instances` containing every instance of class. You might be tempted to add the following line to your constructor:

```
instances.add(this);
```

But then other threads can use `instances` to access the object before construction of the object is complete.

Distributed Systems Have Similar Problems

With code running on different distributed processes we cannot use the synchronised mechanism in the same way.

Depending on your implementation architecture, another solution (like a re-usable pattern) may exist or you will have to manage it yourselves (if necessary)