Università degli Studi di Bologna
Facoltà di Ingegneria

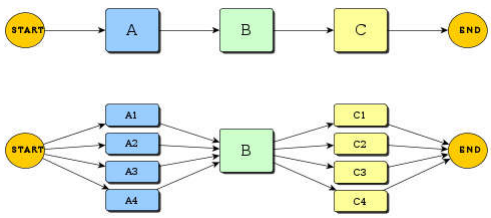
Principles, Models, and Applications for Distributed Systems M

Java Threads

Jacopo De Benedetto

CONCURRENT PROGRAMMING

Concurrent programming is a form of computing in which several computations are executing ***concurrently*** (during overlapping time periods) instead of ***sequentially*** (one completing before the next starts) processes.

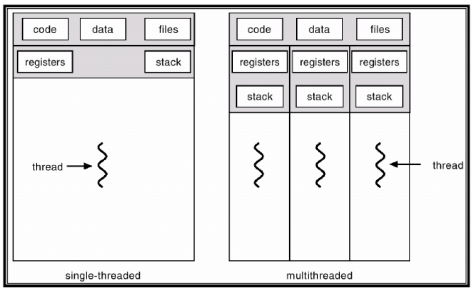


PROCESSES AND THREADS

A **process** is an instance of a computer program that is being executed. A computer **program** is a passive collection of instructions; a **process** is the actual execution of those instructions.

A **thread** (sometimes called ***lightweight process***) is the smallest sequence of programmed instructions that can be managed independently by a scheduler (typically part of the operating system)

A **process** may be made up of multiple **threads** that execute instruction concurrently.



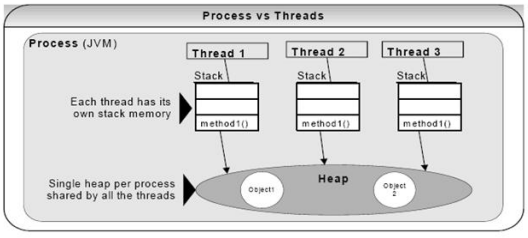
MULTITHREADING

Java is a **multithreaded programming language**.

A multithreaded program contains two or more parts that can run **concurrently**.

Threads exists within a process: every process has at least one.

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



DEFINE A THREAD

Each thread is associated with an instance of the class `Thread`.

An application that creates an instance of `Thread` must provide the code that will run in that thread.

There are two ways to do this:

- Provide a *runnable object* that **implements** the `Runnable` interface
- Extending the class `Thread` (that implements itself the Runnable interface)

IMPLEMENTING RUNNABLE INTERFACE

STEP 1: implement `run()` method

- `public void run() { … }`

STEP 2: create a `Thread` object

- `Thread t = new Thread(runnableObj, threadName);`

STEP 3: start the thread by calling `start()` method on the Thread object:

- `t.start();`

THREAD PRIORITIES

Every Java thread has a **priority** that helps the operating system determine the **order** in which threads are **scheduled**.

Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependentent.

EXTENDING THE Thread CLASS

STEP 1: extend the `Thread` class

- `class MyThread extends Thread { … }`

STEP 2: override the inherited `run()` method

- `public void run() { … };`

STEP 3: create an instance of the class and call the inherited `start()` method

- `MyThread t = new MyThread();`
- `t.start();`

`Thread.sleep()` METHOD

`Thread.sleep(long millis)`
Causes the current thread to suspend execution for a specified period.

The sleep period can be terminated by interrupts:
When another thread interrupts the current thread while **`sleep`** is active, it throws an **`InterruptedException`**

INTERRUPTS

An **interrupt** is an indication to a thread that it should stop what it is doing and do something else.

It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.

Is possible to send an interrupt to a thread by invoking **`interrupt()`** on the Thread object for the thread to be interrupted.

SUPPORTING INTERRUPTION

OPTION 1: catching the **InterruptedExeption**

```
try {
 Thread.sleep(10000); // sleep for 10 seconds
}
catch (InterruptedException e) {
 // the thread has been interrupted
 // do something
}
```

OPTION 2: verifying if the thread is interrupted

```
if (Thread.interrupted()) {
 // the thread has been interrupted
 // do something
}
```

`join()` METHOD

The `join` method allows one thread to wait for the completion of another. If t is a Thread object whose thread is currently executing,

```
t.join();
```

causes the current thread to pause execution until t's thread terminates

Like `sleep()`, `join()` responds to an interrupt by exiting with an `InterruptedException`.

THREAD SYNCHRONIZATION

Threads communicate primarily accessing shared resources. This form of communication is extremely efficient, but makes two kinds of errors possible:

- **thread interference**
  happens when two operations, running in different threads, but acting on the same data, **interleave**: the two operations consist of multiple steps, and the sequences of steps overlap.

- **memory consistency errors**
  occur when different threads have inconsistent views of what should be the same data. The causes of memory consistency errors are complex and beyond the scope of this…

SYNCHRONIZED BLOCKS

The Java programming language provides two basic synchronization idioms: **synchronized statements** and **synchronized methods**.

Declaring a statement (or a method) `synchronized` has two effects:

- is not possible for two invocations of synchronized statements on the same object to interleave. When one thread is executing a synchronized statement for an object, all other threads that invoke synchronized statements for the same object will be suspended until the first thread exits the synchronized statement;
- when the thread exits a synchronized statement, an *happens-before* relationship is automatically established with any subsequent invocation of a synchronized statement on the same object. This guarantees that changes to the state of the object are visible to all threads.

LOCKS AND SYNCHRONIZATION

Synchronization is built around an internal entity known as the **intrinsic lock** or **monitor lock** (often refers as **monitor**).

When a thread invokes a synchronized statement, it acquires the lock for the object (other threads will block when attempt to acquire the lock) and releases it when the method returns.

The lock release occurs even if the return was caused by an uncaught exception.

*Every object has only **one lock**: if the object has two or more **synchronized staments (or methods)**, they can be only executed **one at a time**!*

The term **deadlock** describes a situation where two or more threads are **blocked forever**, waiting for each other.

EXAMPLE

- **Synchronized statement**

```
synchronized(obj){
  // synchronized statement
  //the lock refers to the object obj
}
```

- **Synchronized method**

```
public synchronized void method(){
  // synchronized method
  //the lock refers to the object that owns the
  //method
}
```

`wait()`, `notify()` AND `notifyAll()`

The purpose of wait/notify mechanism is to allow synchronization between threads based on a condition.

- **`wait()`**
  Causes the current thread to wait, releasing the **monitor** for the object, until another thread invokes the `notify()` method or the `notifyAll()` method for this object or.

- **`notify()`**
  Wakes up a single thread (chosen by the scheduler) that is waiting on this object's **monitor**. If any threads are waiting on this object, one of them is chosen to be awakened (the choice depends on the JVM implementation).

- **`notifyAll()`**
  Wakes up all threads that are waiting on this object's **monitor**

LIFE CYCLE OF A THREAD