Remember Turn on Default Principal To Role Mapping in order to avoid mappings

cphbusiness

COPENHAGEN BUSINESS ACADEMY

Threads in Java

This week

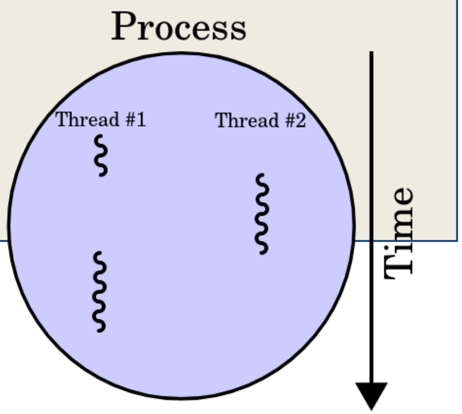
- Good news:
  - Threads are fun!
  - Threads are ridiculously useful

- Bad news:
  - Threads are hard ← but that’s why we’re here

Why multithreading?

- Responsiveness (Concurrency)
- Performance (Parallelism)

Diagram on whiteboard
* Sequential execution
* Concurrent (tasks can start, run, complete in overlapping time e.g. processes on single core)
* Parallel (tasks can run simultaneaously e.g. multicore cpu)



Why multithreading?

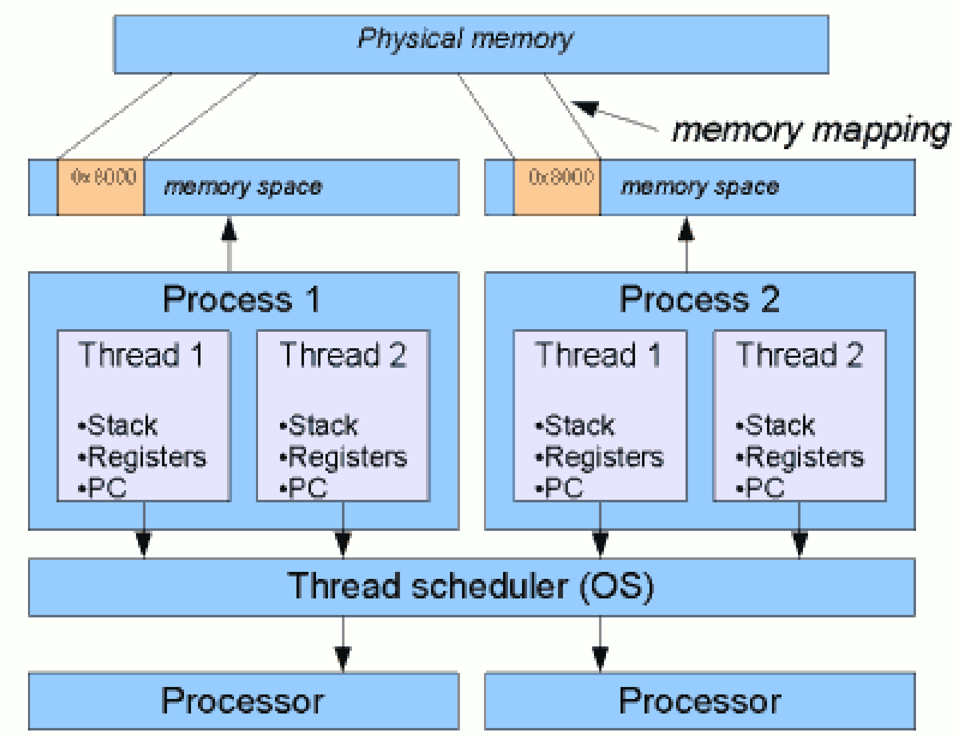
- Responsiveness (Concurrency)
- Performance (Parallelism)

Diagram on whiteboard
Sequential execution on UI thread

Typical relationship between Threads and Processes

- Each **process** has its own memory space. When Process 1 accesses some given memory location, say 0x8000, that address will be mapped to some physical memory address1. But from Process 2, location 0x8000 will generally refer to a completely different portion of physical memory.

- A **thread** is essentially a subdivision of a process that shares the memory space of its parent process.
- Each thread however has its own private stack and registers, including program counter. These are essentially the things that threads need in order to be independent



http://www.javamex.com/tutorials/threads/how_threads_work.shtml

Creating a thread in java

By inheriting from thread
(Here with an anonymous class)

```
Thread t1 = new Thread() {
        @Override
        public void run() {
                //Do stuff a new thread yall
        }
};
t1.start();
```

Creating a thread in java

By inserting a Runnable object
(Here with an anonymous class)

```
Runnable r = new Runnable(){
        @Override
        public void run() {
                //Do stuff a new thread yall
        }
};
Thread t1 = new Thread(r);
t1.start();
```

Creating a thread in java

With a lambda (anonymous function) - from java 8 and forward

```
Thread t1 = new Thread(() -> {
    //Do stuff in a separate thread!
});
t1.start();
```

Creating a thread in java

When we want one thread to wait for another, we can use the “**join**” method

```
Thread t1 = new Thread(() -> {
    //Do stuff in a separate thread!
});
t1.start();
t1.join();
...
```

The calling thread will wait for t1 to finish before it continues.

Creating a thread in java

Implementing the Runnable Interface

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

Task task1 = new Task();
Thread  t1 = new Thread(task1);
t1.start();
```

Creating a thread in java

Demo time!

Let’s print some numbers from different threads.

The Problem with Threads

- Non-determinism
  - Event order uncertain
  - Visibility of data-changes uncertain

We can fix all of that!
But the solutions create new problems.

Race conditions

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.

```
class MutableInteger {
    private int i = 0;
    public int get() {
        return i;
    }

    public void increment() {
        this.i++;
    }
}
```

Even simple statements can translate to multiple steps by the virtual machine

1. Retrieve the current value of i.
2. Increment the retrieved value by 1.
3. Store the incremented value back in

Suppose both Thread A and Thread B invokes increment at about the same time

1. Thread A: Retrieve i.
2. Thread B: Retrieve i.
3. Thread A: Increment retrieved value; result is 1.
4. Thread B: Increment retrieved value; result is 1.
5. Thread A: Store result in i; i is now 1.
6. Thread B: Store result in i; i is now 1.

Race conditions

Demo time!

Synchronization

Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible:

- –*Race conditions (interleaving access)*
- –*Memory Consistency Errors*.

The tool needed to prevent these errors is *synchronization*.

Synchronization

What we’ve seen in the demos so far are examples of **race conditions**.

Race condition: When the behaviour of a program depends on the ordering/interleaving of threads, *and shouldn’t.*

Race conditions can occur when multiple threads perform actions that are not ***atomic*** with respect to each other.

Synchronization

An operation is ***atomic*** if it has no “intermediary” state. It is either fully completed, or hasn’t begun at all.

Hardly anything is atomic in java.

Synchronization

Is integer increment atomic?

```
j++
```

Synchronization

Nope.

```
++i
```

But variable read and writes are ATOMIC:

```
int j = i;
j = i + 1;
i = j;
return i;
```

Synchronization

In Java, the following are atomic:

- Reads/writes of pointers/ object references
- Reads/writes of *single precision* primitive types

As always, consult the official specifications when you can’t afford to guess.
https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html

Synchronized Block

```
public synchronized void add(int value)
{
    ...
}
```

- Code in the Synchronized Block can only execute if the lock is free.
- If a Thread executes code inside the block the lock is taken, other callers will be blocked until the lock is free.

Synchronized Block

synchronized blocks are ***atomic*** with respect to other blocks that take the same lock!

Synchronized Block

Demo time!

https://github.com/HartmannDemoCode/Sem3

Multi-threading with Swing

Lengthy calculation can halt the GUI

when the calculation is done in the same thread as the GUI it can block the GUI in a way that makes it not responding to the user.
Solution: do work in other thread.

```
private void btnCalculate2ActionPerformed(java.awt.event.ActionEvent evt) {
        long in = Long.parseLong(txt1.getText());
        FibCalc f = new FibCalc(in, this);
        Thread t1 = new Thread(f);
        t1.start();
    }
```

Make objects accessable from threads

Use the constructor

```
public class MyThread extends Thread {

    private Counter counter;

    public MyThread(Counter counter) {
        this.counter = counter;
    }

    @Override
    public void run() {
        counter.increment();
    }
}
```

java.util.concurrent.atomic.*

This library provides atomic objects to use like:

- Integer
- Boolean
- Reference

Presenting Thread1 exercise

Find the exercises and class material here:
https://github.com/Cphdat3sem2018f/week1-threads