ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Parallelism on the JVM

Parallel Programming in Scala

Aleksandar Prokopec

JVM and parallelism

There are many forms of parallelism.

JVM and parallelism

There are many forms of parallelism.

Our parallel programming model assumption – multicore or multiprocessor systems with shared memory.

JVM and parallelism

There are many forms of parallelism.

Our parallel programming model assumption – multicore or multiprocessor systems with shared memory.

Operating system and the JVM as the underlying runtime environments.

Processes

Operating system – software that manages hardware and software resources, and schedules program execution.

Processes

Operating system – software that manages hardware and software resources, and schedules program execution.

Process – an instance of a program that is executing in the OS.

Processes

Operating system – software that manages hardware and software resources, and schedules program execution.

Process – an instance of a program that is executing in the OS.

The same program can be started as a process more than once, or even simultaneously in the same OS.

Processes

Operating system – software that manages hardware and software resources, and schedules program execution.

Process – an instance of a program that is executing in the OS.

The same program can be started as a process more than once, or even simultaneously in the same OS.

The operating system multiplexes many different processes and a limited number of CPUs, so that they get *time slices* of execution. This mechanism is called *multitasking*.

Processes

Operating system – software that manages hardware and software resources, and schedules program execution.

Process – an instance of a program that is executing in the OS.

The same program can be started as a process more than once, or even simultaneously in the same OS.

The operating system multiplexes many different processes and a limited number of CPUs, so that they get *time slices* of execution. This mechanism is called *multitasking*.

Two different processes cannot access each other's memory directly – they are isolated.

Threads

Each process can contain multiple independent concurrency units called *threads*.

Threads

Each process can contain multiple independent concurrency units called *threads*.

Threads can be started from within the same program, and they share the same memory address space.

Threads

Each process can contain multiple independent concurrency units called *threads*.

Threads can be started from within the same program, and they share the same memory address space.

Each thread has a program counter and a program stack.

Threads

Each process can contain multiple independent concurrency units called *threads*.

Threads can be started from within the same program, and they share the same memory address space.

Each thread has a program counter and a program stack.

JVM threads cannot modify each other's stack memory. They can only modify the heap memory.

Creating and starting threads

Each JVM process starts with a **main thread**.

Creating and starting threads

Each JVM process starts with a **main thread**.

To start additional threads:

1. Define a `Thread` subclass.
2. Instantiate a new `Thread` object.
3. Call `start` on the `Thread` object.

The `Thread` subclass defines the code that the thread will execute. The same custom `Thread` subclass can be used to start multiple threads.

Example: starting threads

```
class HelloThread extends Thread {
  override def run() {
    println(”Hello world!”)
  }
}

val t = new HelloThread

t.start()
t.join()
```

Time for a demo!

Atomicity

The previous demo showed that separate statements in two threads can overlap.

In some cases, we want to ensure that a sequence of statements in a specific thread executes at once.

Atomicity

The previous demo showed that separate statements in two threads can overlap.

In some cases, we want to ensure that a sequence of statements in a specific thread executes at once.

An operation is *atomic* if it appears as if it occurred instantaneously from the point of view of other threads.

Atomicity

The previous demo showed that separate statements in two threads can overlap.

In some cases, we want to ensure that a sequence of statements in a specific thread executes at once.

An operation is *atomic* if it appears as if it occurred instantaneously from the point of view of other threads.

Let's see a demo:

```scala
private var uidCount = 0L
def getUniqueId(): Long = {
  uidCount = uidCount + 1
  uidCount
}
```