Multithreading

In Java

Lukas Wais

July 7, 2020

1 Introduction
- Definitions
- Process & Program
- Threads
- Multithreading
- Problems with Multithread

2 Threads in Java
- How to create Threads
- Solutions for the Multi User Problems

3 Outlook

Do you know any of these?

- Program
- Process
- Thread

Program versus Thread

- **Programs** are the sets of instructions and (static) data that describe how to execute a certain task (job?)
- **Processes** are instances of running programs.

Program versus Process

Informal

- **Programs** = recipe
- **Processes** = cooking the actual meal

Process states

A process can have 3 different states:

1. Ready: runnable, but temporarily stopped in favour of another running process
2. Running: using the CPU
3. Blocked: waiting for external event (e.g. I/O interrupt)

Process States

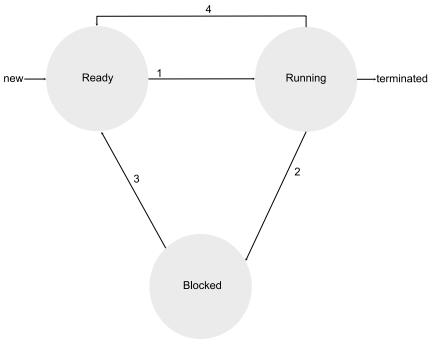

Figure: Process State Model

Thread

Definition

Every Process has at least one thread. Threads expand the process model to have multiple parallel tasks, which are sharing the **same** address space. You can use the same variables in a Java class in two different threads.

Thread = Ausführungsfaden.

Why Threads

Important

Processes each have their own address space

Why Threads

- There may be multiple processes running the same program, but they do not share any internal state.
- **Multiple parallel tasks share one address space makes programming simpler.**
- Example: Word Processor:
  - Reading data from keyboard
  - Writing
  - Spell checking
- all running at the same time (with a single CPU, running interleaved = verschachtelt) and not blocking each other

Why Multithreading?

- Splitting up computation heavy tasks.
- User Interfaces.
- Client Server applications $\rightarrow$ Sender/Receiver threads.

More Threads more Problems

Similar to multi user databases and other multi user systems.

- Deadlock
- Starvation (more a CPU problem)
- Race Condition

Race Condition

- Wikipedia article

We have two different possibilities to create new threads in Java:

- Create a *Thread* subclass.
- Implement the *Runnable* interface.
  - Anonymous implementation of *Runnable*.
  - Lambda implementation of *Runnable*.

The are no rules wheter you should extend the *Thread* class or implement *Runnable*. Both are working fine.

extends Thread

```
public class MyThreads extends Thread {

  @Override
  public void run() {
    System.out.println("This thread started");
  }

  public static void main(String[] args) {
    new MyThreads().start();
  }
}
```

Listing 1: extends and Override run

extends Thread

```
public class MyThreads {

  public static void main(String[] args) {
    Thread myThread = new Thread(() -> {
      System.out.println("This thread started");
    });
    myThread.start();
  }
}
```

Listing 2: extends and Override run with lambda

implements Runnable

```
public class MyThreads implements Runnable {

  @Override
  public void run() {
    System.out.println("This thread started");
  }
  public static void main(String[] args) {
    new MyThreads().run();
  }
}
```

Listing 3: implements Runnable

implements Runnable

```
public class MyThreads {

  public static void main(String[] args) {
    Runnable runnable = () ->  System.out.println("Thread started");
    runnable.run();
  }
}
```

Listing 4: implements Runnable with lambda. **functional intferace**

Locks

- synchronized blocks
- java.util.concurrent, CopyOnWriteArrayList
- volatile variables
- java.util.concurrent.atomic, AtomicInteger

What we are going to do next

- ExecutorService()
- ThreadPool()

Use those locks sparingly

Keep in mind, those locks are **locking resources**. You may revert your performance boost of multiple threads. At worst you are creating deadlocks.