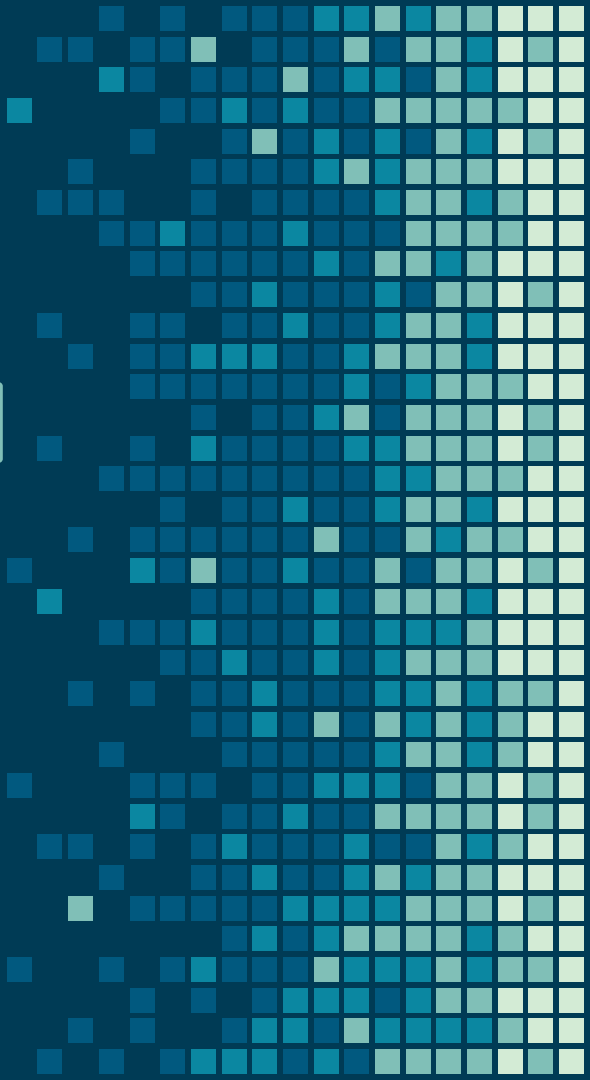
Multithreading basics – part 1

Parallelism, Concurrency, Threads

Tamás Ambrus



Terminology

Parallelism, Concurrency

Parallelism vs. Concurrency

These are two very different concepts but they are often meshed together in imperative programming because they only implement concurrency but not parallelism. Parallelism comes up mainly in some domain specific languages or pure functional languages like Haskell.

Parallelism: parallel execution of a program that is **deterministic**: always yields the same output for the same program.

Concurrency: decomposing a program into order-independent pieces so that the parts of it may run in parallel. Can be **highly non-deterministic** because of task scheduling.

More info at https://stackoverflow.com/questions/1050222/what-is-the-difference-between-concurrency-and-parallelism

Example – parallel code

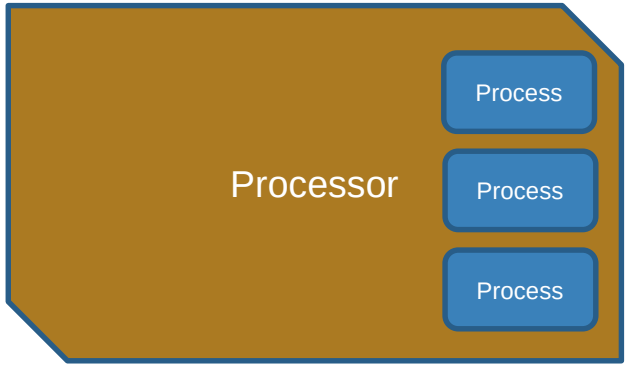
Here's a truly parallel code, written in Haskell:

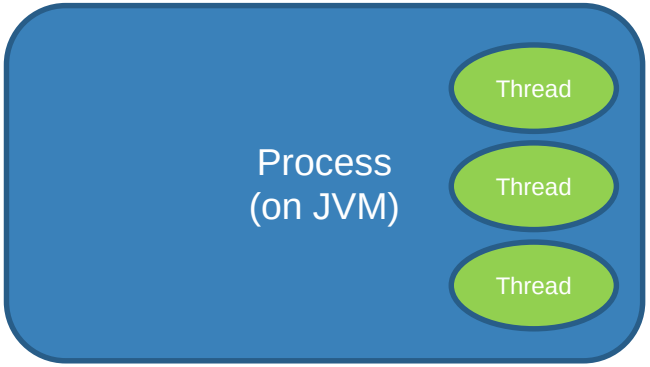
```
parfib :: Int -> Par Int
parfib n
  | n <= 2 = return 1
  | otherwise = do
    x <- spawn $ parfib (n-1)
    y <- spawn $ parfib (n-2)
    x' <- get x
    y' <- get y
    return (x' + y')
```

Task scheduling

Concurrency: decomposing a program into order-independent pieces so that the parts of it may run in parallel. **Highly non-deterministic** because of task scheduling.

Processes (and threads within it) that run on the same processor, have to share running time. Therefore each needs to be scheduled with a tiny piece of time. Let's refer to this as **task scheduling**.





Task scheduling

So who decides when to run a thread for a little while? The platform.

- the scheduling is so random, no one can tell the order
- your threads may finish in a different order you would expect
- so task scheduling makes your application non-deterministic, each run can end in a different result
- that – as you will see – is the cons of concurrency

More info at https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html

Pros of concurrency

- speeds up computations
- increasing program throughput: different parts of our program that don't depend on each other may run in parallel
- high responsiveness for IO: IO is slow, let's do some computations on other CPU cores while we are waiting for data on another
- some problems are well suited for a concurrent algorithm
- faking parallel execution: if we only have 1 CPU core, let's not block the whole operating system waiting for user input

Threads

Thread, Runnable, wait, notify, interruption

Thread

In Java, threads are the lowest level of concurrency.

- each thread has its own stack
- but all threads within the same process share the **SAME** memory

There is two way to define a type that can behave as a thread:

- extending the `Thread` class, override the `run()` method
- implementing `Runnable` interface, override the `run()` method
  - and then pass to a Thread object

More info at https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

Starting a thread

```java
public class Worker extends Thread {

    @Override
    public void run() {
        System.out.println("Hello ");
    }

    public static void main(String[] args) {
        Worker w = new Worker();
        w.start();
        System.out.println("world!");
    }

}
```

OR

```java
public class Worker implements Runnable {

    @Override
    public void run() {
        System.out.println("Hello ");
    }

    public static void main(String[] args) {
        Thread t = new Thread(new Worker());
        t.start();
        System.out.println("world!");
    }

}
```

More info at https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html

Waiting for a thread

If a thread finishes its run() method, it terminates.

- reaches its final state: TERMINATED

All Java applications have at least 1 thread: „main”

- sometimes a thread has to wait on one/many other(s) to continue its job
- for pausing a thread until another one terminates

we use Thread::join

```
@Override
public void run() {
    System.out.println("Hello ");
}

public static void main(String[] args) {
    List<Thread> threads = new ArrayList<>();
    for (int i = 0; i < 10; ++i) {
        threads.add(new Thread(new Worker()));
    }
    threads.forEach(Thread::start);
    threads.forEach(t -> {
        try {
            t.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
    System.out.println("world!");
}
```

More info at https://docs.oracle.com/javase/8/docs/api/index.html?java/lang/Thread.State.html

Example – synchronization (wrong)

```
public class ConcurrencyGoneWrong {

    private static int counter = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 1_000_000; i++) {
                    counter++;
                }
            }
        });
        t.start();

        for (int i = 0; i < 1_000_000; i++) {
            counter--;
        }
        t.join();

        System.out.println(counter);
    }
}
```

Remember: threads within the same process use the same memory.

these are not atomic operations, thus the scheduler may interrupt them

Example – synchronization (wrong)

```java
public class ConcurrencyGoneWrong {

    private static int counter = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 1_000_000; i++) {
                    counter++;
                }
            }
        });
        t.start();

        for (int i = 0; i < 1_000_000; i++) {
            counter--;
        }
        t.join();

        System.out.println(counter);
    }
}
```

- most people think that the scheduler lets run a single line like that without interruption

- but `counter++` means this:
  - getting the value of `counter`
  - add 1 to it
  - store the incremented value

- so it's not atomic, furthermore, almost nothing is atomic
  - the only atomic operations are reading/writing at most 32 bit primitive variables and references

- that's why we need the `synchronized` keyword

More info at https://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html

Example – synchronization (correct)

```
public class ConcurrencyDoneRightAndSlow {

    private static Object monitor = new Object();
    private static int counter = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(() -> {
            for (int i = 0; i < 1_000_000; i++) {
                synchronized (monitor) {
                    counter++;
                }
            }
        });
        t.start();

        for (int i = 0; i < 1_000_000; i++) {
            synchronized (monitor) {
                counter--;
            }
        }
        t.join();

        System.out.println(counter);
    }
}
```

Remember: threads within the same process use the same memory.

Synchronization, monitors

Monitors in concurrent programming allow **mutual exclusion** between concurrently running processes (or threads in this case).

- if one thread is executing a **critical block** no other thread may enter a critical block of the same monitor

How do we create a monitor in Java? Nothing's easier: every object in Java is itself a monitor. All we need is to use them with the `synchronized` keyword.

More info at https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Synchronization, monitors

```
public class SynchronizedRules {

    public synchronized void instanceMethod() {
        // ...
    }
    // is equal to
    public void instanceMethod2() {
        synchronized (this) {
            // ...
        }
    }



    public static synchronized void staticMethod() {
        // ...
    }
    // is equal to
    public static void staticMethod2() {
        synchronized (SynchronizedRules.class) {
            // ...
        }
    }

}
```

Every code piece that uses a resource (e.g. a class field) that may be reached concurrently, should be put in a synchronized block in order to avoid thread interference.

Example – synchronization, wait, notifyAll

```
public class ConcurrentQueue<T> {

    private final List<T> queue;
    private final int capacity;

    public ConcurrentQueue(int capacity) {
        this.queue = new ArrayList<>(capacity);
        this.capacity = capacity;
    }

    public synchronized void enqueue(T elem) throws InterruptedException {
        while (queue.size() >= capacity) {
            this.wait();
        }
        queue.add(elem);
        this.notifyAll();
    }

    public synchronized T dequeue() throws InterruptedException {
        while (queue.isEmpty()) {
            this.wait();
        }
        T elem = queue.remove(0);
        this.notifyAll();
        return elem;
    }
}
```

Wait, notify, notifyAll

There is not much left you haven't seen and learned from type Object:

- wait(), wait(long), wait(long, int)
- notify(), notifyAll()

The wait methods send their invoker thread to WAITING or TIMED_WAITING state from RUNNABLE state.

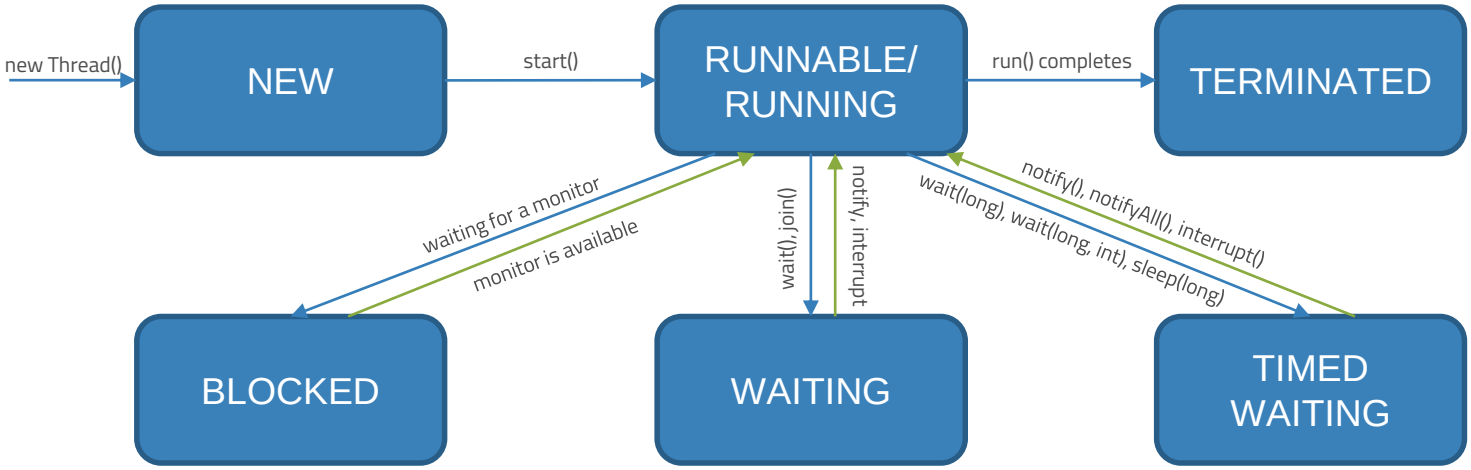
The thread keeps waiting until another thread calls some notify on the very same object on which this thread is waiting. (→ RUNNABLE)

Requirement: all code pieces that call wait or notify must be in a synchronized block. That synchronized block's monitor must be the object on which wait or notify is called.

More info at https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

Thread states

So far we have seen all states that a thread is able to have:



More info at https://docs.oracle.com/javase/8/docs/api/index.html?java/lang/Thread.State.html

Interruption

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

- calling on a thread: t.interrupt()
- how you make your thread notice interruption:

```
public class Worker implements Runnable {

    @Override
    public void run() {
        boolean running = true;
        while (running) {
            doSomethingWithoutWaitingOrTimedWaiting();
            if (Thread.interrupted()) {
                running = false;
            } else {
                try {
                    Thread.sleep(TEN_SECONDS);
                } catch (InterruptedException e) {
                    running = false;
                    Thread.currentThread().interrupt();
                }
            }
        }
    }
```

Methods like join(), sleep() or wait() may throw InterruptedException

More info at https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#interrupt--