Concurrency in Java

By Tomasz Mozolewski @2011

Why bother?

- Multi core / processor machines
- Processor hitting limit on performance
- Complexity of algorithms and requirements
- Appetite Grows with Eating

When to use concurrent threads

- Performance
  - CPU is the bottle neck
  - Won't overload other systems
  - Parallel access to other systems
- Scalability*
  - High number of independent requests
  - Future growth

* More things need to be considered for scalability

When to avoid

- If serial solution is fast enough
- Benefit of concurrent processing is small
- Lack of experienced developers

What if I don’t use it?

- Frameworks will use
- Web container and servlets
- Singletons – not all are thread safe
- Datasources and other resource pools

Main pitfalls

- Dead locks
- Starvation
- Live locks
- Race condition

Dead locks

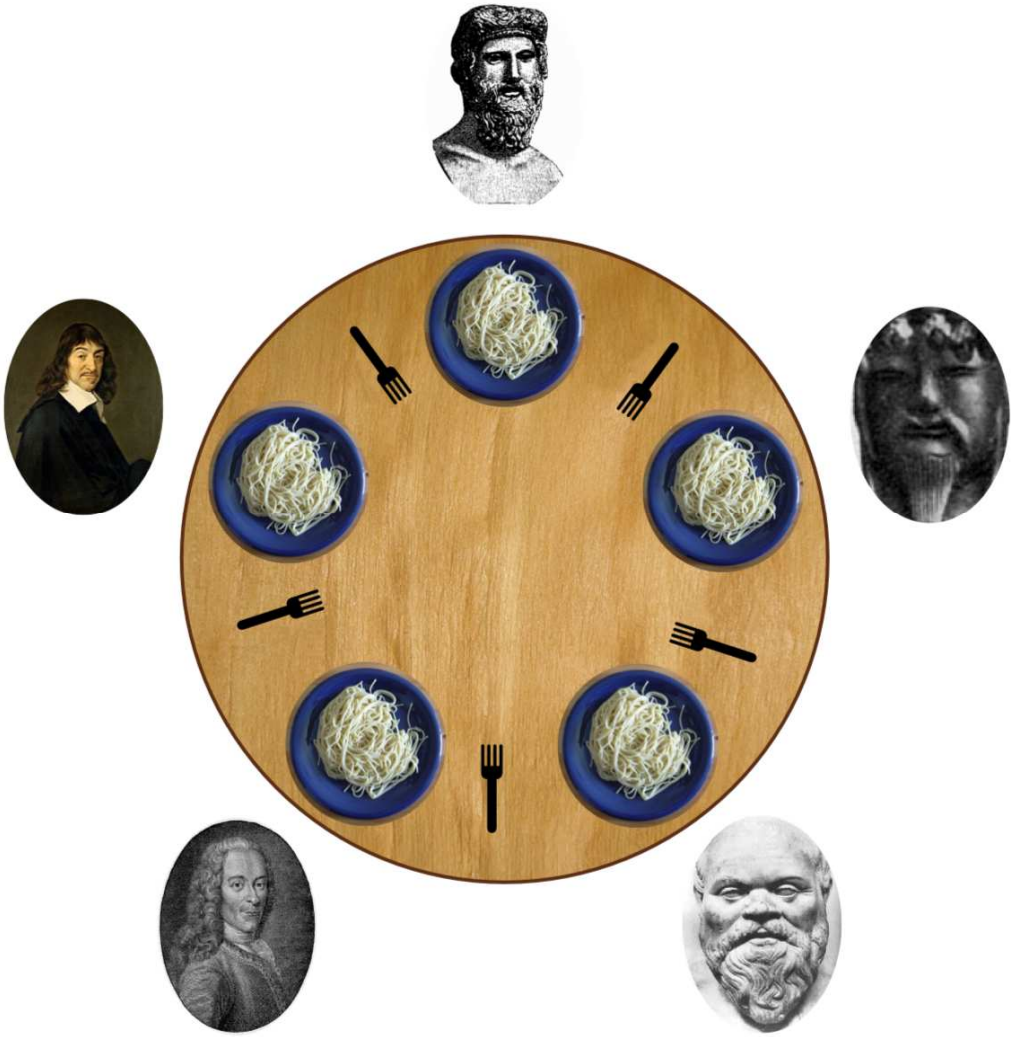
Dead locks

- Two or more threads
- Each thread holds one lock
- They wait for each other
- ???
- Avoid using more than one protected resource
- Always obtain locks in the same sequence
- Use helper lock that will protect both resources

Dead lock anti-patterns

- Timed locks
- Forced unlock
- Complex solutions

Starvation



Starvation

- Unable to obtain locks to perform work
- Non-deterministic lock allocation
- No guarantee of ever obtaining lock
- ???
- Lock smallest portion of code
- Do not perform any I/O while holding lock
- Give each protected resource it’s own lock

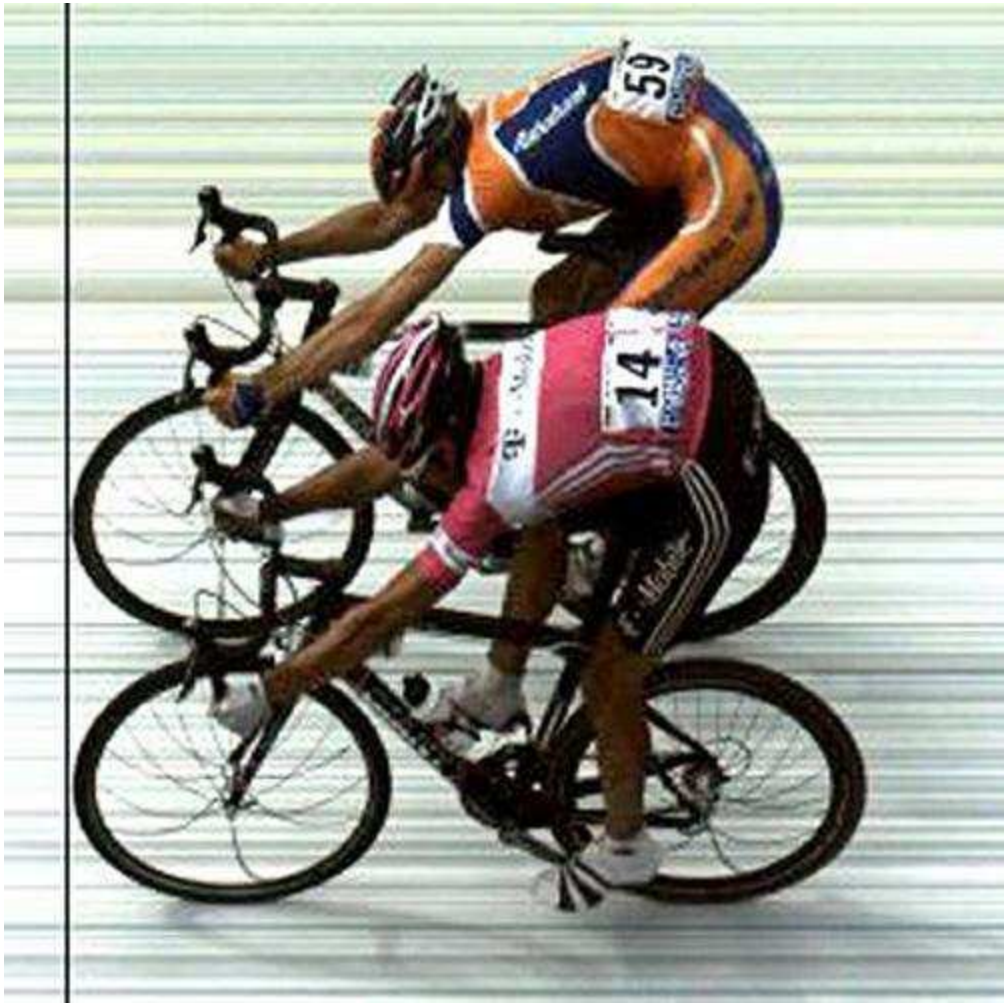
Live locks

- Same net effect as dead locks
- Symptom: high CPU usage
- Threads constantly retry
- ???
- Wait for lock, don’t use retry logic
- Avoid infinite loops (if lock is not obtained)
- Set limits (time, repetitions) and move on

Live lock anti-patterns

- Randomized wait
- Complex solutions

Race condition



Race condition

- Two threads update the same memory
- No deterministic way of predicting outcome
- Very hard to troubleshoot
- ???
- Synchronization
- Locks
- Thread-safe objects

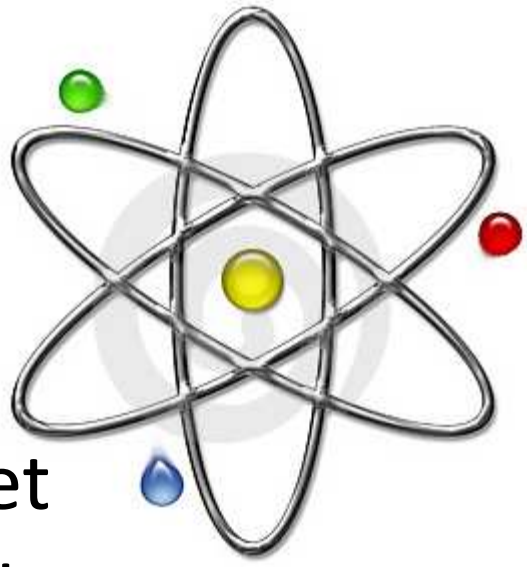
Solutions

- Immutable objects
- Atomic objects
- Concurrent objects
- Synchronization

Immutable objects

- All attributes final and set in constructor
- Cannot be modified (no setter methods)
- Example: String
- Collections.unmodifiable…
- All reachable objects need to be immutable!

Atomic objects



- All operations are atomic
- 2 in 1: addAndGet, compareAndSet
- Will not participate in more complex ops

- Examples:
- AtomicBoolean, AtomicInteger, AtomicLong
- AtomicIntegerArray, AtomicLongArray
- AtomicReference, AtomicReferenceArray

Concurrent objects

- Thread safe without using locks
- Improved performance

- Examples:
- ConcurrentHashMap, ConcurrentLinkedQueue
- ConcurrentSkipListMap, ConcurrentSkipListSet
- ArrayBlockingQueue

Synchronization

- Synchronized keyword

- LockSupport
- ReentrantLock
- ReentrantReadWriteLock
- ReentrantReadWriteLock.ReadLock
- ReentrantReadWriteLock.WriteLock

API Examples

- Executors
- CountDownLatch
- BlockingQueue
- Lock

Executors

```
class NetworkService implements Runnable {
  private final ServerSocket serverSocket;
  private final ExecutorService pool;

  public NetworkService(int port, int poolSize)
      throws IOException {
    serverSocket = new ServerSocket(port);
    pool = Executors.newFixedThreadPool(poolSize);
  }

  public void run() { // run the service
    try {
      for (;;) {
        pool.execute(new
Handler(serverSocket.accept()));
      }
    } catch (IOException ex) {
      pool.shutdown();
    }
  }
}

class Handler implements Runnable {
  private final Socket socket;
  Handler(Socket socket) { this.socket = socket; }
  public void run() {
    // read and service request on socket
  }
}
```

CountDownLatch

```
class Driver { // ...
  void main() throws InterruptedException {
    CountDownLatch startSignal = new
CountDownLatch(1);
    CountDownLatch doneSignal = new
CountDownLatch(N);

    for (int i = 0; i < N; ++i) // create and start threads
      new Thread(new Worker(startSignal,
doneSignal)).start();

    doSomethingElse();           // don't let run yet
    startSignal.countDown();     // let all threads
proceed
    doSomethingElse();
    doneSignal.await();          // wait for all to finish
  }
}
```

```
class Worker implements Runnable {
  private final CountDownLatch startSignal;
  private final CountDownLatch doneSignal;

  Worker(CountDownLatch startSignal,
CountDownLatch doneSignal) {
    this.startSignal = startSignal;
    this.doneSignal = doneSignal;
  }

  public void run() {
    try {
      startSignal.await();
      doWork();
      doneSignal.countDown();
    } catch (InterruptedException ex) {} // return;
  }
  void doWork() { ... }
}
```

BlockingQueue

	Throws exception	Special value	Blocks	Times out
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	not applicable	not applicable

BlockingDeque

	First Element (Head)		Last Element (Tail)	
	Throws exception	Special value	Throws exception	Special value
Insert	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Examine	getFirst()	peekFirst()	getLast()	peekLast()

Lock

```
Lock l = ...;
    l.lock();
    try {
        // access the resource protected by this lock
    } finally {
        l.unlock();
    }
```

Reference

- Brian Goetz
  Java Concurrency in Practice
- Java API
  http://download.oracle.com/javase/6/docs/api/

Questions