Java and Android Concurrency

Introduction

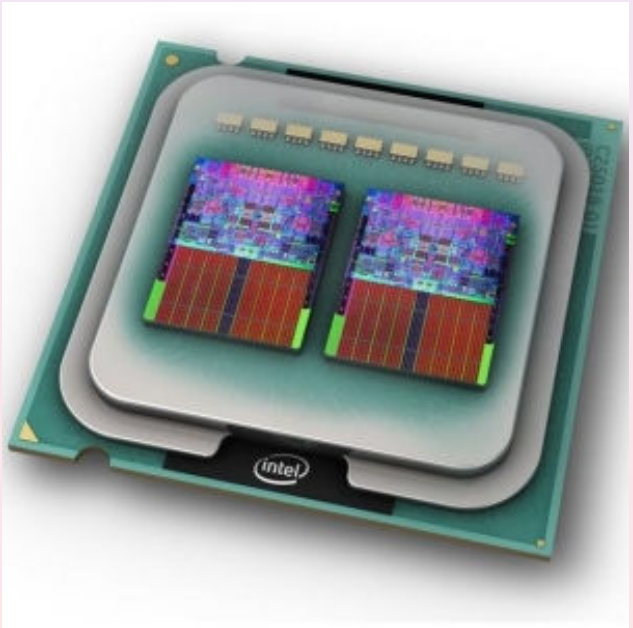
fausto.spoto@univr.it

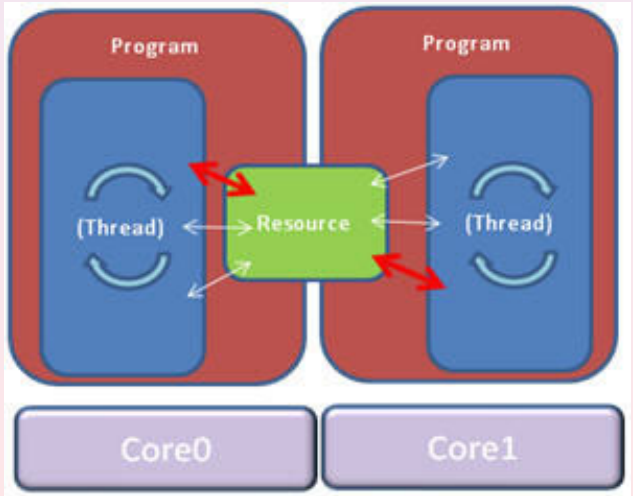
git@bitbucket.org:spoto/java-and-android-concurrency.git

git@bitbucket.org:spoto/java-and-android-concurrency-examples.git

Why Concurrency Matters

- on monocore architectures, it allows one to keep the processor busy
  - hence its use in all operating systems
  - and its presence in programming languages: C, Java etc
- on multicore architectures, it also allows one to use all computing cores to solve a single problem
  - getting important nowadays





Why Concurrency Matters More and More

Modern frameworks are inherently multithreaded

Servlet containers run each request in a separate thread. You might be writing concurrent programs without even creating a single thread in your code!

Mobile devices must be responsive

- code in the user interface thread must terminate quickly, or otherwise the user experience gets degraded: long running tasks must be delegated to non-user interface threads
- while a task is running, the user might need to start another app, make a call, surf the internet etc. The running task must continue in the background

Why Concurrency is Difficult

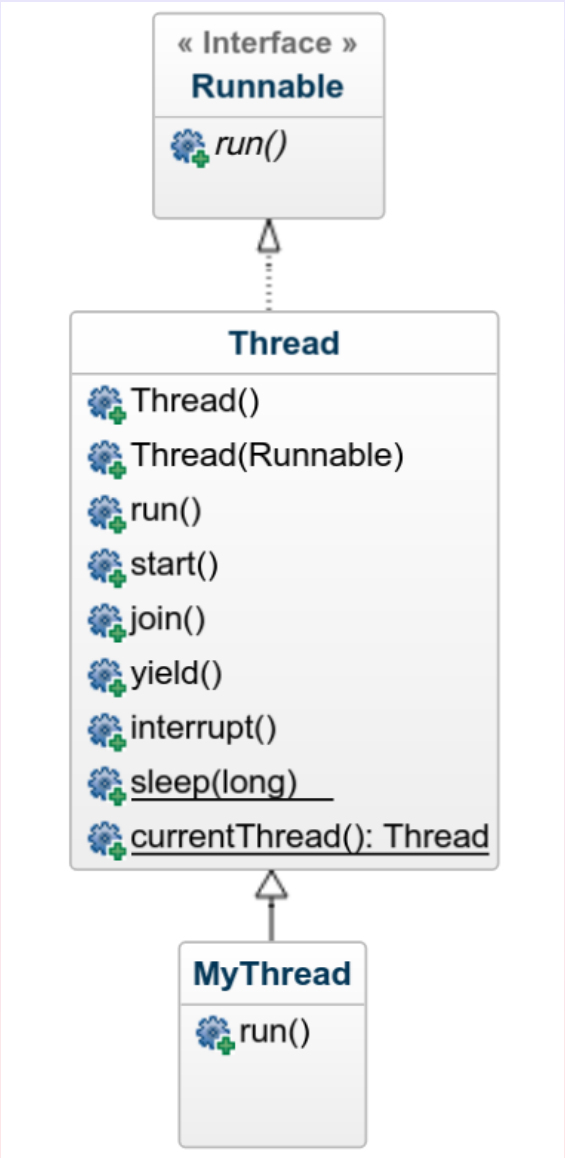
- protocols might be wrong
  - race conditions
  - deadlocks
- protocols might be inefficient
  - livelocks
  - slower than monothreaded!
- data can be shared
- data can be modified
- data can be cached on each single core
  - each core might have a different view of the memory

Programmers are not enough qualified to deal with concurrency

Concurrency in Java

- from Java 1.0:
  - *multithreading*: a thread is a process with shared heap
  - extends `java.lang.Thread` and overrides `run`
  - relatively slow to start
  - `synchronized` blocks and methods
  - each object has a lock accessible through `wait/notify/notifyAll`
  - `volatile` variables are never kept into caches
  - a formal Java memory model
- from Java 1.5:
  - improved memory model
  - `java.lang.concurrent.*` has many clever classes for concurrency: concurrent hashmaps, latches, futures, callables
  - executors recycle threads to avoid their startup cost
- from Java 1.7:
  - fork executors share tasks across threads for divide and conquer
- from Java 1.8:
  - parallel collections with lambda expression tasks

Task Executor and Task Specification



Creation of a Thread

A thread can be created by subclassing the `java.lang.Thread` class:

```
public class MyThread extends Thread {
  @Override
  public void run() { ... do something here ... }
}
...
new MyThread().start();
```

A thread can also be created by specifying the task in the constructor of a new `java.lang.Thread`:

```
public class MyRunnable implements Runnable {
  @Override
  public void run() { ... do something here ... }
}
...
new Thread(new MyRunnable()).start();
```

The Meaning of synchronized

The compiler translates

```
synchronized (expression) {
  body
}
```

into

```
final temp = expression;
get the lock of temp
body
release the lock of temp
```

temp is constant hence lock and unlock are paired

Java's intrinsic locks are reentrant

The Meaning of a synchronized Instance Method

The compiler translates

```
synchronized T foo(pars) { body }
```

into

```
T foo(pars) {
  synchronized (this) { body }
}
```

that is into

```
T foo(pars) {
  get the lock of this
  body
  release the lock of this
}
```

"this" is constant in Java hence lock and unlock are paired

The Meaning of a synchronized Static Method of Class C

The compiler translates

```
synchronized static T foo(pars) { body }
```

into

```
static T foo(pars) {
  synchronized (C.class) { body }
}
```

that is into

```
T foo(pars) {
  get the lock of C.class
  body
  release the lock of C.class
}
```

C.class is constant in Java hence lock and unlock are paired

The Meaning of `wait/notify/notifyAll` (Historical!)

`expression.wait()`

Waits until somebody will notify on the value of `expression`, temporarily releasing any lock held by the current thread

`expression.notify()`

Wakes up a thread waiting on the value of `expression`, if any. If more threads are waiting, one of them is non-deterministically chosen and awakened

`expression.notifyAll()`

Wakes up all threads waiting on the value of `expression`, if any. The awakened threads must recheck the condition they were waiting for

These calls must occur only when the thread has already synchronized on the value of `expression`

Programmers Do it Wrong

"race conditions only occur in books"

`RaceCondition.java` (when racers collide)

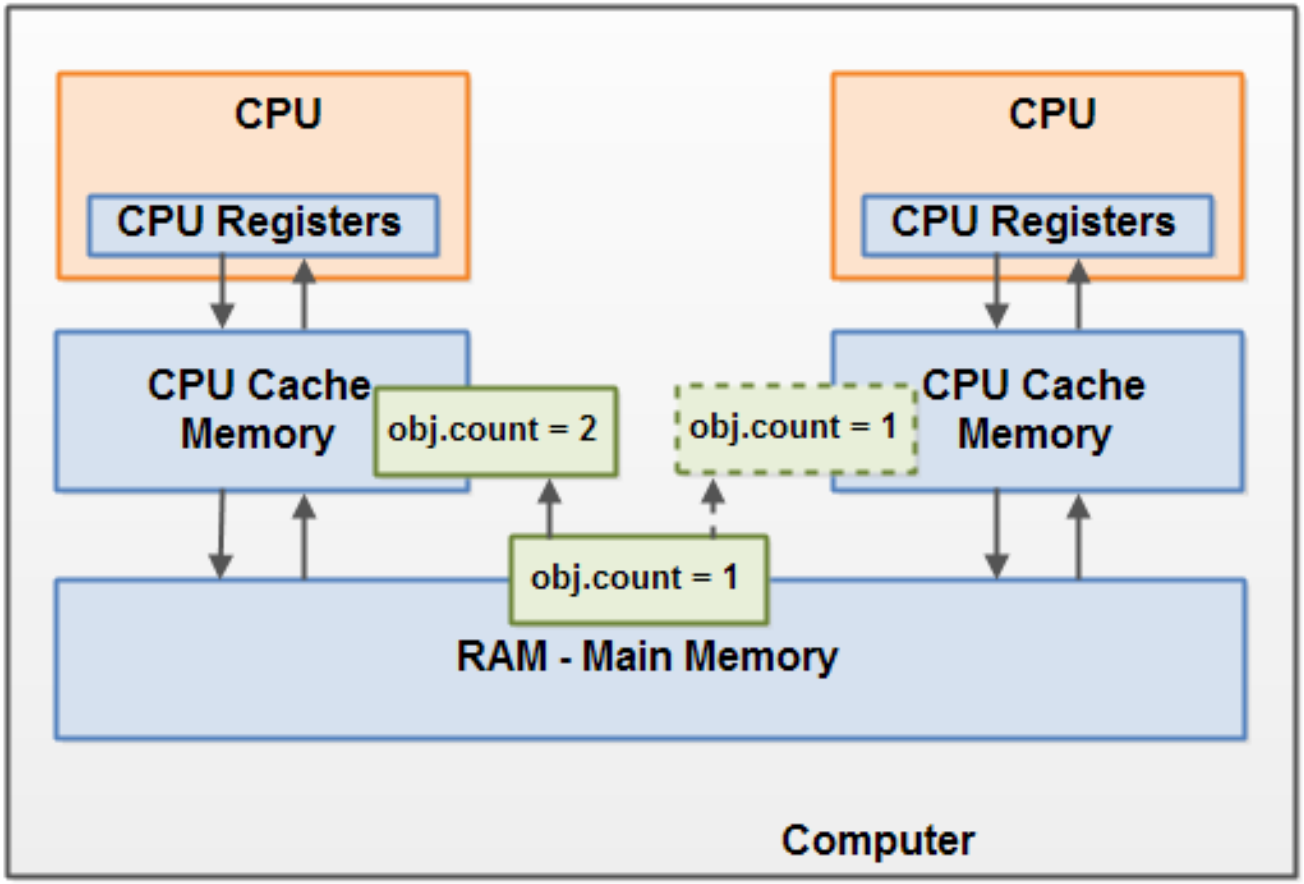
"parallelism increases speed"

`Two2One.java` (when one is better than two)

"deadlocks do not exist in practice"

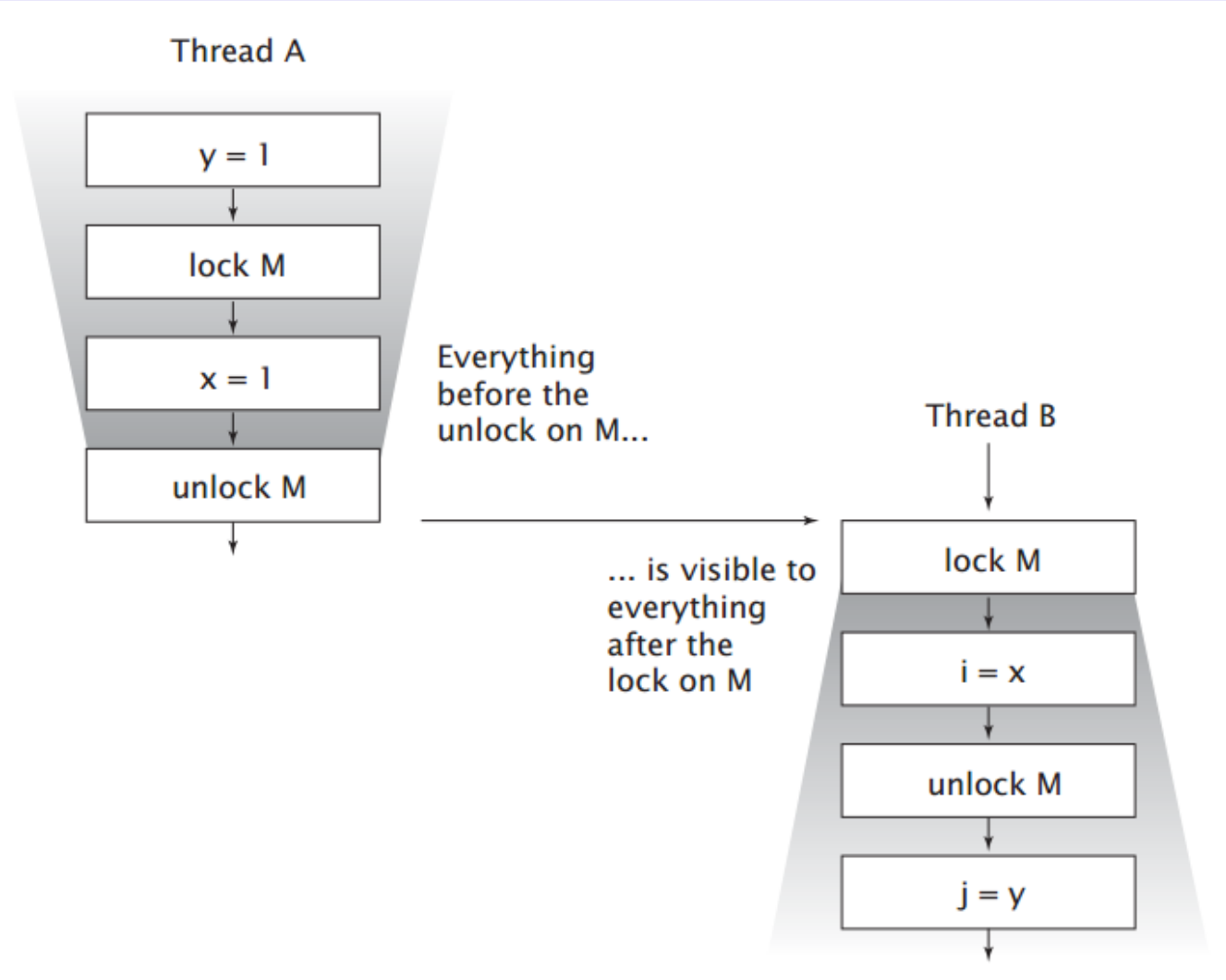
`Philosophers.java` (when philosophers hang)

The Visibility Problem



The meaning of `volatile`

The Java Memory Model



The *happens-before* relation holds in many other situations as well

The Thread-Safeness Problem

- thread-safe libraries are being developed nowadays
  - their classes should be thread-safe
- but what does thread-safeness mean exactly?
  - it can be used in a multithreaded way?
  - any multithreaded execution can be rephrased as a sequential execution?
- verifying thread-safeness is still impossible today
  - immutable classes are thread-safe!
  - use thread-safe classes from the Java standard library
  - for more complex cases, there are heuristics

The GUI Problem

- GUI toolkits are normally monothreaded (Swing, Android, . . .)
- operations on widgets must only be performed from the graphical thread
- long-running operations block the graphical thread
- they must be offloaded to worker threads
- when a worker thread terminates
  - it must be able to notify the user through the graphical thread
  - views might have disappeared in the meanwhile (Android)

The @GuardedBy/@Holding Annotations

@GuardedBy

States that a field or parameter is only accessed by holding a lock

- introduced by Brian Goetz
- used for the NASA PathFinder project
- partially checked and inferred by some analysis tools

@Holding

States that a method is only called by holding a lock

- partially checked and inferred by some analysis tools

Possibilities for @GuardedBy/@Holding

@GuardedBy/@Holding("this")

the lock on the receiver of a non-static field or method must be held

@GuardedBy("itself")

the lock on the same parameter or field must be held

@GuardedBy/@Holding("field-name")

the lock on the named field of the receiver of a non-static field or method must be held

@GuardedBy/@Holding("Class.field-name")

the lock on the named static field of the named class must be held

Possibilities for `@GuardedBy/@Holding`

@GuardedBy/@Holding("Class.class")

the lock of the unique class object representing the class named `Class` must be held

@GuardedBy/@Holding("foo()")

the lock on the return value of the named instance method called on the receiver of a non-static field or method must be held. Method `foo` must return a reference type

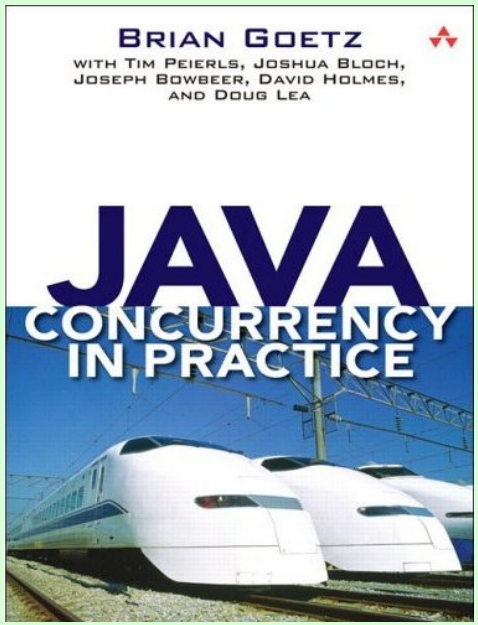
@GuardedBy/@Holding("Class.foo()")

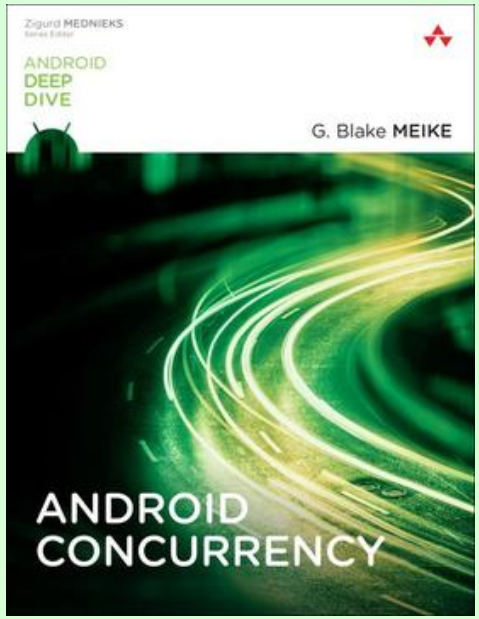
the lock on the return value of the named static method of the named class must be held. Method `foo` must return a reference type

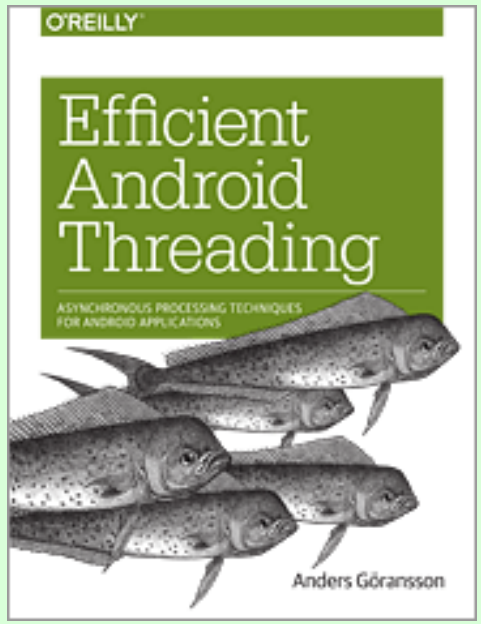
Tools for Concurrent Software Development

- the standard `-Xprof` Java profiler is a basic tool for simple profiling: identifies blocking time and deadlocks
- the YourKit Java profiler provides detailed information on block time and monitor usage and identifies deadlocks
- static checkers such as Julia and FindBugs

References


BRIAN GOETZ
WITH TIM PEIERLS, JOSHUA BLOCH,
JOSEPH BOWBEER, DAVID HOLMES,
AND DOUG LEA
JAVA
CONCURRENCY
IN PRACTICE


Zigurd MEDNIEKS
ANDROID
DEEP
DIVE
G. Blake MEIKE
ANDROID
CONCURRENCY


O'REILLY
Efficient
Android
Threading
ASYNCHRONOUS PROCESSING TECHNIQUES
FOR ANDROID APPLICATIONS
Anders Göransson