Thread Safety

Pieter van den Hombergh

Fontys Hogeschool voor Techniek en Logistiek

March 16, 2015

Overview of Part I: Fundamentals

- Introduction to threads
- Thread safety
- Sharing objects
- Composing objects
- Building blocks
- Summary
- Links
  - http://java.sun.com/docs/books/tutorial/ **java/generics/index.html**
  - http://java.sun.com/docs/books/tutorial/ **essential/concurrency/index.html**

Introduction

- History
- Benefits
- Risks: safety, liveness, performance
  - safety: Unpredictable ordering of operations in multiple threads (in absence of synchronization).
  - liveness: A concurrent application's ability to execute in a timely manner.
  - performance: Includes: poor service time, responsiveness, throughput, resource consumption, scalability.
- Threads are everywhere
  - Your java program: a main thread
  - Usefull classes/frameworks made available for you: Timer, Servlets and JavaServer Pages (JSP), Remote Method Invocation (RMI), Swing and AWT (GUI apps.).

Thread safety

If multiple threads access the same mutable state variable without appropriate synchronization, *your program is broken*.

There are three ways to fix it:

- *Don't share* the state variable across threads
- Make the state variable *immutable*; or
- Use *synchronization* whenever accessing the state variable

A definition:

Thread safety A class is *thread-safe* if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by running environment, and with no additional synchronization or other coordination on the part of the calling code.

An example from SUN: SimpleThreads.java

```
private static class MessageLoop implements Runnable {
    public void run() {
        String importantInfo[] = {"Mares eat oats",
                                  "Does eat oats", "Little lambs eat ivy",
                                  "A kid will eat ivy too"};
        try {
            for (int i = 0; i < importantInfo.length; i++) {
                //Pause for 4 seconds
                Thread.sleep(4000);
                //Print a message
                threadMessage(importantInfo[i]);
            }
        } catch (InterruptedException e) {
            threadMessage("I wasn't done!");
        }
    }
}
```


An example from SUN: (2)

```
public static void main(String args[])
    throws InterruptedException {
    //Delay, in millis before we interrupt MessageLoop
    long patience = 1000*60*60; //thread (default 1 hour).
    //If command line argument present, prints patience.
    if (args.length > 0) {
        try {
            patience = Long.parseLong(args[0]) * 1000;
        } catch (NumberFormatException e) {
            System.err.println("Argument must be an integer.");
            System.exit(1);
        }
    }
    threadMessage("Starting MessageLoop thread");
    long startTime = System.currentTimeMillis();
    Thread t = new Thread(new MessageLoop());
    t.start();
    threadMessage("Waiting for MessageLoop thread to finish");
```


An example from SUN:(3)

```
    while (t.isAlive()) {
        threadMessage("Still waiting...");
        //Wait maximum of 1 second for MessageLoop thread
        //to finish.
        t.join(1000);
        if (((System.currentTimeMillis()-startTime)>patience)
            && t.isAlive()) {
            threadMessage("Tired of waiting!");
            t.interrupt();
            //Shouldn't be long now -- wait indefinitely
            t.join();
        }
    }
    threadMessage("Finally!");
}
```

Object orientation

- When designing thread-safe classes, good object-oriented techniques - **encapsulation, immutability, and clear specification of invariants** - are your best friends.
- Thread-safe classes encapsulate any needed synchronization so that clients need not provide their own.
- Stateless objects are always thread-safe. This includes
  - memberless classes
  - classes with only final members like `java.lang.Integer` and `java.lang.Long`

Atomicity

- Even an increment operation: `count++;` is not **atomic** although it might look as if it is an indivisible operation!
- Compound actions: operations **A** and **B** are **atomic** with respect to each other if, from the perspective of a thread excuting **A**, when another thread executes **B**, either all of **B** has executed or none has. An **atomic operation** is one that is atomic with respect to all operations, including itself, that operate on the same state.
- Where practical, use existing thread-safe objects, like `java.util.concurrent.atomic.AtomicLong`, to manage your class's state. It is simpler to reason about the possible states and state transitions for existing thread-safe objects than it is for arbitrary state variables, and this makes it easier to maintain and verify thread safety.

Race conditions

- A race condition occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime; in other words: when getting the right answer depends on lucky timing.
- Most encountered race condition: *check-then-act*: you observe something to be true, (file X doesn't exist) and then take action based on that observation (create X); but in fact the observation could have become invalid between the time you observed it and the time you acted on it (someone else created X in the meantime), causing a problem (unexpected exception, overwritten data, file corruption).

Listing 2.3. Race condition in lazy initialization.

```java
@NotThreadSafe
public class LazyInitRace {
    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
        if (instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}

class ExpensiveObject { }
```

Locking

- When multiple variables determine the state of an object: just make all variables threadsafe???
- Making all attribute objects threadsafe is not sufficient!
- "To preserve state consistency (or invariants), update related state variables in a single atomic operation."

Intrinsic Locks

- **Every object** in java can act as a lock for purpose of synchronization!
- These locks are called *intrinsic locks* or *monitor locks*.
- Intrinsic locks in java act as *mutexes*: `mutual exclusion locks`, which means that at most one thread may own the lock and can proceed.

```
synchronized (lock) {
   // Access or modify shared state
   // guarded by lock
}
```


Reentrancy

- When a thread requests a lock that is already held by another thread, the requesting thread blocks.
- The request will however succeed if that **same** thread holds the lock: locks are acquired on a per-thread basis.
- Reentrancy is implemented by associating an acquisition count and an owning thread with each lock . The lock is released when the count reaches zero, obviously after an exit of a synchronized block.
- Reentrancy simplifies development of object-oriented concurrent code and can avoid deadlock in situations such as in the code on the next slide.
- Note that this also works for synchronisation used in both super and subclass. In the **object**, the lock is the **same**.

Code that would deadlock

Code that would deadlock if intrinsic locks were not reentrant.

```
public class Widget {
    public synchronized void doSomething() {
        ...
    }
}

public class LoggingWidget extends Widget {
    public synchronized void doSomething() {
        System.out.println(toString() +
                   ": calling doSomething");
        super.doSomething();
    }
}
```

Guarding states with locks

- For each mutable state variable that may be accessed by more than one thread, **all** accesses to that variable must be performed with the **same** lock held. In this case, we say that the variable is *guarded by* that lock.
- You as programmer should construct the **locking protocols** or **synchronization policies** that let you access shared states safely.
- Every shared mutable variable should be guarded by exactly one lock. Make it clear to maintainers which lock that is. (Annotation: **@GuardedBy(..)**)

Guarding states with locks II

- For every invariant that involves more than one variable, *all* the variables involved in that invariant must be guarded by the *same* lock.
- Declaring every method synchronized will not prevent race conditions! See the following code: (Vector's methods are all synchronized)

```
if (!vector.contains(element))
    vector.add(element);
```

Performance

- Guarding each state variable with the object's intrinsic lock may lead to bad performance. If for example only one client thread may execute the service method in a servlet application. See the `SynchronizedFactorizer` and `CachedFactorizer`(book: Listings 2.6 and 2.8).
- There is frequently a tension between simplicity and performance. When implementing a synchronization policy, resist the temptation to prematurely sacrifice simplicity (potentially compromising safety) for the sake of performance.
- Avoid holding locks during lengthy computations or operations are at risk of not completing quickly such as network or console I/O.

Summary

Immutability: All concurrency issues boil down to coordinating access to mutable state. The less mutable state, the easier it is to ensure thread safety.

Final: Make fields final unless they need to be mutable.

Thread safe: Immutable objects are automatically thread safe.

Encapsulation: Immutable objects simplify concurrent programming tremendously. They are simpler and safer, and can be shared freely without locking or defensive copying.

Lock: Guard each shared mutable variable with a lock.

Summary (2)

Thread Safety
HOM
Introduction
Introduction
Thread Safety
Thread Safety
Locking
Liveliness and Performance
Summary
lab

Invariants: Guard all variables in an invariant with the same lock.

Compound actions: Hold locks for the duration of compound actions.

Multiple threads: A program that accesses a mutable variable from multiple threads without synchronization is a broken program.

I don't need ... Don't rely on clever reasoning about why you don't need to synchronize.

Design process: Include thread safety in the design process - or explicitly document that your class is not thread safe.

Document: Document your synchronization policy.

Restaurant task, explain please

Thread Safety
HOM
Introduction
Introduction
Thread Safety
Thread Safety
Locking
Liveliness and Performance
Summary
lab

Just to remember the speaker.

- Quiz: Why is the original restaurant (Donald only) threadsafe?
- In this version: Use simple synchronisation primitives to create a threadsafe application as in “roll you own”. Do not peek. NO frameworks allowed, no gui needed.
- Document your design and code (javadoc!!!).
- Hand in on paper.
  - Make sure the frontpage contains all author names.
  - One hands the full document, the other only (a copy of) the frontpage (as a reference to the original, complete doc).
  - You may choose to work allone if you think you cannot find or are dissatisfied with your partner. In those cases, create a personal repository.