Threads Synchronization

Radu Nicolescu
Department of Computer Science
University of Auckland

15 May 2013

Consider again this counter class

```
class Counter {
    private int c = 0;
    public void increment() {
        int r = c;
        r += 1;
        c = r;
    }
    public void decrement() {
        int r = c;
        r -= 1;
        c = r;
    }
    // continued on next slide
```

Counter class

```
class Counter {
    // continued from previous slide
    public int value() {
        return c;
    }
}
```

What is wrong with this class?

- when used from a single thread
- when used from several threads concurrently

A counter object and a thread class that uses it

```
Counter Ctr = new Counter();
class MyThread extends Thread {
    public void run() {
        try {
            for (int i=0; i<100; i++) {
                Ctr.increment();
                Ctr.decrement();
            }
        } catch (Exception ex) {}
    }
}
```

If all threads terminate properly, the final counter value should be 0, isn't it? FALSE!

The main program of the counter demo

```
private void main() throws Exception {
    Thread[] T = new Thread[10];

    for (int k=0; k < T.length; k++)
        T[k] = new MyThread();

    for (int k=0; k < T.length; k++)
        T[k].start();

    for (int k=0; k < T.length; k++)
        T[k].join();
}
```

Here all threads do terminate properly, so the final counter value must be 0, isn't it? FALSE!!

What is the problem and what it its solution?

- Because of thread interleaving, or of multi-core execution, the result can well be different of 0!
- Demo program: NonSynchronized
- Solutions: run increment and decrement as a critical section

Critical sections and mutual exclusion

Definition (Critical section)

A critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread

Attention: A critical region need not be contiguous, it may consists of fragments, which may span over several methods or even several objects

Mutual exclusion

Critical sections are typically enforced by mutual exclusion algorithms

Critical sections in Java

A few simple critical sections in Java:

- Synchronized methods (demo program: Synchronized)
- Synchronized statements (demo program: Synchronized2)
- Reentrant locks (demo program: Locked)

Attention: critical section reduce the potential for parallelism in a program, affecting its performance, or even creating deadlocks

Tradeoff: performance vs. size of critical section

There are quite a few more sophisticated ways to enforce critical sections, but we won't study them in this lecture

Synchronized methods

To make a method synchronized, add the **synchronized** keyword to its declaration:

```
class Counter {
    private int c = 0;
    public synchronized void increment() {
        ...
    }
    public synchronized void decrement() {
        ...
    }
    public synchronized int value() {
        ...
    }
}
```

Synchronized methods

The **synchronized** keyword prevents thread interference, between threads accessing the same object

For a given object, the union of its **synchronized** methods form together a protected critical section

Besides its other functions, any object has the builtin capacity to work like a lock (or monitor), allowing only one thread at a time in a critical region

For its own synchronized methods, the object is known as an intrinsic lock

Synchronized statements

To use **synchronized** statements, you must explicitly indicate the object offering the lock

- the *lock* object applies to blocks of statements, not necessarily to whole methods
- allowing finer granularity (thus more parallelism and better performance)
- the *lock* object could be different from the current object
- an explicit *lock* can enforce mutual exclusion over a wider critical section, consisting of methods from several objects
- allowing more flexibility

Synchronized statements

In this simple scenario, the *explicit lock* is **this** (the current object) and it applies to whole methods body – making it equivalent to the scenario using synchronized statements

```
class Counter {
    private int c = 0;
    public void increment() {
        synchronized(this) {...}
    }
    public void decrement() {
        synchronized(this) {...}
    }
    public int value() {
        synchronized(this) {...}
    }
}
```

Reentrant locks

Reentrant locks allow more granularity and even better performance, but require more coding discipline

The critical section can have a dynamic shape, *not* restricted to structured blocks

You create a lock instance by:

```
Lock lck = new ReentrantLock();
```

Setting the lock indicates the start of the critical section:

```
lck.lock();
```

Releasing the lock indicates the end of the critical section:

```
lck.unlock();
```

Reentrant locks

The following scenario is equivalent to the previous ones

```
Lock lck = new ReentrantLock();
...
public void increment() throws Exception {
    lck.lock();
    try {
        int r = c;
        r += 1;
        c = r;
    } finally {
        lck.unlock();   // !!!
    }
}
```

Reentrant locks

You must release the lock, else the critical section doesn't end

The recommended way is to *unlock* locks in **finally** clauses. Why?

```
        lck.lock();
        try {
            ...
        } finally {
            lck.unlock();   // !!!
        }
    }
```

Synchronization quiz

The following questions consider the code snippet presented in the next slide:

- How many critical sections do you see?
- What's the extent of each critical section?
- Can you protect the same critical regions using synchronized methods (without making drastic changes to the code structure)?
- Can you protect the same critical regions using reentrant locks?

Synchronization quiz

```
class Top {
    Object A = new Object();
    Object B = new Object();
    Object C = B;
    class Alpha {
        public void X() {
            synchronized(A) {...}
            synchronized(B) {...}
        }
    }
    class Beta {
        public void Y() {
            synchronized(B) {...}
            synchronized(C) {...}
        }
    }
}
```

Pessimistic and optimistic concurrency management

- Pessimistic strategies (locks): synchronized methods, synchronized blocks, reentrant locks
  - Take all necessary measure to prevent any possible thread interference
  - This is a costly approach, in terms of resources and performance, with significant overheads
- In many practical cases, locks are rarely contended, if ever
- Optimistic strategies: develop better designs, which maximize the performance and minimize the risks
  - Detect when interferences occur and only then take corrective actions (e.g., repeat something until it succeeds)

More about locks

We first study the **pessimistic** strategies

What are the similarities and the differences between reentrant locks and synchronized methods or statements?

Similarities: they enforce exclusive access to a shared resource: *only one thread at a time* can acquire the lock (required to access the shared resource).

Differences: do reentrant locks offer additional functionality, not available for synchronized methods and blocks?

When to use one and when the other?

The Lock interface

We look now in more detail at the specific and additional functionality offered by the Lock interface and ReentrantLock class:

- interface Lock
- class ReentrantLock implements Lock

What "reentrant" means?

Consider the following scenario:

```
public class ReentrantSample{
  public synchronized outer(){
    inner();
  }
  public synchronized inner(){
    //do something
  }
}
```

What happens if a thread acquires the lock on a ReentrantSample object and enters outer()?

Will it be next allowed to enter inner(), which requires to re-acquire the same lock?

Yes, synchronized methods and blocks are reentrant; and so are ReentrantLocks (which explains their name)

Static vs dynamic locking policies

Synchronized methods or statements force all lock acquisition and release to occur in a block-structured way (which is also indicated by the lexical structure). For example:

acquire A, acquire B, release B, release A

```
synchronized (objA) {
    ...
    synchronized (objB) {
        ...
    }
    ...
}
```

Static vs dynamic locking policies

RentrantLock's work in a dynamic way. For example:

acquire A, acquire B, release B, release A

```
lockA.lock(); ...;  lockB.lock(); ...;
lockB.unlock(); ... ; lockA.unlock(); ...;
```

or

acquire A, acquire B, release A, release B

```
lockA.lock(); ...;  lockB.lock(); ...;
lockA.unlock(); ...;  lockB.unlock(); ...;
```

More flexible, but also more risky (if not properly implemented)

Basic Lock interface

Lock implementations provide additional functionality (API) over the use of synchronized methods and statements:

- void lock() : Acquires the lock
- void lockInterruptibly() throws InterruptedException : Acquires the lock, unless the current thread is *interrupted* (otherwise there is no easy way to interrupt a thread waiting to acquire the lock)
- boolean tryLock() : Non-blocking attempt, acquires the lock only if it is free at the time of invocation
- boolean tryLock(long time, TimeUnit unit) throws InterruptedException : Acquires the lock with *timeout*, i.e. if it is free within the given waiting time, and the current thread has not been *interrupted*
- void unlock() : Releases the lock

Basic Lock interface

Details of the timeout facility:

```
Lock lock = ...;

if ( lock.tryLock(50L, TimeUnit.SECONDS) ) ...

if ( lock.tryLock(50L, TimeUnit.HOURS) ) ...
```

Slightly better than having to convert everything into milliseconds...

Additional ReentrantLock functionality (API)

- int getHoldCount() : Queries the number of holds on this lock by the current thread
- protected Thread getOwner() : Returns the thread that currently owns this lock, or null if not owned
- protected Collection<Thread> getQueuedThreads() : Returns a collection containing threads that may be waiting to acquire this lock
- int getQueueLength() : Returns an estimate of the number of threads waiting to acquire this lock

Where would you use these additional methods?

Mainly for debugging, to check your logic

Additional ReentrantLock functionality (API)

getHoldCount() could be zero, one, or also greater than one (because of reentrant invocations)

Example, ensure that you don't reenter again a critical section:

```
class X {
    ReentrantLock lock = new ReentrantLock();
    public void m() {
      assert lock.getHoldCount() == 0;
      lock.lock();
      try {
        ...
      } finally {
        lock.unlock();
      }
    }
 }
```

Atomic variables

For simple operations, use Atomic∗ classes, such as: AtomicInteger, AtomicLong, AtomicReference, AtomicIntegerArray, etc.

For example, AtomicInteger provides *atomic* methods for:

- addAndGet(): add and return the updated value
- getAndAdd(): add and return the original value
- incrementAndGet(): increment and return the updated value
- getAndIncrement(): increment and return the original value

These operation use much more efficient hardware locks, not software locks. More about this later, when we talk about optimistic approaches.

Thread local storage

Generic class ThreadLocal<T> provides *thread local storage* (TLS) for variables, which are

1. declared only once in the source code, but
2. automatically multiplied to independent copies, one for each thread, separately initialised.

Thus, a TLS variable is a thread global variable which is unique each thread.

In Java, class Random is thread-safe, so you could directly use it in your assignment, even if it gets called by different threads.

However, you can avoid contentions and speed a bit the evaluation by using a thread local random class; there is a predefined one: ThreadLocalRandom from java.util.concurrent.

Thread local storage

```
class SimpleThread extends Thread {
  private static long tid() {
    return Thread.currentThread().getId(); }

  // "magically", one tli instance for each thread
  private static final ThreadLocal<Long>
    tli = new ThreadLocal<Long>() {... "magic" ...}

  // get() returns the thread local instance
  @Override public run() {
    System.out.format("%d %d %n",
      tid(), tli.get().intValue());
  }
}
```

Thread local storage

```
...
// "magically", one tli instance for each thread
private static final ThreadLocal<Long>
    tli = new ThreadLocal<Long>() {
      @Override // "magically", called exactly once by each thread
      protected Long initialValue() {
        return tid() + 1000;
      }
    };
...
```

Note that ThreadLocalRandom is already properly initialised, so you just use it (see API reference).