**Software Testing, Quality Assurance and Maintenance** Winter 2015

# Lecture 10 — January 26, 2015

*Patrick Lam* *version 0*

We are going to completely switch gears and talk about testing concurrent programs next. For those of you in 3A, SE 350 is the first real exposure to these concepts; you'll see them more in CS 343. ECE 459 also teaches you how to leverage parallelism.

**Context: Multicores are everywhere today!** For the past 10 years, chips have not been getting more GHz. We still have more transistors though. Hardware manufacturers have been sharing this bounty with us, through the magic of multicore processors!


If you want performance today, then you need parallelism.

The dark side is that concurrency bugs will bite you. 

> "More often than not, printing a page on my dual-G5 crashes the application. The funny thing is, printing almost never crashes on my (single-core) G4 PowerBook."
>
> `http://archive.oreilly.com/pub/post/dreaded_concurrency.html`

The most famous kind of concurrency bug is the race condition. Let's look at this code.

```
1  #include <iostream>
2  #include <thread>
3
4  int counter = 0;
5
6  void func() {
7      int tmp;
8      tmp = counter;
9      tmp++;
10     counter = tmp;
11 }
12
13 int main() {
14     std::thread t1(func);
15     std::thread t2(func);
16     t1.join();
17     t2.join();
18     std::cout << counter;
19
20     return 0;
21 }
```

When we run it:

```
plam@polya /tmp> ./a.out
2
plam@polya /tmp> ./a.out
2
plam@polya /tmp> ./a.out
1
plam@polya /tmp> ./a.out
1
plam@polya /tmp> ./a.out
2
plam@polya /tmp> ./a.out
2
```

Yes, that's a race condition.

- A race occurs when you have two concurrent accesses to the same memory location, at least one of which is a **write**.

Race conditions arise between variables which are shared between threads. Note that when there's a race, the final state may not be the same as running one access to completion and then the other.

**Tools to the rescue.** While races may be entertaining, race conditions are never good. You have several dynamic analysis tools at your disposal to eradicate races, including:

- Helgrind (part of Valgrind)
- lockdep (Linux kernel)
- Thread Analyzer (Oracle Solaris Studio)
- Thread Analyzer (Coverity)
- Intel Inspector XE 2011 (formerly Intel Thread Checker)

We can run the race condition shown above under Helgrind:

```
plam@polya /tmp> g++ -std=c++11 race.C -g -pthread -o race
plam@polya /tmp> valgrind --tool=helgrind ./race
[...]
==6486== Possible data race during read of size 4 at 0x603E1C by thread #3
==6486== Locks held: none
==6486==    at 0x400EA1: func() (race.C:8)
==6486==    by 0x402254: void std::_Bind_simple<void (*())()>::_M_invoke<>(std::_Index_tuple<>) (functional:1732)
==6486==    by 0x4021AE: std::_Bind_simple<void (*())()>::operator()() (functional:1720)
==6486==    by 0x402147: std::thread::_Impl<std::_Bind_simple<void (*())()> >::_M_run() (thread:115)
==6486==    by 0x4EF196F: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.20)
==6486==    by 0x4C2FO56: mythread_wrapper (hg_intercepts.c:234)
==6486==    by 0x56650A3: start_thread (pthread_create.c:309)
==6486==    by 0x595FCCC: clone (clone.S:111)
==6486==
==6486== This conflicts with a previous write of size 4 by thread #2
==6486== Locks held: none
==6486==    at 0x400EB1: func() (race.C:10)
==6486==    by 0x402254: void std::_Bind_simple<void (*())()>::_M_invoke<>(std::_Index_tuple<>) (functional:1732)
==6486==    by 0x4021AE: std::_Bind_simple<void (*())()>::operator()() (functional:1720)
==6486==    by 0x402147: std::thread::_Impl<std::_Bind_simple<void (*())()> >::_M_run() (thread:115)
==6486==    by 0x4EF196F: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.20)
==6486==    by 0x4C2FO56: mythread_wrapper (hg_intercepts.c:234)
==6486==    by 0x56650A3: start_thread (pthread_create.c:309)
==6486==    by 0x595FCCC: clone (clone.S:111)
==6486==  Address 0x603e1c is 0 bytes inside data symbol "counter"
```

**Not enough.** OK, great. Now you've eliminated all races (as required by specification). Of course, there are still lots of bugs that your program might contain. Some of them are even concurrency bugs. Here, there are no longer any contended accesses, but we cheated by caching the value. Atomic operations would be safe.

```
1 #include <iostream>
2 #include <thread>
3 #include <mutex>
4
5 int counter = 0;
6 std::mutex m;
7
8 void func() {
9     int tmp;
```

```
10
11      m.lock();
12      tmp = counter;
13      m.unlock();
14      tmp++;
15      m.lock();
16      counter = tmp;
17      m.unlock();
18  }
```

We can test our code for additional concurrency bugs:

- run the code multiple times
- add noise (sleep, more system load, etc)
- Helgrind and friends
- force scheduling (e.g. Java PathFinder)
- static approaches: lock-set, happens-before, state-of-the-art techniques

**Reentrant/recursive Locks** What happens if you have two requests for a POSIX/C++11 lock?

If the requests are in different threads, the second thread waits for the first thread to unlock. But, if the requests are in the same thread, that thread waits for itself to unlock. . . forever!

To avoid this unhappy situation, we can use *recursive* locks. Each lock knows how many times its owner has locked it. The owner must then unlock the same number of times to liberate.

Java locks work this way, e.g.

```
1  class SynchronizedIsRecursive {
2    int x;
3
4    synchronized void f() {
5      x--;
6      g(); // does not hang!
7    }
8
9    synchronized void g() {
10     x++;
11   }
12 }
```

Although every Java object is a lock, and we can `synchronized()` over every lock, `ReentrantLocks` are more special:

- we can explicitly `lock()` & `unlock()` them,

- (or even `trylock()`)!

However, inexpertly-written Java programs might hog the lock. To avoid that, use a `try/finally` construct:

```
Lock lock = new ReentrantLock();
lock.lock();
try {
  // you got the lock! workworkwork
} finally {
  // might have thrown an exception
  lock.unlock();
}
```

## Tools for detecting lock usage issues

The reference for the next example is Engler et al [ECCH00]. This example falls short of excellence:

```
/* 2.4.0:drivers/sound/cmpci.c:cm_midi_release: */
lock_kernel(); // [PL: GRAB THE LOCK]
if (file->f_mode & FMODE_WRITE) {
  add_wait_queue(&s->midi.owait, &wait);
  ...
  if (file->f_flags & O_NONBLOCK) {
    remove_wait_queue(&s->midi.owait, &wait);
    set_current_state(TASK_RUNNING);
    return -EBUSY; // [PL: OH NOES!!1]
  }
  ...
}
unlock_kernel();
```

The problem: lock() and unlock() calls must be paired! They are on the "happy path", but not on the `-EBUSY` path. [ECCH00] describes a tool that allows developers to describe calls that must be paired.

Another example:

```
foo(p, ...)
bar(p, ...);
```

```
foo(p, ...)
bar(p, ...);
```

```
foo(p, ...)
// ERROR: foo, no bar!
```

Our tool might then give the following results: 23 errors, 11 false positives. A false positive is something where the tool reports an error, but for instance, the error is in an infeasible path.

The next challenge is: how do we find such rules? In particular, we want to find rules of the form "A() must be followed by B()", or"`a(); ... b();`", which denotes a MAY-belief that a() follows b(). iComment by Lin Tan et al propose mining comments to find them [TYKZ07].

Here is some OpenSolaris code that demonstrates expectations. Also different locking primitives.

```
1  /* opensolaris/common/os/taskq.c: */
2  /* Assumes: tq->tq_lock is held. */
3            /* ↪ consistent ✓ */
4  static void taskq_ent_free(...) { ... }
5
6  static taskq_t
7  *taskq_create_common(...) { ...
8    // [different lock primitives below:]
9    mutex_enter(...);
10   taskq_ent_free(...); /* ← consistent ✓ */
11   ...
12 }
```

Getting back to actual bugs, here is a bad comment automatically detected by iComment:

```
1  /* mozilla/security/nss/lib/ssl/sslsnce.c: */
2  /* Caller must hold cache lock when calling this. */
3  static sslSessionID * ConvertToSID(...) { ... }
4
5  static sslSessionID *ServerSessionIDLookup(...)
6  {
7    ...
8    UnlockSet(cache, set); ...
9    sid = ConvertToSID(...);
10   ...
11 }
```

We observe a specification in the comment at line 2, and then a usage error at line 9 where we unlock the cache and then call `ConvertToSID`. The badness of the comment was confirmed by Mozilla developers.

Issue: Comments are not updated alongside code. **Bad comments can and do cause bugs.**

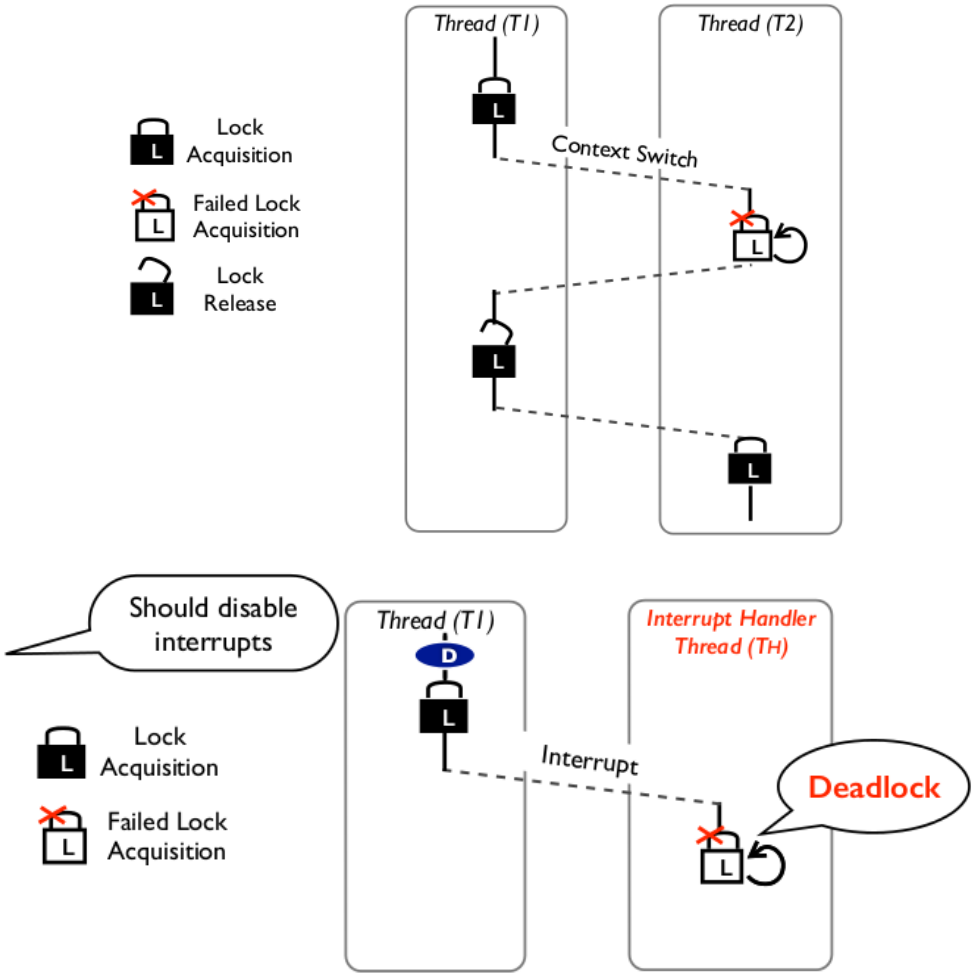
Here's another bad comment in the Linux kernel, also automatically detected.

```
1  // linux/drivers/ata/libata-core.c:
2
3  /* LOCKING: caller. */
4  void ata_dev_select(...) { ...}
5
6  int ata_dev_read_id(...) {
7    ...
8    ata_dev_select(...);
9    ...
10 }
```

Once again, the specification at line 3 states that the caller is to lock. But line 8 calls `ata_dev_select()` without holding the lock. The badness of this comment was confirmed by Linux developers.

## Deadlocks

Another concurrency problem is deadlocks. We focus on a particular form of deadlock here, which occurs when code may get interrupted by interrupt handlers, and the code shares locks with the interrupt handler. This problem inspired aComment [TZP11].




In particular, if the spinlock is taken by code that runs in interrupt context (either a hardware or software interrupt), then code requesting the lock must use the spin_lock form that disables interrupts. Otherwise, sooner or later, the code will deadlock. ಠ_ಠ

Here's an example of the right way to do things:

```
1    spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
2    unsigned long flags;
3    spin_lock_irqsave(&mr_lock, flags);
4    /* critical section... */
5    spin_lock_irqrestore(&mr_lock, flags);
```

- `spin_lock_irqsave()` disables interrupts locally and provides spinlock on symmetric multiprocessors (SMPs).
- `spin_lock_irqrestore()` restores interrupts to state when lock acquired.

This covers both interrupt and SMP concurrency issues.

## References


[ECCH00] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In Michael B. Jones and M. Frans Kaashoek, editors, *OSDI*, pages 1–16. USENIX Association, 2000.

[TYKZ07] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: bugs or bad comments?*/. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *SOSP*, pages 145–158. ACM, 2007.

[TZP11] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. acomment: mining annotations from comments and code to detect interrupt related concurrency bugs. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *ICSE*, pages 11–20. ACM, 2011.