Software Testing, Quality Assurance & Maintenance—Lecture 10

Patrick Lam
University of Waterloo

January 26, 2015

Course roadmap

- ✓ Introduction (faults etc)
- ✓ Graph coverage
- ☐ Testing Concurrent Programs

Concurrency In Your Curriculum

SE350 (OS):
    first exposure to multiprocessing

CS343 (Concurrency):
    (obvious)

ECE459 (4B, P4P):
    learn more about leveraging parallelism!

Context: Multicores, everywhere, today

For past 10 years, chips not getting faster.

Solution:

Multicores!

Today: if you want performance,
then you need parallelism.

Implication: 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

Race Conditions

credit: me

Alas!

```
#include <iostream>
#include <thread>

int counter = 0;

void func() {
    int tmp;
    tmp = counter;
    tmp++;
    counter = tmp;
}

int main() {
    std::thread t1(func);
    std::thread t2(func);
    t1.join();
    t2.join();
    std::cout << counter;

    return 0;
}
```

Racy Output

```
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
```

Race Conditions [from ECE459 slides]

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

When there's a race, the final state may not be the same as running one access to completion and then the other.

Race conditions arise between variables which are shared between threads.

Tools for Detecting Races

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

and more

Helgrind Example

```
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<>
==6486==    by 0x4021AE: std::_Bind_simple<void (*())()>::operator()() (fu
==6486==    by 0x402147: std::thread::_Impl<std::_Bind_simple<void (*())()
==6486==    by 0x4EF196F: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6
==6486==    by 0x4C2F056: 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<>
==6486==    by 0x4021AE: std::_Bind_simple<void (*())()>::operator()() (fu
==6486==    by 0x402147: std::thread::_Impl<std::_Bind_simple<void (*())()
==6486==    by 0x4EF196F: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6
==6486==    by 0x4C2F056: 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"
```

Eliminating Races Ain't Enough

Race-freedom required by specification,
but doesn't guarantee bug-freedom.

```
#include <iostream>
#include <thread>
#include <mutex>

int counter = 0;
std::mutex m;

void func() {
    int tmp;

    m.lock();
    tmp = counter;
    m.unlock();
    tmp++;
    m.lock();
    counter = tmp;
    m.unlock();
}
```

Testing Race-Free Programs

Race free? That don't impress me much.
Now what?

- run your 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?

Reentrant/recursive Locks

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

Different threads:
second thread waits for first to unlock.

Reentrant/recursive Locks

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

Different threads:
    second thread waits for first to unlock.

Same thread:
    first thread waits for first to unlock. . . forever!

However, you can use *recursive* locks.

Reentrant/recursive Locks

Each lock knows how many times its owner has locked it.

Must unlock same number of times to liberate.

Java locks work this way, e.g.

```
class SynchronizedIsRecursive {
  int x;

  synchronized void f() {
    x--;
    g(); // does not hang!
  }

  synchronized void g() {
    x++;
  }
}
```

Java `ReentrantLock`s

Although every Java object is a lock,
`ReentrantLocks` are more special.

- can explicitly `lock()` & `unlock()` them,
- or even `trylock()`!

CAVEAT: not cool to hog the lock—do this

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

Part I

Bad Lock Usage

Reference:

Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, Benjamin Chelf.
“Bugs as Deviant Behavior: a general approach to inferring errors in system code.”
ACM Symposium on Operating Systems Principles, 2001.

. . . nice try

```
/* 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();
```

Problem: lock() and unlock() must be paired!

Deriving “A() must be followed by B()”

“`a(); ... b();`”
denotes a MAY-belief that a() follows b().

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

Results: 23 errors, 11 false positives.

Reference:

Lin Tan, Ding Yuan, Gopal Krishna, Yuanyuan (YY) Zhou.
“/* iComment: Bugs or Bad Comments? */”.
ACM Symposium on Operating Systems Principles, 2007.

Locks in OpenSolaris

```
/* opensolaris/common/os/taskq.c: */
/* Assumes: tq->tq_lock is held. */
        /* ↪ consistent ✓ */
static void taskq_ent_free(...) { ... }

// ...

static taskq_t
*taskq_create_common(...) { ...
  // [different lock primitives below:]
  mutex_enter(...);
  taskq_ent_free(...); /* ← consistent ✓ */
  ...
}
```

Locks in Mozilla

A bad comment automatically detected by iComment:

```
/* mozilla/security/nss/lib/ssl/sslsnce.c: */
/* Caller must hold cache lock when calling this. */
static sslSessionID * ConvertToSID(...) { ... }

...

static sslSessionID *ServerSessionIDLookup(...)
{
  ...
  UnlockSet(cache, set); ...
  sid = ConvertToSID(...);
  ...
}
```

Specification in comment.

Lock released before calling ConvertToSID()

Mismatch!

Bad comment already confirmed by Mozilla developers after reporting.

Comments are not updated accordingly.

Bad comments can and do cause bugs.

Locks in the Linux kernel

Another bad comment automatically detected by iComment:

```
// linux/drivers/ata/libata-core.c:

/* LOCKING: caller. */
void ata_dev_select(...) { ...}

int ata_dev_read_id(...) {
  ...
  ata_dev_select(...);
  ...
}
```

Specification in comment.

No lock held before calling ata_dev_select.

Mismatch!

Bad comment already confirmed by Linux developers after reporting.

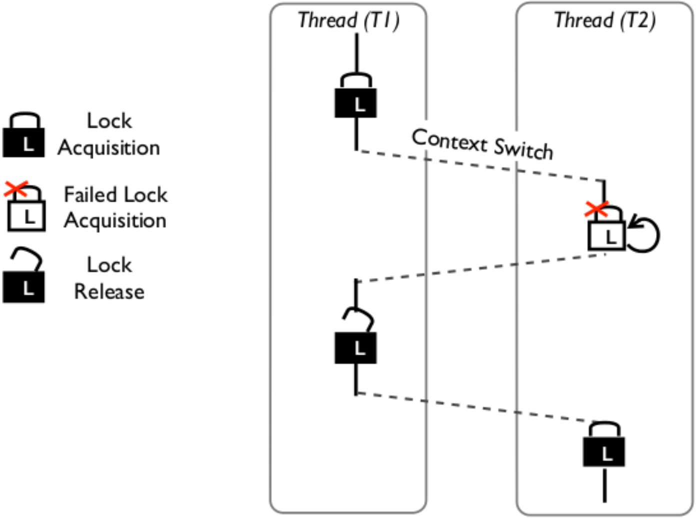
Deadlocks

Interrupts Complicate OS Synchronization

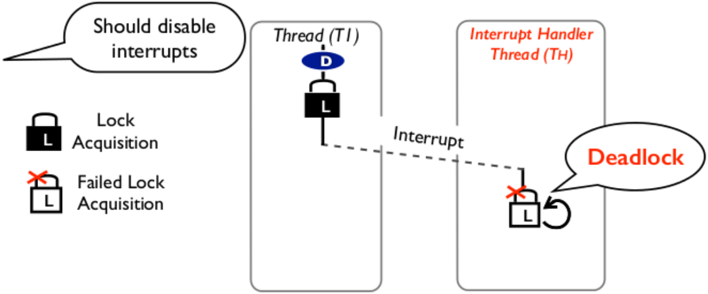
Reference:

Lin Tan, Yuanyuan (YY) Zhou, Yoann Padioleau.
"aComment: Mining Annotations from Comments and Code to Detect Interrupt-Related Concurrency Bugs."
International Conference on Software Engineering, 2011.

Interrupts Complicate OS Synchronization



Interrupts Complicate OS Synchronization



If: spinlock taken by code that runs in interrupt context (hw or sw).
Then: must use spin_lock form that disables interrupts.
Otherwise: sooner or later, you'll deadlock. ರ_ರ

Disabling interrupts: spin_lock_irqsave

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;
spin_lock_irqsave(&mr_lock, flags);
/* critical section... */
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.