Debugging Data Races

Dr. Cliff Click
rocket.realtime.school@gmail.com
cliffc@acm.org
cliffc.org/blog

Agenda

- What is a Data Race?
- Common Data Races
- Debugging Techniques & Tools
- QA & Testing
- Wrap Up

A Short Debugging Tale


```
if (method.hasCode() != true)
  return false;
code = method.getCode();
...setup;
code.execute();  // Throws NPE rarely!!!
return true;
```

- Example is real; simplified for slide
  - Many more wrapper layers removed
  - Shown “as if” aggressive inlining already
- I've debugged dozens of slight variations
- Apparently I'm not alone:

http://opera.cs.uiuc.edu/paper/asplos122-lu.pdf

Learning from mistakes --
A Comprehensive Study on Real World Concurrency Bug Characteristics

What IS a Data Race?

- Formally:
  - Two threads accessing the same memory
  - At least one is writing
  - And no language-level ordering
- Informally:
  - Broken attempt to use more CPUs
  - (but can happen with 1 CPU)
- Generally because 1 CPU is too slow
  - End of frequency scaling   :-(
  - Multi-core, big server, etc

Timeline of a Data Race

T1	T2
	flush_old_cache
r1 = _code if(!r1) ...	
	_code = null
r1 = _code call exec(r1)	

- **Two threads**

Timeline of a Data Race

T1	T2
	flush_old_cache
r1 = _code if(!r1) ...	
	_code = null
r1 = _code call exec(r1)	

- Two threads
- **Accessing same memory**

Timeline of a Data Race

T1	T2
	flush_old_cache
r1 = _code if(!r1) ...	
	_code = null
r1 = _code call exec(r1)	

- Two threads
- Accessing same memory
- **At least one is writing**

Timeline of a Data Race

T1	T2
	flush_old_cache
r1 = _code if(!r1) ...	
	_code = null
r1 = _code call exec(r1)	

- Two threads
- Accessing same memory
- At least one is writing
- **No language-level ordering**

Timeline of a Data Race

rocketrealtime.com

T1	T2
	flush_old_cache
r1 = _code if(!r1) ...	
	_code = null
r1 = _code call exec(r1)	

- OK if:
  - Write before 1st Read OR
  - Write after 2nd Read
- Broken if in-between
- Pot-Luck based on OS thread schedule
- Crashes rarely in testing
- More context switches under heavy load
- Crash routine in production

What IS a Data Race?

- When & Why can loads and stores move?
- Compiler moves for scheduling
- Hardware moves the effect for timing
  - Covering cache-miss costs
- Allowed unless explicitly denied
  - Via **lock/synchronized** or **volatile**
- Requires TWO or more threads… (obvious)
- Requires ordering on ALL threads
  - Not just on the writer...

Reordering Memory Ops

T1	T2
_data = stuff	
_init = true	
	r1 = _init if(!r1) ...
	r2 = _data

- Writing 2 fields
- Can T2 see stale _data?
- Yes!

Reordering Memory Ops

T1	T2
	r2 = _data
_data = stuff	
_init = true	
	r1 = _init if(!r1) ...

- Writing 2 fields
- Can T2 see stale _data?
- Yes!
- **Compiler can reorder**
  - Standard faire for -O
- Java: make _init volatile
- C/C++: use ‘atomic’

Reordering Memory Ops

T1	T2
	r1 = _init
	if(!r1) ... // predict
	r2 = _data
_data = stuff	
_init = true	
	...r1 true
	...so keep r2

- Writing 2 fields
- Can T2 see stale _data?
- Yes!
- **Hardware can reorder**
- Load _init misses cache
- Predict r1==true
- Speculatively load _data early, hits cache
- _init comes back true
- Keep speculative _data

Reordering Memory Ops

T1	T2
_data = stuff –membar—— _init = true	
	r1 = _init if(!r1) ... –membar—— r2 = _data

- Writing 2 fields
- Can T2 see stale _datum?
- No!
- Need store-side ordering
- Need load-side ordering
- Included in Java volatile

Reordering Memory Ops

T1	T2

	r1 = _init
	if(!r1) ... // predict
	r2 = _data

_data = stuff membar _init = true	
	...r1 true ...so keep r2

- Writing 2 fields
- Can T2 see stale _data?
- Yes!
- **Missing load-ordering**
- Read of _init misses
- Predict branch
- Fetch _data early
- Confirm good branch

Reordering Memory Ops

T1	T2
_data = stuff	
_init = true	
	r1 = _init if(!r1) ... –membar—— r2 = _data
...stuff actually ...visible	

- Writing 2 fields
- Can T2 see stale _data?
- Yes!
- **Missing store ordering**
- Write of _data misses
- Write of _init hits cache
- T2 reads _init
- T2 reads stale _data

Agenda

- What is a Data Race?
- Common Data Races
- Debugging Techniques & Tools
- QA & Testing
- Wrap Up

Common Data Races

- My experiences only*
- Double-read with write in the middle:

```
if( _p != null )
  ... _p._fld...
```

```
_p = null;
```

- ...and it's usually a null write
- Two writes with a read in the middle:

```
_size *= 2;
_array=new[_size];
```

```
..._array[_size-1]...
```

- Double Checked Locking:

```
if( _global == null )
  synchronized(x)
    if( _global == null )
      _global = new ...;
```

*Learning From Mistakes – ASPLOS 2008

Double Checked Locking

T1	T2
r1 = new ...	
r1._fld = stuff	
...write misses	
_global = r1	
	r1 = _global
	if(!r1) ...
	r2 = r1._fld
...write of _fld	
...happens here	

–membar——
unlock

- Initializing global singleton
- Can T2 see stale _fld?
- Yes!
- Misplaced store-ordering
- Unlock puts barrier AFTER both writes
  - Not between
- Fix: make _global volatile

More On Double-Read

- `if( _p != null ) {..._p._fld...}`
- Compiler likes to CSE both loads together
  - No bug if CSE'd together
  - C: Crashes in debug build, not product build
  - Java: Crashes before high-opt JIT kicks in
- Crashes when context-switch between reads
- i.e., just as heavy load hits system
- If you survive startup, might last a long time
- Bug can persist for years
  - Plenty of personal experience here...

Getting Clever w/HashMap

- Common caching case: rare writer, many readers
- Using a HashMap unsafely and catching NPE
  - But not catching rarer AIOOBE
  - Bug bit both a customer AND in-house engineer
- Idea: HashMap w/single writer, many readers
  - Thinking: No locking needed since 1 writer
  - Readers sometimes see ½ of 'put'
  - Throw NPE occasionally; Fix: catch NPE & retry

Getting Clever w/HashMap

- Writer can be mid-resize, reader hashes to larger table
  - But does lookup on smaller table
  - Throws AIOOBE – not caught, program crash
- Reader calls size(), size() calls resize, and…
  - Reader is now writing (resizing) the table
  - Other Reader throws AIOOBE – not caught, program crash
  - Or list corrupted; cyclic..
    - touching threads hang forever spinning on the cycle
    - Transaction times out, retries
    - Threadpool launches another thread…. that also hangs
    - Slowly all cores burned on threads spinning in table
    - Server grinds to a halt

Agenda

- What is a Data Race?
- Common Data Races
- Debugging Techniques & Tools
- QA & Testing
- Wrap Up

Debugging Techniques

- Visual Inspection
  - Very slow, fairly accurate, “State of the Practice”
- Printing
  - Changes timing, can hide bugs, “HeisenBugs”
  - I use a better “printing” solution
- Static Analysis Tools
  - STILL not very good, decades later
  - FindBugs best easiest answer

Visual Inspection

- Easy to get started with
- Works on core files; works after the fact
  - Just obtaining the code is often a problem
- Very slow per LOC
- Sometimes can make a more directed search
  - e.g., Stack trace points out where somebody failed
  - Play mental Sherlock Holmes w/self
- Requires Memory Model expert, domain expert
- Does Not Scale

Visual Inspection

- Biggest Flaw: Not Knowing The Players
- Maintainer cannot name shared variables
  - Or which threads can access them
  - Or when they are allowed to touch
- Sometimes suffices to Make Access Explicit
  - Large Flashy Comments on shared variables
  - At least the Players become obvious
- Can also look for common failures

Visual Inspection

- Avoiding Double-Read:
  - Often requires changing accessor patterns
  - No more “if(isReady()) ... get()” pattern
  - Return flag & value in 1 shot, cache in a local variable:

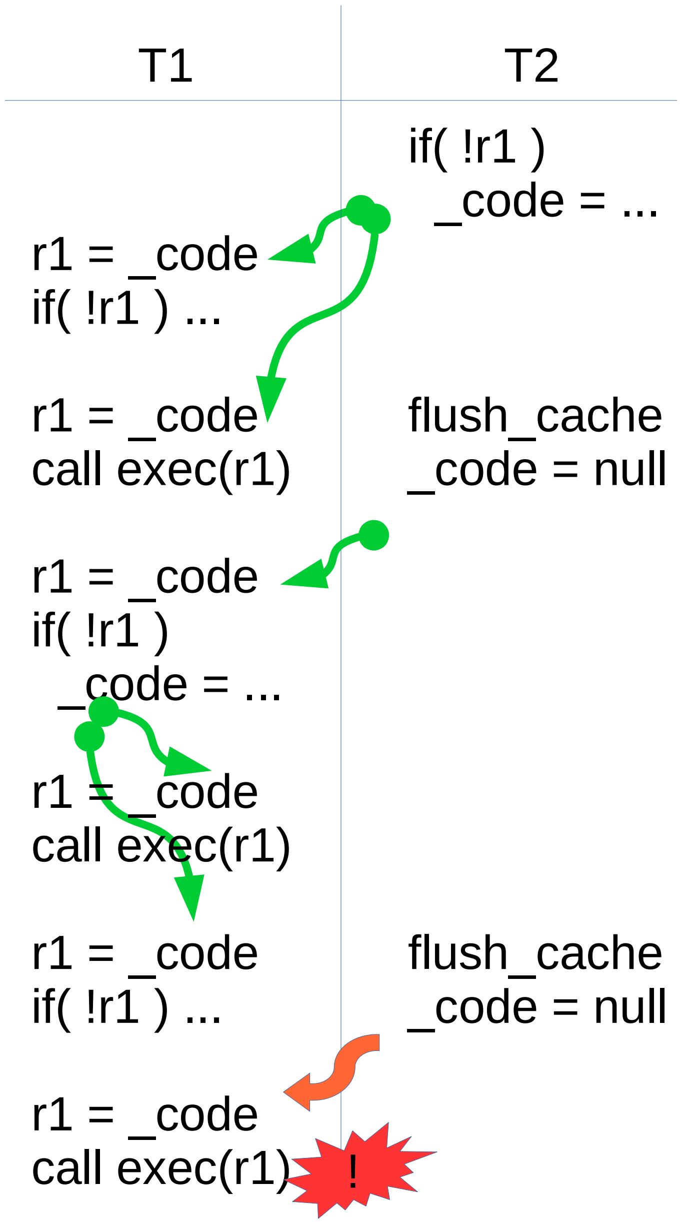
```
tmp = get();
if( tmp != null ) ...tmp._fld...
```

- No more accessors around fields
- Every field load & store is part of the algorithm
- And must be explicit to be inspected

Visual Inspection

- 2nd Biggest Flaw: forgetting "The Cycle"
- Concurrent code does: {start, do, end}
- Carefully inspect: two threads both doing {start, do, end}
- But code in a cycle!
  - ...start, do, end, stuff, start, do, end, stuff...
- Inspect: two threads both doing {end, stuff, start}
- Inspect: {start, do, end} vs {end, stuff, start}
  - (chasing each others' tail)

The Cycle



- Endlessly Repeat Cycle:

```
r1=_code;
if(!r1) _code=...;
r1=_code;
call exec(r1);
flush;
_code=null;
```

- On multiple threads
- Must check all interleavings

Printing

- Printing / Logging / Events
  - "Make noise" at each read/write of shared variable
  - Inspect trace after crash
  - Serialized results...
  - ...but I/O blocks; changes timing, hides data-race bugs
  - Also OS can buffer per-thread; WYSI is not WYG
- Well known "HeisenBug" symptom:
  - Never crashes when printing
  - Or under the debugger
  - Or on my desktop

Cheaper "Printing"

- I use this hack for HeisenBugs:
- Write "event tokens" (ints) to per-thread ring-buffer
  - Per-thread buffer: No contention to write
  - Tokens: **no complex String creation**, no object creation, no cache-misses
  - Ring buffer: much less overhead & **no blocking**
- Very less likely to hide bug
- Works Distributed!
  - Debugged H2O's clustering comms using this...
- Hard to read the results, so...

Printing Cheap “Printing”

- Per-Thread Ordering w/TimeStamp
  - Slap a System.nanoTime in the per-thread event buffer
- Post-process the crash
  - **Sort** all ring-buffers by nanoTime
  - **Print** a time-line just **before** the crash
  - AND **after** the crash
  - 99% chance the “guilty thread” stands out
- Rather heavy-weight technique
  - Need to know where to target it

Tools

rocketrealtime.com

- **Not Ready for Prime Time**
  - Most tools simply don't scale
  - 10x slowdowns, high false positive rates
  - Or require PhD to use
- Recommend: FindBugs
  - Scales to production use
  - Simple pattern-matching
    - But finds only the common bugs
    - The “common cold” is called “common” for a reason
  - Definitely limited in scope

Defensive Locking

- Protecting against “unexpected” Data Race
  - Lock unlocked Code & Collections
  - Supposed to be no contention
  - No-contention lock cost is low, so…
  - If no data race, very low cost
- Detecting “unexpected” Data Race:
  - Throw exception if racing in “unexpected” code
  - Requires extra word, ½ thin-lock cost, try-lock
  - Catches BOTH reader and writer
    - ... at moment of race!

Other Techniques

- Formal proofs?
  - Still not ready for prime-time
  - Although hardware designers make it work for them
- Statistical
  - I get X fails/month; what happens that often?
- I did a bunch of home-grown tools:
  - NISB: catch real races when they happen, 20xslower
  - Also Detect when common collections are used racily
  - Not widely available, more proof-of-concept

Agenda

- **What is a Data Race?**
- **Common Data Races**
- **Debugging Techniques & Tools**
- **QA & Testing**
- **Wrap Up**

Testing Concurrent Code

- Sequential
  - Deterministic
  - 100% code coverage
  - Repeatable, Reliable
  - Same results when:
    - Changing hardware
    - Changing memory
    - Changing load

- Concurrent
  - Non-deterministic
  - <<1% state coverage
  - Heisen-Bug
  - **Different** when results:
    - Changing hardware
    - Changing memory
    - Changing load

Testing Concurrent Code

- New failure modes:
  - Deadlock, livelock, missed signals, notifies
  - Synchronization and atomicity failures
  - Data races
  - Performance failures
- Failures in sequential code are deterministic:
  - Same input, same failure
- Failures in concurrent code are probabilistic:
  - Might require hugely unlucky timing to crash

Testing Concurrent Code

- Split out concurrency & application logic
  - Test app logic as normal single-threaded
  - Test concurrency without app complexity
  - Focus concurrency testing
- Requires a different QA plan for concurrency

QA Plan

- The goal of QA is not to “find all the bugs”
  - Because this is impossible
- Goal of QA is really to *increase confidence*
- QA approaches include:
  - Education, training, careful design
  - Code review
  - Static analysis (tools)
  - Testing
    - Unit, integration, load, performance tests
    - Statistical analysis of crashes

QA Plan

- “Absence of evidence is not evidence of absence”
  - Testing can only find errors, not correctness
  - Even more true with rare probabilistic failures
- Testing, code review, and static analysis are all subject to diminishing returns
  - Tend to find different types of problems
  - So combine them!
    Lame but true… worth doing it all

Code Reviews

- Expensive and Effective
  - Can spot bugs that occur rarely in practice
  - Can spot bugs that won't happen on specific hardware (e.g. desktop vs mobile)
  - Often improves general code and comment quality
- Might require a culture shift!!!
- And one that’s worth it
  - Education all around, buy-in to the solution

Static Analysis

- FindBugs…
  - Can check rules/patterns
    - e.g. “Hold a lock consistently when accessing a field”
    - Highly automatable
    - Plan to deal with false positives
- Annotate concurrency design!
  - Very helpful for both humans and automatic tools

Unit Tests

- Basic safety & liveness
  - If I do one X, can I do one Y?
- Basic concurrency:
  - If I do 10 X's in parallel, do 10 Y's also work?
  - Basic deadlock & concurrency testing
- Load testing:
  - If I do 1e6 X's in parallel, then 1e6 Y's, performance is "about" 1e6 times doing it once?
  - Tests rare timing-related events
  - Some livelock testing

Unit Test Framework Issues

- This blocks the usual test harnesses:

```
void test() {
  BoundedBlockingQueue buf = new BoundedBlockingQueue(1);
  buf.put("abc");
  buf.put("def");
  assertEquals("abc", buf.take());
  assertEquals("def", buf.take());
}
```

Queue full, so blocks

- Exceptions in Threads ignored:

```
void test() {
  UnboundedQueue buf = new UnboundedQueue();
  buf.put("abc");
  Thread t = new Thread() {() → assertEquals("oops", buf.take());}
  t.join();
}
```

Exception thrown, thread dies

Thread joins, test completes normal

White-Box Tests

- Controlled interleaving:
  - Force T1 to advance, then T2, then back to T1
  - Can force weird interleavings
  - While moving at “debugging-speed”
- Requires a new testing support harness
  - Internal clock for “ticks”
  - Block threads until “tick”, advance until “tick”
  - Hooks in code under test
- Can test e.g. blocking and narrow races

White Box Testing

- Harness maintains global clock
  - Only advance when all threads blocked
  - Can wait-till-clock-value
  - Plays well with debuggers (unlike sleep())

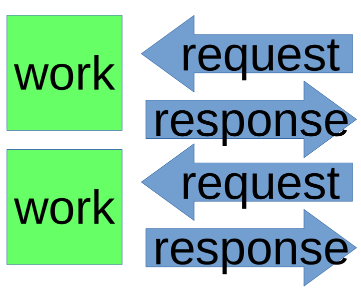
```
void T1() {
  buf.put("abc");
  assertEqual(0, getTick()); // blocks until T2 is wait-for-1
  buf.put("def");
  assertEqual(1, getTick()); // blocks until T2 exits
}

void T2() {
  waitForTick(1); // blocks until T1 is getTick()==0
  assertEquals("abc",buf.get());
  assertEquals("def",buf.get());
}
```

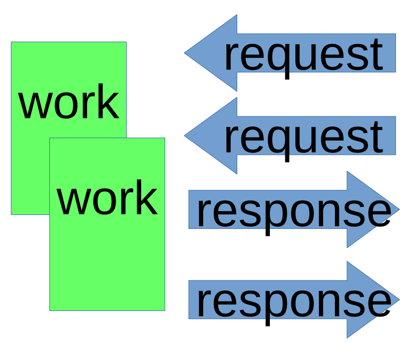
Load Testing

- Easy to say “do X 1e6 times”
  - Reasonable for Big Data / Batch processing
- Highly <u>unrealistic</u> for irregular compute!
  - Need a mix of request types **and** timing
  - You don’t get {nothing, a million page hits, nothing}
- Service Requests:
  - **`curl`** 1e6 mixed URLs to web server
  - But don’t fire off #N until #N-1 completes
  - So web server only services 1 request at a time

Load Testing

- Service Requests:
  - Not firing off request #N until #N-1 completes
  - Unrealistic single-threaded latency reported
- Must fire new requests with e.g. an exponential distribution, and **independent** of results
- Load tool must be parallel & concurrent as well!
- See Gil Tene’s work with the Jitter Meter





Load Testing

rocketrealtime.com

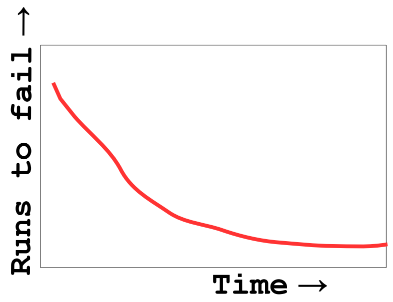
- “Lab” environment must match “production”
  - Full-speed network, full-speed DB, full-size gear
  - Or else, “lab performance testing” is unrealistic
  - Seen this conflict in many large companies
- Can’t (typically) test system at full load from dev desk
  - So all kinds of weird behaviors only show up later
- Worth spending the hardware $$$ to get smoother lab → production workflow

Statistical Testing

- See failure in testing, hard to repeat
  - Never fails on desktop, or with more debug logic
- Same solution as hardware guys:
  - Statistics!
  - Repeat-until and count failure rate
- Get a machine dedicated to the problem
- Run it hard, under heavy load, over and over
  - Days maybe
- See what the failure rate is...

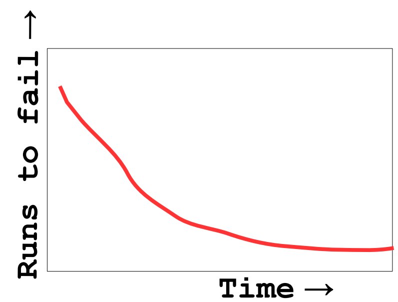
Statistical Testing

- Now you have a "guaranteed fail" box
  - Just takes, e.g. a day, or large X runs to make sure
- Slowly add debug info, logging, printouts
  - Fail rate doesn't change?
    - You're probably not "too close" to the bug
  - Fail rate drops off?
    - You're tweaking important timing, near the bug
- Basically, you can now zero in on the bug
  - But it just takes X runs to make sure
    - Where X might be big



Statistical Testing

rocketrealtime.com



- Once you have the bug fix in hand
  - Box can be used to test “enough”
- And likely its more than one bug, interlocked
  - So you’ll go back to the same setup a few times
- General rule:

Probabilistic events require many runs and get tracked with statistics

- And this works for software, same as hardware

Distributed App Testing

- Much of parallel advice applies
  - Replace “data race in shared memory” with “network race in shared cluster”
  - Replace “how many threads” with “how many nodes”?
    - And 5 is a good start
- Can test on vboxs or even processes on 1 real hardware node
  - Network costs really low so…
- MUST also test with real network latencies

Distributed App Testing

- Need real latencies to see real interleavings
- Need real loads same as parallel case
- Also: inject network failures
  - Dropped / dup'ed UDP packets
  - Broken TCP connections
  - Retry logic will get used, needs testing also
- Load testing, statistical testing all apply
  - Used the "cheaper printing" to help debug H2O

Agenda

- **What is a Data Race?**
- **Common Data Races**
- **Debugging Techniques & Tools**
- **QA & Testing**
- **Wrap Up**

Writing Data Races

rocketrealtime.com

- Anti-patterns:
  - Double-Checked Locking

```
if( _global == null )
  synchronized(x)
    if( _global == null )
      _global = new ...;
```

  - Or double-read w/rare null writer
    - Hidden by accessors

```
if( hasFoo() )
  getFoo().doIt();
```

  - Multiple calls to a thread-safe collection are ***not*** thread-safe between calls:

```
if( (tmp=cache.get(Key)) == null )
      cache.put(Key,(tmp=compute_value()));
...tmp...
```

  - Two racing writers compute 2 `tmp`'s
  - Each thinks it has the **only** copy, both are updated
  - And one of the updates is lost

Writing Data Races

- Often hidden by Good Programming Practice
- Already solving a Large, Complex Problem
- Using abstraction, accessors
  - Giving meaning to memory access
  - In context of Large, Complex Problem
- Need more speed
- So introduce Concurrency, Threads

The Pitfall

- End up adding Concurrency to Large Complex Problem
- **Fail to recognize Concurrency it's own (subtle) Complex Problem**
- Needs its own kinds of wrappers, access control
  - Design API around concurrent access!
  - It's not a bolt-on after-the-fact kind of feature
- Interviews w/Data-Race Victims:
  - Don't know which thread can touch what or when
  - Surprised by the interleaving that triggers the bug

Don't Go There...

- Best answer: don't write concurrency bugs!
- Use the 'immutable' object pattern
- Use private data
- Use well-tested java.util.concurrency.*

But When You Must...

- Admit to self: ***Here Be Dragons***
- Think Before You Write, and ...
- Document, Document, Document!

Q&A

Debugging Data Races

Dr. Cliff Click
rocket.realtime.school@gmail.com
cliffc@acm.org
cliffc.org/blog