Testing Concurrent Software

Bill Pugh
Professor of Computer Science, University of Maryland

Brian Goetz
Senior Staff Engineer, Sun Microsystems

Cliff Click
Distinguished Engineer, Azul Systems

TS-2220
The Bottom Line

Some good news, some bad news

Testing concurrent software is difficult, but not impossible.

By a combination of multiple techniques (careful design, static analysis, code review, extensive testing), you can get the upper hand on concurrency bugs.
What This Talk Is, and Isn't

- Building correct concurrent software is a big topic
  - We can't teach you to do that in an hour (or a week)
- We'll discuss ways for effectively creating tests as part of a QA plan for concurrent software
- We assume you already have idea of what to do and what not to do
  - See also
    - *Java Concurrency in Practice*, Goetz et al.
    - *Concurrent Programming in Java*, Lea
    - TS-2388: Effective Concurrency for the Java Platform (Friday, 10:50am)
Agenda

Introduction
Creating a Test Plan
Unit Testing
Concurrent Failure Modes
Performance Testing
System Testing
Summary
Testing Concurrent Software

Like testing sequential code...

- Test cases for sequential code...
  - ...may test safety or performance (or both)
  - ...exercise code and assert invariants and postconditions
  - ...try to explore as much of the state space as possible
    - One rough measure of this is code coverage
  - ...try to find combinations of inputs and actions that are most likely to cause failure

- Test cases for concurrent code do the same
  - So we already know how to do it, right?
Testing Concurrent Software
Like testing sequential code... but different

- Concurrent programs have more failure modes than sequential ones
  - Liveness failures: deadlock, livelock, missed signals
  - Safety failures: synchronization errors, atomicity failures

- Failures in sequential programs are largely *deterministic*
  - Same input, same failure

- Many failures unique to concurrent programs are *rare probabilistic events*
  - Some bugs require exquisitely unlucky timing
Testing Concurrent Software
More extensive testing required

- State space is much larger due to thread interactions
- Need more intensive tests
  - Run for longer periods
  - Look for rare probabilistic failures
  - Account for impact of GC, JITing, etc
- Must test on multiple platforms
  - Different CPU architectures, JVMs, number of CPUs
  - Some tests don't happen on some architectures
- Tests must be written to avoid masking bugs
Design For Testability

Concurrent programming is hard enough

• Where possible, separate concurrency logic from business and functional logic
  • Concurrency is challenging enough
  • Even harder when mixed in with your business logic!

• Isolate concurrency by extracting concurrent abstractions
  • Such as bounded buffers, semaphores, thread pools
  • Use the ones from java.util.concurrent where possible
    • Implement your own only if the provided ones don't fit

• Testing a single concurrent abstraction is a lot easier than testing an entire application
Agenda

Introduction
Creating a Test Plan
Unit Testing
Concurrent Failure Modes
Performance Testing
System Testing
Summary
Building a QA Plan

Testing is only part of it

- 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
    - Understanding the concurrent design/implementation of what you have
  - Manual code review
  - Static analysis (automated code review)
  - Testing
    - Unit tests, load tests, performance tests, system tests
Building a QA Plan

Testing is only part of it

- Testing can never show the absence of errors, only their presence
  - Even more true with rare probabilistic failures
- Testing, code review, and reviewing analysis reports are all subject to *diminishing returns*
  - Luckily, also tend to find different types of problems
- By combining them, you buy more confidence for your QA budget than testing alone
Manual Code Review

Expensive, but effective

- Expert review is often the best way to find subtle concurrency bugs
  - Can spot bugs that occur extremely rarely in practice
  - Can find bugs that won't happen on specific hardware
  - Often improves general code and comment quality

- Doesn't scale well
  - Useful for small, isolated concurrent components
  - Really, really hard, even for experts, to manually review large or subtle components

- Expensive to do frequently
  - Typically done by senior developers or consultants
Static Analysis
Automated code review

- Analyzes a program without running it
- Can check rules/patterns
  - Such as “Hold a lock consistently when accessing a field”
- Annotations that document concurrency design are very helpful
  - For both humans and automatic tools
  - See Java Concurrency in Practice, FindBugs, and Fluid from SureLogic
- See TS-2007: Improving Software Quality with Static Analysis
Concurrent Testing Scenarios

Lots of reasons to test...

- Unit testing functionality
  - Basic tests of safety and liveness (can be sequential)
- Unit testing functionality under concurrent stress
  - Looking for rare, timing-related interactions
  - Attempting to explore more of the state space
- Component performance testing
  - Evaluate performance or scalability of a concurrent abstraction under varying load
- System stress testing
  - Test a large application to see if it works
Agenda

Introduction
Creating a Test Plan
Unit Testing
Concurrent Failure Modes
Performance Testing
System Testing
Summary
Unit Testing

Don't forget the basics

• Start with basic unit tests
  • Some tests can be sequential – goal is to establish that documented sequential functionality works at all
    • Easier to debug basic functionality in sequential environment
  • But many concurrent classes have behavior that cannot be tested with just one thread
    • Testing blocking behavior requires at least two threads
      • One thread that performs an operation that blocks
      • Another thread that then performs an action that unblocks the first thread
Unit Testing
Some behaviors require multiple threads to test

- **Exchanger**
  - Inherently requires two threads to exchange

- **CyclicBarrier**
  - Inherently requires N threads to reach a barrier point

- **Lock**
  - If one thread holds it, does it actually block other threads?
  - When holding thread releases it, can another acquire it?

- **BlockingQueue**
  - Threads block if they try to add too many elements
  - Blocked threads unblock when room is made
  - Threads block if they try to remove nonexistent elements
Unit Testing

Framework support

• JUnit 4 and TestNG support timeouts

• TestNG supports concurrent testing
  • To allow tests to finish faster
  • For stress testing

• Addons to JUnit 4 also support concurrent testing

• But neither provides good support for single test cases that require coordination of multiple threads
Unit Testing

More framework support needed

```java
void testPutThenTake() throws InterruptedException {
    BoundedBlockingQueue<Integer> buf
        = new BoundedBlockingQueue<Integer>(1);

    buf.put(42);
    assertEquals(42, buf.take());
}

void testPutPutTakeTake() throws InterruptedException {
    BoundedBlockingQueue<Integer> buf
        = new BoundedBlockingQueue<Integer>(1);
    buf.put(42);
    buf.put(17);
    assertEquals(42, buf.take());
    assertEquals(17, buf.take());
}
```

This blocks and can’t get unstuck
Unit Testing
More framework support needed

```java
void testPutPutTakeTake() throws InterruptedException {
    final BoundedBlockingQueue<Integer> buf
        = new BoundedBlockingQueue<Integer>(1);
    Thread t = new Thread() {
        public void run() {
            assertEquals(42, buf.take());
            assertEquals(17, buf.take());
        }
    };
    t.start();
    buf.put(42);
    buf.put(17);
    t.join();
}
```

Won't compile; `take()` throws `InterruptedException`

Assertion failure won't be noticed by JUnit
Unit Testing
More framework support needed

• Exception in second thread isn't seen by JUnit
  • Propagates up call stack of thread
    • Printed to console
  • Test always passes
    • JUnit unaware of exception

• Must ensure that exception in any thread is propagated back to the testing framework
  • Requires lots of messy boilerplate code
  • Runnables can't throw checked exceptions

• We need something better
Unit Testing
Necessity is the mother of invention

- At UMD, we teach writing concurrent abstractions
  - Blocking queue, etc.
- We have a fairly elaborate automated system for testing functional correctness of student work
  - The Marmoset project
- Need to have reliable, repeatable tests for concurrent functionality
  - And allow students to write such tests
- Developed new framework for concurrent tests
  - Which you can download and use
MultithreadedTestCase (aka MTC)
Adding support for multiple test threads

- Same test, rewritten with MTC
  - Framework infers test lifecycle from method names

```java
class TestPutPutTakeTake extends MTC {
    BoundedBlockingQueue<Integer> buf;

    void initialize() {
        buf = new BoundedBlockingQueue<Integer>(1);
    }

    void threadPutPut() throws InterruptedException {
        buf.put(42);
        buf.put(17);
    }

    void threadTakeTake() throws InterruptedException {
        assertEquals(42, buf.take());
        assertEquals(17, buf.take());
    }
}
```
Multithreaded Test Case

Adding support for multiple test threads

- Uses same ideas as JUnit
  - Run `initialize()` method (if it exists)
  - Run all `threadXxx()` methods concurrently
  - Run `finish()` method (if it exists)

- yeah, doing it with annotations would be cooler
  - But just needed something that worked

- Does this test case test what we wanted?
  - No, didn't check blocking behavior

- Can use `sleep` and `System.currentTimeMillis`
  - Imprecise, doesn't work with debuggers, ugly
class MyTest extends MultithreadedTestCase {
    OnePlaceBuffer<Integer> buf;
    public void initialize() {
        buf = new OnePlaceBuffer<Integer>();
    }
    public void thread1() {
        buf.put(42);
    }
    public void thread2() {
        Thread.sleep(1000);
        buf.put(17);
    }
    public void thread3() {
        Thread.sleep(2000);
        assertEquals(42, buf.get());
    }
    public void thread4() {
        Thread.sleep(3000);
        assertEquals(17, buf.get());
    }
}
Unit Testing Blocking Operations

Thread 1

Tick 0
- put 42

Tick 1
- put 17 (blocks)

Thread 2

- wait for tick 1
- take 42
- take 17

This call to put should not return until after the call to take has started.
Unit Testing

Adding support for blocking operations

- System maintains a global *tick counter*
  - Starts at zero
  - Advanced only when all threads are waiting/blocked
  - Tests can wait until counter gets to a particular value
  - Tests can check the current value

- Plays well with debuggers
  - unlike using Thread.sleep()
Unit Testing
Using the tick counter to test blocking operations

- With tick counter support, we can now test blocking operations

```java
void threadPutPut() throws InterruptedException {
    buf.put(42);
    assertEqual(0, getTick());
    buf.put(17);
    assertEqual(1, getTick());
}

void threadGetGet() throws InterruptedException {
    waitForTick(1);
    assertEquals(42, buf.take());
    assertEquals(17, buf.take());
}
```
Example: Unit Testing a Lock
Using the tick counter to test blocking operations

```java
void threadFirstLocker() {
    lock.lock();
    assertEquals(0, getTick());
    waitForTick(2);
    lock.unlock();
}

void threadSecondLocker() {
    waitForTick(1);
    assertFalse(lock.tryLock());
    assertEquals(1, getTick());
    lock.lock();
    assertEquals(2, getTick());
    lock.unlock();
}
```
MTC – History and Future

Try it – and contribute!

• We've been using this
  • In courses at Univ. of Maryland
  • To rewrite all of the TCK tests for JSR-166
    • Results are a lot simpler than the original 166 TCK tests!

• Once you've constructed a test case
  • Can run it once (for tests designed to be deterministic)
  • Can run it many times (for nondeterministic tests)

• Our implementation available from:
  • http://findbugs.cs.umd.edu/

• Hopefully, someone else will improve on it
Runtime tools

- You can use various tools to perform dynamic instrumentation and testing for concurrency:
  - data race detection
  - introduce additional thread interleaving
Preventing Data Races

• One programming technique to prevent races:
  • Ensure that for every shared field x, there is some lock L such that no thread accesses x without holding lock L

• Note: This is not the *only* way to avoid races
  • There are fancier, more complicated techniques
Detecting Races with checkSync

- Algorithm
  - Locks_held(t) = set of locks held by thread t
  - For each shared field x, C(x) := { all locks }
  - On each access to x by thread t,
    - C(x) := C(x) ∩ locks_held(t)
    - If C(x) = ∅ then issue a warning

- From Savage et al, “Eraser: A Dynamic Race Detector for Multithreaded Programs,” TOCS 1997
An Improvement

- Unsynchronized *reads* of a shared location are OK
  - As long as no one *writes* to the field after it becomes shared
- Track state of each field
  - Only enforce locking protocol when it becomes shared and written
Agenda

Introduction
Creating a Test Plan
Unit Testing
Concurrent Failure Modes
Performance Testing
System Testing
Summary
Concurrent Failure Modes

Things that can't go wrong in sequential programs

- Most features of the Java language are designed for repeatability across runs and platforms
  - e.g., floating point behavior
- ...except for *threads*
  - Even correct programs can vary their behavior
  - Some errors only manifested through very particular interleavings or timings
- Many failures in concurrent programs are rare, probabilistic events

* (and identity hash code)
Concurrent Failure Modes

Synchronization errors

• If a variable (field or array element):
  • Is accessed by two or more threads, and
  • At least one of those accesses is a write, and
  • The variable is not a volatile field

• Then the accesses must be ordered by synchronization
  • synchronized, java.util.concurrent.locks.Lock

• Otherwise, your code is bad
  • Code with synchronization errors has exceptionally subtle semantics
Concurrent Failure Modes

Atomicity failures

- Even without synchronization errors, can still have nasty, timing-dependent concurrency bugs
  - Occur when threads interact in an unexpected way
- These are usually *atomicity failures*
  - A sequence of actions thought of as an atomic unit, but not adequately protected from interference
- Volatiles cannot prevent atomicity failures!
  - Requires using locking or atomic variables
Concurrent Failure Modes

Atomicity failures

• Typical causes of atomicity failures
  • Check-then-act
    
    ```java
    if (foo != null) { // Another thread could set
      foo.doSomething(); // foo to null
    }
    
    Value v = map.get(k); // Even if Map is thread-safe,
    if (v == null) { // two threads might call get,
      v = new Value(k); // both see null, and both
      map.put(k, v); // add a new Value to map
    }
    ```

  • Read-modify-write
    
    ```java
    ++numRequests; // Really three separate actions
    // (even if volatile)
    ```
Concurrent Failure Modes

Rare interleavings

- Some interleavings are rare if interpreted
  - Compiler can aggressive reorder operations
    - Invisible to correctly synchronized code

- Some interleavings are rare on a 1-CPU system
  - OS context switches only happen at designated points

- More CPU's generate more interleavings;
  Want more threads than CPUs
  - About twice as many active threads as cores is generally good
Concurrent Failure Modes
Generating more interleavings

• Use a multicore or multiprocessor system
• Avoid synchronization in test harness or debugging code
  • e.g. System.out.println()
  • May cause bugs to disappear
• Or force “bad” interleavings
  • e.g. barrier sync before suspicious code
  • Sprinkling Thread.yield() or Thread.sleep()
  • Perhaps with a bytecode rewriting tool
Testing Components

Testing for races

- Generate as many interleavings as possible
- Main challenge: find testable properties that
  - Fail with high probability if something goes wrong
  - Don't artificially limit the concurrency of the test
  - Introduce no additional synchronization
- Errors may be masked by the test program
  - Test program messes with timings
  - Test program synchronization may mask data races
  - Delays in test program may mask race conditions
Testing Components

Testing for races

- Obvious test for bounded buffer: everything that goes in comes out (and no extras)
  - Without getting in the way...
- Checksum elements as they go in or out
  - Keep per-thread checksums, combine them at end
    - So no synchronization during test run!
  - Need an order-insensitive checksum (e.g., sum, xor)
  - Use deterministic termination criteria
- Don't share RNGs between threads
- Prevent compiler from “pruning” code you are testing
Testing Components
Testing under concurrent stress

```java
void testPutsAndTakes() {
    for (int i = 0; i < nPairs; i++) {
        pool.execute(new Producer());
        pool.execute(new Consumer());
    }
    barrier.await(); // wait for all threads to be ready
    barrier.await(); // wait for all threads to finish
    assertEquals(putSum.get(), takeSum.get());
}

class Consumer implements Runnable {
    public void run() {
        try {
            barrier.await();
            int sum = 0;
            for (int i = nTrials; i > 0; --i)
                sum += bb.take();
            takeSum.getAndAdd(sum);
            barrier.await();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```
Experience at Azul
The world is full of undiagnosed synchronization errors

• When customer's code fails
  • Azul's JVM has a switch enables a version of java.util that checks at runtime for concurrent access to non-thread-safe collections
    • And throws an exception when it finds it
    • On both threads

• Slight performance hit, but decent at finding bugs
• We've implemented our own that you can use
  • Added to JSR166 contributions
Lock Implementations for Debugging

Tools for building test cases

- **UncontendableLock**
  - Implements Lock, but throws an exception if contention is actually seen
  - Use when your design says you don't need a lock – but want to verify that at runtime
    - Use runtime flag choose this or NoOpLock
    - Also a ReadWriteLock version

- **SlowReleasingLock**
  - Delegates to ReentrantLock
  - But pauses after releasing a lock
    - Will cause atomicity failures to be more common
Lock Implementations for Debugging
Contributed to JSR166

- Look at contributions section of JSR166
  - [http://gee.oswego.edu/dl/concurrency-interest/](http://gee.oswego.edu/dl/concurrency-interest/)
- These and related locks for debugging
- Should Java 7 support runtime flags to check for bad concurrent access to standard non-thread-safe classes?
  - One extra field
  - Minimal overhead if not enabled
  - About half the cost of regular locks if enabled
Dynamic Tools For Debugging

- We've talked about just a few ideas for trying to identify probabilistic faults
- This is an active research area
  - Keep your eyes out for other tools that can help
- For example, IBM's *ConTest*
  
  
  - Systematically and transparently schedules execution to increase the likelihood that race conditions, deadlocks and other intermittent bugs will appear
Agenda

Introduction
Creating a Test Plan
Unit Testing
Concurrent Failure Modes
Performance Testing
Summary
Performance Testing
Scalability vs. Performance

• How fast is it?
  • Without contention?
  • With expected contention?

• Does performance fall off a cliff under higher than expected contention?

• Performance tests must reflect realistic use cases
  • Selecting these is often the hardest part
  • Usually extensions of safety tests

• Secondary goal: empirically select parameters
  • Buffer sizes, queue sizes, pool sizes
Performance Testing

Parallel bottlenecks

• Need to watch out for contention points
  • Bottlenecks that don't scale with your application

• One bottleneck can prevent the entire application from scaling

• If it isn't a bottleneck, keep it simple
  • A simple, blocking, thread-safe class is going to be easier to get right than one designed for concurrent access
Performance Testing

Tool support
Performance Testing
Using JMX and jconsole to measure contention

- Can access JMX through jconsole
- ```setThreadContentionMonitoringEnabled(true)``` 
  - Allows you to get total time spent waiting for contended locks
  - Can also set this through jconsole
- Won't tell you which lock is contended
  - But will tell you if you have an issue
Performance Testing

GC bottlenecks

- Never call `System.gc()`
  - Forces a horrible, slow, stop the world collection

- If you use any RMI or EJB, Sun's JVM calls `System.gc()` every 60 seconds
  - Bug # 4403367
  - Totally kills scalability, particularly with large heap

- Workaround for Sun's bug
  - Set `-Dsun.rmi.dgc.server.gcInterval=2000000000`
Performance Testing

Document concurrency requirements

• Document whether a class is supposed to handle concurrent requests
  • Concurrent classes are not just thread-safe – they are designed to perform well under concurrent access

• Document how many concurrent operations it can handle
  • With default parameters, ConcurrentHashMap tops out at about 16 concurrent updates
    • But effectively no limit on concurrent reads

• Test to see if your expectations are being met
Performance Testing

What are we testing for?

• Performance tests often derived from safety tests
  • With some timing added

• Can learn many things from performance tests
  • Throughput under specific parameters
  • Sensitivity to varying parameters
  • Scalability with increasing thread count

• Exercise care applying results of component tests
  • Most tests are unrealistic simulations of the application
  • Component tests usually focus on extreme contention
Performance Testing

Common pitfalls

• Watch out for these when writing performance tests!
  • Introducing timing or synchronization artifacts
  • Not accounting for compilation or GC
  • Unrealistic sampling of code paths
  • Unrealistic degrees of contention
  • Dead code elimination
    • Make sure every result is used and unguessable

• Avoiding these often requires “tricking” the compiler – which is hard!
Agenda

Introduction
Creating a Test Plan
Unit Testing
Concurrent Failure Modes
Performance Testing
System Testing
Summary
System Testing

Touchpoints

• Get a machine with as many cores as possible
  • At least as many as will be used in production

• Log every error
  • If an probabilistic error occurs only once every 4 hours, you need to have good logging

• Verify concurrent expectations
  • Use UncontendableLocks where appropriate
  • If a method is only supposed to be invoked in the event thread, check it
System Testing

Using aspects

• You can use Aspect Oriented Programming (AOP) to inject runtime assertions
  • That System.gc isn't called
  • That Swing methods are called from the event thread

• Or to swap in debugging versions of classes
  • Substitute versions of HashMap that check for improper concurrent access
  • Substitute version of Lock that looks for deadlock risks

• See “Testing with Leverage, part III” (Goetz)
  • Contains precooked code, ready to use
Agenda

Introduction
Creating a Test Plan
Unit Testing
Concurrent Failure Modes
Performance Testing
System Testing
Summary
Summary...

- Testing concurrent software is hard!
  - Keep your expectations appropriate
  - Testing is not going to give high confidence you don't have rare probabilistic bugs

- Separate business logic from concurrency logic
  - Easier to get each right
  - Easier to test

- Use precooked code, already picked over by experts, when possible
  - java.util.concurrent is pretty darn good
  - But only because they've done everything recommended here, fixing bugs in the process
For More Information

- Other sessions and BOFs
  - TS-2388: Effective Concurrency for the Java Platform (Friday, 10:50am)
  - TS-2007: Improving Software Quality with Static Analysis
  - BOF-2864: Experiences with Debugging Data Races

- Books
  - *Java Concurrency in Practice*, Goetz et. al.
  - *Concurrent Programming in Java*, Doug Lea
Q&A

Bill Pugh
Professor of Computer Science, University of Maryland

Brian Goetz
Senior Staff Engineer, Sun Microsystems

Cliff Click
Distinguished Engineer, Azul Systems