Concurrency WS 2010/2011
Testing Concurrent Programs for Correctness and Performance

Annette Bieniusa

December 20, 2010

Testing concurrent programs[1]

- Use and extend ideas for testing sequential programs
- Nondeterminism (scheduling, interaction of threads,...) increase the complexity
- Test code can influence the timing or synchronization of your code!

[1] Slides are based on the book Java Concurrency in Practice

Kinds of tests

- Tests for safety: usually check for invariants
  - e.g. in a linked list, compare size of list with number of elements
  - need some sort of atomic snapshot or test points
- Tests for liveness: progress and nonprogress
  - Is a thread blocked or running slowly?
  - How long do you have to test?
- Tests for performance

Throughput the rate at which a set of concurrent tasks is completed;

Responsiveness the delay between request and completion of some action (also called latency);

Scalability the improvement in throughput as more resources are made available.

Outline

Testing for correctness

- Start with testing in a sequential context
- Identify invariants and postconditions (according to the specification!)
- Write unit tests, e.g. with JUnit

$\Rightarrow$ Remove first all bugs that are not related to concurrency issues!

Test setup

- Testing frameworks are usually not concurrency-friendly
- E.g. threads are not associated with a test run
- Threads have to report success or failure information back to the main test runner thread
- You have to provide start up/tear down/thread joining

Have a look at the testing framework for JSR166:

`http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/test/tck/JSR166TestCase.java`

Blocking Operations

How do you test that a thread *does not* proceed?

- If the method returns normally, the test has failed
- Problem: once the method blocks successfully, you have to convince it to unblock again
- Idea: make an interrupt after some time delay
- You might have to fine-tune the delay!

Example: BoundedBuffer

```
1   void testTakeBlockWhenEmpty() {
2     final BoundedBuffer<Integer> bb =
3         new BoundedBuffer<Integer>(10);
4     Thread taker = new Thread() {
5       public void run() {
6         try {
7           int unused = bb.take();
8           fail(); //if we get here, it is an error
9         } catch (InterruptedException success) { }
10      }};
11    try {
12      taker.start();
13      Thread.sleep(LOCK_DETECT_TIMEOUT);
14      taker.interrupt();
15      taker.join(LOCK_DETECT_TIMEOUT);
16      assertFalse(taker.isAlive());
17      //do not query Thread.getState()
18    } catch (Exception unexpected) {
19      fail();
20    }}
```

Detecting deadlocks

- Use open calls, i.e. never call an alien method while holding a lock
- Prefer timed lock attempts as this allows for adding log output
- Thread dumps show locking cycles (ctrl- on Unix, ctrl-Break on windows)

Outline

Detecting data races

- Chicken-egg problem: the test programs themselves are concurrent programs

Identify properties that can easily be checked and fail with high probability if something goes wrong, but do not let the testing code limit concurrency artificially.

$\Rightarrow$ Heisenbugs are bugs that disappear when you are debugging and testing

Detecting data races

Example: Order-insensitive checksum function for set implementations

- Use a checksum per thread and combine the results in the end
- Do not use consecutive integers as a smart compiler could precompute the result
- Beware: Random number generators can be a great bottleneck!
  $\Rightarrow$ Each thread needs own RNG, or a pseudo-random function (Marsaglia RNG)

```
1  //start input can be hashCode or nanoTime
2  static int xorShift(int y) {
3    y ^= (y << 6);
4    y ^= (y >>> 21);
5    y ^= (y << 7);
6    return y;
7  }
```

Generating more interleavings

- Use more active threads than CPUs
- Test on a variety of systems (different processor architectures, OS, VMs, ...)
- Employ `Thread.yield` or `Thread.sleep(10)`
- Special tools like ConTest

Outline

Testing performance

- What are typical usage patterns of the class under test?
- Optimize the common case, not the rare one
- Often used to determine sizings for various bounds (e.g. number of threads, buffer capacities)
- Include a timer when starting and shutting down the test threads

Timer

```
1   public class BarrierTimer implements Runnable {
2     private boolean started;
3     private long startTime, endTime;
4     public synchronized void run() {
5       long t = System.nanoTime();
6       if (!started) {
7         started = true;
8         startTime = t;
9       } else {
10        endTime = t;
11      }
12      public synchronized void clear() {
13        started = false;
14      }
15      public synchronized long getTime() {
16        return endTime - startTime;
17      }
18  }
```

Measuring responsiveness

- Small variance of service time gives better predictability
- Histograms of task completion give nice visualization
- Beware: Too small timer granularity can distort the measurements!

Pitfall: Garbage collection

- Timing of GC is unpredictable and can yield strange results (e.g. stop-the-world GCs)
- JVM flag -verbose:gc gives some information
- Solution 1: ensure that no GC run is necessary by increasing the heap size
- Solution 2: let the program run long enough to include several GC cycles
- Use realistic sampling of code paths

Pitfall: Dynamic compilation

- Timing of JIT is unpredictable and can yield strange results
- Code can be also decompiled or recompiled (after loading new classes, for example)
- Timing tests should run only after all code has been compiled (as this usually the case for real programs)
- Solution 1: let the program run for a long time (at least several minutes)
- Solution 2: “warm-up” runs

Pitfall: Work loads

- Often two sorts of work: accessing shared data and thread-local computation
- Their balance gives different levels of contention $\Rightarrow$ different performance/scaling
- Try to approximate the thread-local computation to get more realistic results

Pitfall: Dead code elimination

- Benchmarks are easy target for optimizers
- This might involuntarily speed up your program...
- Make sure that every computed result is somehow used by your program (in a way that does not require synchronization or substantial computation)
- Cheap trick

```
1        if (foo.x.hashCode() == System.nanoTime())
2           System.out.print(" ");
```

- Beware of static test input!

Beyond testing

- Code review
- Static analysis tools aim at detecting certain bug patterns
  - Inconsistent synchronization
  - Unreleased locks
  - Double-checked locking
  - Starting a thread from a constructor
  - sleep/notify/wait errors

Tools

- VisualVM (profiling)
- FindBugs - Find Bugs in Java Programs
- ConTest - A Tool for Testing Multi-threaded Java Applications