William Pugh, Nathaniel Ayewah
University of Maryland

Unit Testing Concurrent Software

MultithreadedTC

Deterministic and repeatable unit tests for concurrent abstractions

http://www.cs.umd.edu/projects/PL/multithreadedtc/

The MultithreadedTC framework was created to make it easier to test small concurrent abstractions. It enables test designers to validate each interleaving of two or more threads separately, even in the presence of blocking and timing issues. It also detects deadlock situations.

FEATURES

Simple Clock MultithreadedTC provides a clock which test designers can use to regulate the activities in multiple threads. *But the clock is not a timer.* It advances to the next requested tick when all threads are blocked.

Just Java Each test case is a Java class. Threads are specified using thread methods which return void, have no arguments and have names prefixed with the word "thread". Java 1.4 compatible version available.

Integrates with JUnit Run the test framework from a JUnit test. Use JUnit assertions to verify the current clock tick. Exceptions thrown in threads will cause entire test to fail.

Concise Code MultithreadedTC eliminates much of the scaffolding code needed when writing concurrent tests, e.g. setting up and tearing down threads, joining threads, using Thread.sleep()

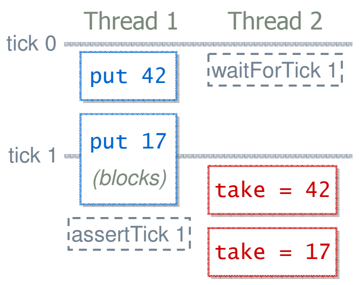
Lines of Code

	Original	MTC
TCK tests for JSR166	8003	7070
ConAn Script	2800	1119
ConAn Java Driver	7356	

Constructs removed from TCK Tests

Anonymous inner classes	257
Thread's join() method	239
try-catch blocks	106
Thread's sleep() method	313

EXAMPLE



Let us validate some properties of a bounded blocking buffer with a capacity of 1 element. We want to ensure that:

(a) The assertion `take = 42` occurs after the call to `put 17`
(b) The call to `put 17` blocks thread 1

Solution 1: Use Thread.sleep() to delay the first statement in thread 2. This introduces unnecessary timing dependence (test does not work well in a debugger or with an ill-timed garbage collector).

Solution 2: Use a latch to coordinate activities in both threads. This will not work because the call to `put 17` blocks thread 1 before the latch can be released.

Solution 3: Use MultithreadedTC!

```
class BoundedBufferTest extends MultithreadedTestCase {
   BoundedBuffer buf;
   void initialize() { buf = new BoundedBuffer(1); }
   public void thread1() {
      buf.put(42);
      buf.put(17);
      assertTick(1);
   }
   public void thread2() {
      waitForTick(1);
      assertTrue(buf.take() == 42);
      assertTrue(buf.take() == 17);
   }
   void finish() { assertTrue(buf.isEmpty()); }
}
```

all tests extend base class

run simultaneously in different threads

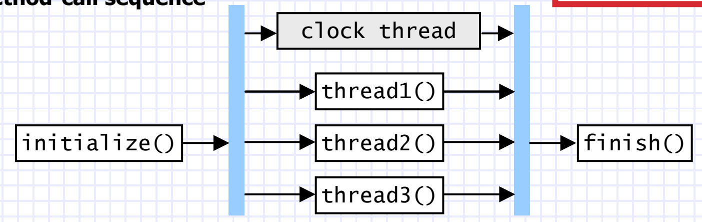
verify unblocking does not occur until tick 1

waits until all threads are blocked

JUnit Test

```
public void testBoundedBuffer()
  throws Throwable {
    TestFramework.runOnce(
      new BoundedBufferTest()
    );
}
```

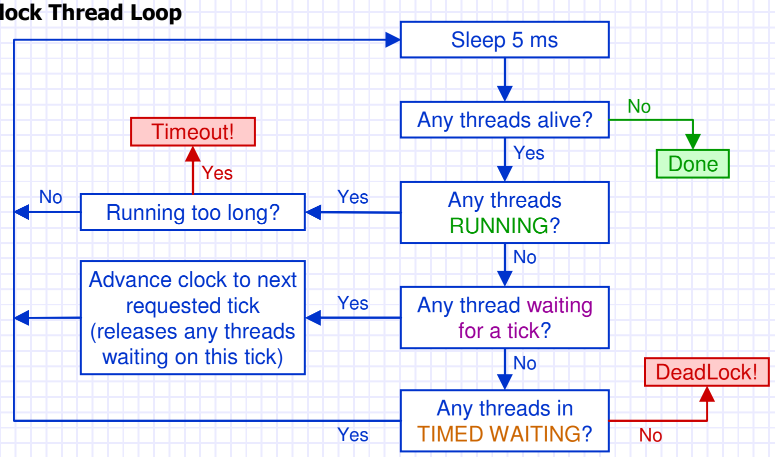
Method-call sequence



Thread Status (Example)

Thread	Status
thread1	RUNNING
thread2	BLOCKED
thread3	TIMED WAITING
thread4	Waiting for Tick 2

Clock Thread Loop


Sleep 5 ms
Any threads alive?
No
Done
Yes
Timeout!
Yes
No
Running too long?
Yes
Any threads RUNNING?
No
Advance clock to next requested tick (releases any threads waiting on this tick)
Yes
Any thread waiting for a tick?
No
DeadLock!
Any threads in TIMED WAITING?
Yes
No

USEFUL METHODS

- `waitForTick(tick)`: cause the host thread to block until the clock reaches `tick`
- `assertTick(tick)`: compare `tick` with the current clock and throw an assertion error if they do not match
- `getThread(threadID)`: returns a reference the the Thread object corresponding to `threadID`
- `freezeClock()`: prevent the clock from advancing until `unfreezeClock()` is called. Useful when doing tests that involve timed waiting and you don't want to advance the clock during the wait.

RELATED WORK

P. B. Hansen, Reproducible testing of monitors, *Software: Practice and Experience,* 1978. Early work on regulating test threads with a clock.

ConAn (Concurrency Analyzer) is a script-based test framework that uses a clock to synchronize the actions in multiple threads. It uses a timer-based clock, and a custom scripting syntax to break up test into tick blocks.

ConTest is a Java testing framework that uses a deterministic replay algorithm to record and replay specific interleavings that lead to faults.

JUnit, TestNG, GroboUtils, and ConTest all provide facilities for running concurrent tests many times to *hopefully* generate a representative set of interleavings.