How should I unit test multithreaded code?

Asked 16 years, 4 months ago Modified 19 days ago Viewed 222k times

808

I have thus far avoided the nightmare that is testing multi-threaded code since it just seems like too much of a minefield. I'd like to ask how people have gone about testing code that relies on threads for successful execution, or just how people have gone about testing those kinds of issues that only show up when two threads interact in a given manner?

This seems like a really key problem for programmers today, it would be useful to pool our knowledge on this one imho.

multithreading unit-testing

Share

Improve this question

Follow

edited Dec 21, 2020 at 18:22

user2357112
278k ● 31 ● 472 ● 552

asked Aug 15, 2008 at 11:44

jkp
81.2k ● 28 ● 106 ● 104

2 I was thinking of posting a question on this exact same issue. While Will makes many of good points below, I think we can do better. I agree there is no single "approach" to dealing with this cleanly. However, "testing as best as you can" is setting the bar very low. I'll return with my findings. – Zach Burlingame Sep 24, 2008 at 19:10

In Java: The Package java.util.concurrent contains some bad known Classes, that may help to write deterministic JUnit-Tests. Have a look at - CountDownLatch - Semaphore - Exchanger – Synox Jan 20, 2010 at 11:25

Can you provide a link to your previous unit testing related question please? – Andrew Grimm May 6, 2010 at 3:55

@Andrew Grimm: stackoverflow.com/questions/11060/… – jkp May 6, 2010 at 6:58

20 I think it's important to note that this question is 8 years old, and application libraries have come quite a long way in the meantime. In the "modern era" (2016) multi-threaded development comes up mainly in embedded systems. But if you're working on a desktop or phone app, explore the alternatives first. Application environments like .NET now include tools to manage or greatly simplify probably 90% of the common multi-threading scenarios. (asnync/await, PLinq, IObservable, the TPL...). Multi-threaded code is hard. If you don't reinvent the wheel, you don't have to retest it. – Paul Williams May 11, 2016 at 15:06

30 Answers

Sorted by: Highest score (default)

278

Look, there's no easy way to do this. I'm working on a project that is inherently multithreaded. Events come in from the operating system and I have to process them concurrently.

The simplest way to deal with testing complex, multithreaded application code is this: If it's too complex to test, you're doing it wrong. If you have a single instance that has multiple threads acting upon it, and you can't test situations where these threads step all over each other, then your design needs to be redone. It's both as simple and as complex as this.

There are many ways to program for multithreading that avoids threads running through instances at the same time. The simplest is to make all your objects immutable. Of course, that's not usually possible. So you have to identify those places in your design where threads interact with the same instance and reduce the number of those places. By doing this, you isolate a few classes where multithreading actually occurs, reducing the overall complexity of testing your system.

But you have to realize that even by doing this, you still can't test every situation where two threads step on each other. To do that, you'd have to run two threads concurrently in the same test, then control exactly what lines they are executing at any given moment. The best you can do is simulate this situation. But this might require you to code specifically for testing, and that's at best a half step towards a true solution.

Probably the best way to test code for threading issues is through static analysis of the code. If your threaded code doesn't follow a finite set of thread safe patterns, then you might have a problem. I believe Code Analysis in VS does contain some knowledge of threading, but probably not much.

Look, as things stand currently (and probably will stand for a good time to come), the best way to test multithreaded apps is to reduce the complexity of threaded code as much as possible. Minimize areas where threads interact, test as best as possible, and use code analysis to identify danger areas.

Share Improve this answer Follow

edited Oct 27, 2020 at 6:26
Pang
10.1k ● 146 ● 85 ● 124

answered Aug 15, 2008 at 13:29
user1228

1 Code analysis is great if you deal with a language/framework that allows it. EG: Findbugs will find very simple and easy shared concurrency issues with static variables. What it can't find is singleton design patterns, it assumes all objects can be created multiple times. This plugin is woefully inadequate for frameworks like Spring. – Zombies Aug 27, 2013 at 23:38

4 there actually is a cure: active objects. drdobbs.com/parallel/prefer-using-active-objects-instead-of-n/… – Dill Jan 8, 2015 at 19:49

13 While this is good advice, I'm still left asking, "how do I test those minimal areas where multiple threads are required?" – Bryan Rayner May 27, 2016 at 20:29

16 "If its too complex to test, you're doing it wrong" - we all have to dive into legacy code that we didn't write. How does this observation help anyone exactly? – Ronna Sep 12, 2016 at 11:49

5 Static analysis is likely a good idea, but it isn't testing. This post really doesn't answer the question, which is about how to test. – Warren Dew Oct 19, 2016 at 1:25

116

It's been a while when this question was posted, but it's still not answered ...

kleolb02's answer is a good one. I'll try going into more details.

There is a way, which I practice for C# code. For unit tests you should be able to program *reproducible* tests, which is the biggest challenge in multithreaded code. So my answer aims toward forcing asynchronous code into a test harness, which works *synchronously*.

It's an idea from Gerard Meszaros's book "xUnit Test Patterns" and is called "Humble Object" (p. 695): You have to separate core logic code and anything which smells like asynchronous code from each other. This would result to a class for the core logic, which works *synchronously*.

This puts you into the position to test the core logic code in a *synchronous* way. You have absolute control over the timing of the calls you are doing on the core logic and thus can make *reproducible* tests. And this is your gain from separating core logic and asynchronous logic.

This core logic needs be wrapped around by another class, which is responsible for receiving calls to the core logic asynchronously and *delegates* these calls to the core logic. Production code will only access the core logic via that class. Because this class should only delegate calls, it's a very "dumb" class without much logic. So you can keep your unit tests for this asychronous working class at a minimum.

Anything above that (testing interaction between classes) are component tests. Also in this case, you should be able to have absolute control over timing, if you stick to the "Humble Object" pattern.

Share Improve this answer Follow

edited Feb 17, 2022 at 15:47

tevemadar
13.2k ● 3 ● 25 ● 56

answered Jan 18, 2009 at 12:30

Theo Lenndorff
4,596 ● 3 ● 29 ● 43

2 But sometimes if the threads cooperate well with each other is also something should be tested, right? Definitely I will separate the core logic from the async part after reading your answer. But I am still gonna test the logic via async interfaces

with a work-on-all-threads-have-been-done callback.
– CopperCash Apr 6, 2015 at 7:17

2 This seems to be great for single thread programs and algorithms that have some form of concurrency but don't really interract with each other. I don't think it will work well testing a truely parrallel algoritm. – Nicolas Bousquet Jul 25, 2020 at 18:24

2 This kind of testing cannot help your possible deadlock problems either, where multiple threads acquire multiple locks in different order and end up waiting for each other to release a lock before continuing. The generic solution is to acquire all needed locks in identical order in all threads and that could be verified by unit tests. – Mikko Rantalainen Jul 20, 2022 at 15:38

75

Tough one indeed! In my (C++) unit tests, I've broken this down into several categories along the lines of the concurrency pattern used:

1. Unit tests for classes that operate in a single thread and aren't thread aware -- easy, test as usual.
2. Unit tests for Monitor objects (those that execute synchronized methods in the callers' thread of control) that expose a synchronized public API -- instantiate multiple mock threads that exercise the API. Construct scenarios that exercise internal conditions of the passive object. Include one longer running test that basically beats the heck out of it from multiple threads for a long period of time. This is unscientific I know but it does build confidence.

3. Unit tests for Active objects (those that encapsulate their own thread or threads of control) -- similar to #2 above with variations depending on the class design. Public API may be blocking or non-blocking, callers may obtain futures, data may arrive at queues or need to be dequeued. There are many combinations possible here; white box away. Still requires multiple mock threads to make calls to the object under test.

As an aside:

In internal developer training that I do, I teach the Pillars of Concurrency and these two patterns as the primary framework for thinking about and decomposing concurrency problems. There's obviously more advanced concepts out there but I've found that this set of basics helps keep engineers out of the soup. It also leads to code that is more unit testable, as described above.

Share Improve this answer Follow

edited Mar 8, 2021 at 15:37
metal
6,332 ● 1 ● 37 ● 49

answered Aug 15, 2008 at 13:26
David Joyner
23.1k ● 4 ● 30 ● 33

64

I have faced this issue several times in recent years when writing thread handling code for several projects. I'm providing a late answer because most of the other answers, while providing alternatives, do not actually answer the question about testing. My answer is addressed to the cases where there is no alternative to multithreaded code; I do cover code design issues for completeness, but also discuss unit testing.

Writing testable multithreaded code

The first thing to do is to separate your production thread handling code from all the code that does actual data processing. That way, the data processing can be tested as singly threaded code, and the only thing the multithreaded code does is to coordinate threads.

The second thing to remember is that bugs in multithreaded code are probabilistic; the bugs that manifest themselves least frequently are the bugs that will sneak through into production, will be difficult to reproduce even in production, and will thus cause the biggest problems. For this reason, the standard coding approach of writing the code quickly and then debugging it until it works is a bad idea for multithreaded code; it will result in code where the easy bugs are fixed and the dangerous bugs are still there.

Instead, when writing multithreaded code, you must write the code with the attitude that you are going to avoid writing the bugs in the first place. If you have properly removed the data processing code, the thread handling code should be small enough - preferably a few lines, at worst a few dozen lines - that you have a chance of writing it without writing a bug, and certainly without writing many bugs, if you understand threading, take your time, and are careful.

Writing unit tests for multithreaded code

Once the multithreaded code is written as carefully as possible, it is still worthwhile writing tests for that code. The primary purpose of the tests is not so much to test for highly timing dependent race condition bugs - it's impossible to test for such race conditions repeatably - but rather to test that your locking strategy for preventing such bugs allows for multiple threads to interact as intended.

To properly test correct locking behavior, a test must start multiple threads. To make the test repeatable, we want the interactions between the threads to happen in a predictable order. We don't want to externally synchronize the threads in the test, because that will mask bugs that could happen in production where the threads are not externally synchronized. That leaves the use of timing delays for thread synchronization, which is the technique that I have used successfully whenever I've had to write tests of multithreaded code.

If the delays are too short, then the test becomes fragile, because minor timing differences - say between different machines on which the tests may be run - may cause the timing to be off and the test to fail. What I've typically done is start with delays that cause test failures, increase the delays so that the test passes reliably on my development machine, and then double the delays beyond that so the test has a good chance of passing on other machines. This does mean that the test will take a macroscopic amount of time, though in my experience, careful test design can limit that time to no more than a dozen seconds. Since you shouldn't have very many places requiring thread coordination code in your application, that should be acceptable for your test suite.

Finally, keep track of the number of bugs caught by your test. If your test has 80% code coverage, it can be expected to catch about 80% of your bugs. If your test is well designed but finds no bugs, there's a reasonable chance that you don't have additional bugs that will only show up in production. If the test catches one or two bugs, you might still get lucky. Beyond that, and you may want to consider a careful review of or even a complete rewrite of your thread handling code, since it is likely that code still contains hidden bugs that will be very difficult to find until the code is in production, and very difficult to fix then.

Share Improve this answer Follow

edited Apr 15, 2017 at 20:19

answered Sep 11, 2015 at 21:00

Warren Dew
8,918 ● 3 ● 33 ● 44

7 Testing can only reveal the presence of bugs, not their absence. The original question asks about a 2-thread problem, in which case exhaustive testing might be possible,

but often it is not. For anything beyond the simplest scenarios you may have to bite the bullet and use formal methods - but don't skip the unit tests! Writing correct multi-threaded code is hard in the first place, but an equally hard problem is future-proofing it against regression. – Paul Williams May 11, 2016 at 13:55

5 Amazing summary of one of the least understood ways. Your answer is bang on the real segregation that ppl generally overlook. – prash Dec 15, 2016 at 9:24

1 A dozen seconds is quite a long time, even if you only have a few hundred tests of that length... – Toby Speight Jun 20, 2018 at 15:41

2 @TobySpeight The tests are long compared to normal unit tests. I've found that half a dozen tests are more than sufficient if the threaded code is properly designed to be as simple as possible, though - needing a few hundred multithreading tests would almost certainly indicate an overly complex threading arrangement. – Warren Dew Jun 21, 2018 at 17:33

2 That's a good argument for keeping your thread logic as separable from the functionality as you can (I know, much easier said than done). And, if possible, breaking the test suite into "every-change" and "pre-commit" sets (so your minute-to-minute tests aren't impacted too much). – Toby Speight Jun 22, 2018 at 7:11

27

I also had serious problems testing multi- threaded code. Then I found a really cool solution in "xUnit Test Patterns" by Gerard Meszaros. The pattern he describes is called **Humble object**.

Basically it describes how you can extract the logic into a separate, easy-to-test component that is decoupled from its environment. After you tested this logic, you can test the complicated behaviour (multi- threading, asynchronous execution, etc...)

Share Improve this answer Follow

edited Aug 20, 2008 at 12:45

answered Aug 20, 2008 at 12:29

ollifant
8,606 ● 10 ● 37 ● 45

19

There are a few tools around that are quite good. Here is a summary of some of the Java ones.

Some good static analysis tools include FindBugs (gives some useful hints), JLint, Java Pathfinder (JPF & JPF2), and Bogor.

MultithreadedTC is quite a good dynamic analysis tool (integrated into JUnit) where you have to set up your own test cases.

ConTest from IBM Research is interesting. It instruments your code by inserting all kinds of thread modifying behaviours (e.g. sleep & yield) to try to uncover bugs randomly.

SPIN is a really cool tool for modelling your Java (and other) components, but you need to have some useful framework. It is hard to use as is, but extremely powerful if you know how to use it. Quite a few tools use SPIN underneath the hood.

MultithreadedTC is probably the most mainstream, but some of the static analysis tools listed above are definitely worth looking at.

Share Improve this answer Follow

answered Jul 8, 2010 at 12:23

xagyg
9,691 ● 2 ● 33 ● 31

19

Awaitility can also be useful to help you write deterministic unit tests. It allows you to wait until some state somewhere in your system is updated. For example:

```
await().untilCall( to(myService).myMethod(),
greaterThan(3) );
```

or

```
await().atMost(5,SECONDS).until(fieldIn(myObject).of
equalTo(1));
```

It also has Scala and Groovy support.

```
await until { something() > 4 } // Scala example
```

Share Improve this answer

edited Oct 20, 2021 at 3:54

Follow

Pang
10.1k ● 146 ● 85 ● 124

answered Nov 19, 2010 at 17:26

Johan
40.4k ● 41 ● 166 ● 273

17

Another way to (kinda) test threaded code, and very complex systems in general is through Fuzz Testing. It's not great, and it won't find everything, but its likely to be useful and its simple to do.

Quote:

> Fuzz testing or fuzzing is a software testing technique that provides random data("fuzz") to the inputs of a program. If the program fails (for example, by crashing, or by failing built-in code assertions), the defects can be noted. The great advantage of fuzz testing is that the test design is extremely simple, and free of preconceptions about system behavior.
>
> ...
>
> Fuzz testing is often used in large software development projects that employ black box testing. These projects usually have a budget to develop test tools, and fuzz testing is one of the techniques which offers a high benefit to cost ratio.

> ...
>
> However, fuzz testing is not a substitute for exhaustive testing or formal methods: it can only provide a random sample of the system's behavior, and in many cases passing a fuzz test may only demonstrate that a piece of software handles exceptions without crashing, rather than behaving correctly. Thus, fuzz testing can only be regarded as a bug-finding tool rather than an assurance of quality.

Share Improve this answer

Follow

edited Dec 4, 2014 at 0:04

ThomasH
840 ● 1 ● 8 ● 23

answered Sep 24, 2008 at 5:16

Robert Gould
69.7k ● 61 ● 191 ● 275

14

Testing MT code for correctness is, as already stated, quite a hard problem. In the end it boils down to ensuring that there are no incorrectly synchronised data races in your code. The problem with this is that there are infinitely many possibilities of thread execution (interleavings) over which you do not have much control (be sure to read this article, though). In simple scenarios it might be possible to actually prove correctness by reasoning but this is usually not the case. Especially if you want to avoid/minimize synchronization and not go for the most obvious/easiest synchronization option.

An approach that I follow is to write highly concurrent test code in order to make potentially undetected data races likely to occur. And then I run those tests for some time :) I once stumbled upon a talk where some computer scientist where showing off a tool that kind of does this (randomly devising test from specs and then running them wildly, concurrently, checking for the defined invariants to be broken).

By the way, I think this aspect of testing MT code has not been mentioned here: identify invariants of the code that you can check for randomly. Unfortunately, finding those invariants is quite a hard problem, too. Also they might not hold all the time during execution, so you have to find/enforce executions points where you can expect them to be true. Bringing the code execution to such a state is also a hard problem (and might itself incur concurrency issues. Whew, it's damn hard!

Some interesting links to read:

- Deterministic interleaving: A framework that allows to force certain thread interleavings and then check for invariants
- jMock Blitzer : Stress test synchronization
- assertConcurrent : JUnit version of stress testing synronization

- Testing concurrent code : Short overview of the two primary methods of brute force (stress test) or deterministic (going for the invariants)

Share Improve this answer Follow

edited Sep 18, 2014 at 12:31

answered Sep 18, 2014 at 10:37

bennidi

2,102 ● 21 ● 28

author refers to randomizaion in testing. It might be QuickCheck, that's been ported to many languages. You can watch talk on such testing for concurrent system here – Max Aug 24, 2015 at 11:11

13

I've done a lot of this, and yes it sucks.

Some tips:

- GroboUtils for running multiple test threads
- alphaWorks ConTest to instrument classes to cause interleavings to vary between iterations
- Create a `throwable` field and check it in `tearDown` (see Listing 1). If you catch a bad exception in another thread, just assign it to throwable.
- I created the utils class in Listing 2 and have found it invaluable, especially waitForVerify and

waitForCondition, which will greatly increase the performance of your tests.

- Make good use of `AtomicBoolean` in your tests. It is thread safe, and you'll often need a final reference type to store values from callback classes and suchlike. See example in Listing 3.
- Make sure to always give your test a timeout (e.g., `@Test(timeout=60*1000)`), as concurrency tests can sometimes hang forever when they're broken.

Listing 1:

```
@After
public void tearDown() {
    if ( throwable != null )
        throw throwable;
}
```

Listing 2:

```
import static org.junit.Assert.fail;
import java.io.File;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;
import java.util.Random;
import org.apache.commons.collections.Closure;
import org.apache.commons.collections.Predicate;
import org.apache.commons.lang.time.StopWatch;
import org.easymock.EasyMock;
import org.easymock.classextension.internal.ClassExten
import static org.easymock.classextension.EasyMock.*;

import ca.digitalrapids.io.DRFileUtils;

/**
 * Various utilities for testing
```

```
  */
public abstract class DRTestUtils
{
    static private Random random = new Random();

/** Calls {@link #waitForCondition(Integer, Integer, P
 * default max wait and check period values.
 */
static public void waitForCondition(Predicate predicat
    throws Throwable
{
    waitForCondition(null, null, predicate, errorMessa
}

/** Blocks until a condition is true, throwing an {@li
 * it does not become true during a given max time.
 * @param maxWait_ms max time to wait for true conditi
 * to 30 * 1000 ms (30 seconds).
 * @param checkPeriod_ms period at which to try the co
defaults
 * to 100 ms.
 * @param predicate the condition
 * @param errorMessage message use in the {@link Asser
 * @throws Throwable on {@link AssertionError} or any
 */
static public void waitForCondition(Integer maxWait_ms
    Predicate predicate, String errorMessage) throws T
{
    waitForCondition(maxWait_ms, checkPeriod_ms, predi
        public void execute(Object errorMessage)
        {
            fail((String)errorMessage);
        }
    }, errorMessage);
}

/** Blocks until a condition is true, running a closur
 * it does not become true during a given max time.
 * @param maxWait_ms max time to wait for true conditi
 * to 30 * 1000 ms (30 seconds).
 * @param checkPeriod_ms period at which to try the co
defaults
 * to 100 ms.
 * @param predicate the condition
 * @param closure closure to run
 * @param argument argument for closure
 * @throws Throwable on {@link AssertionError} or any
 */
static public void waitForCondition(Integer maxWait_ms
    Predicate predicate, Closure closure, Object argum
{
    if ( maxWait_ms == null )
        maxWait_ms = 30 * 1000;
    if ( checkPeriod_ms == null )
        checkPeriod_ms = 100;
    StopWatch stopWatch = new StopWatch();
    stopWatch.start();
    while ( !predicate.evaluate(null) ) {
        Thread.sleep(checkPeriod_ms);
        if ( stopWatch.getTime() > maxWait_ms ) {
            closure.execute(argument);
        }
    }
}

/** Calls {@link #waitForVerify(Integer, Object)} with
 * for {@code maxWait_ms}
 */
static public void waitForVerify(Object easyMockProxy)
    throws Throwable
{
    waitForVerify(null, easyMockProxy);
}

/** Repeatedly calls {@link EasyMock#verify(Object[])}
 * max wait time has elapsed.
 * @param maxWait_ms Max wait time. <code>null</code>
 * @param easyMockProxy Proxy to call verify on
 * @throws Throwable
 */
static public void waitForVerify(Integer maxWait_ms, O
    throws Throwable
{
    if ( maxWait_ms == null )
        maxWait_ms = 30 * 1000;
    StopWatch stopWatch = new StopWatch();
    stopWatch.start();
    for(;;) {
        try
        {
            verify(easyMockProxy);
            break;
        }
        catch (AssertionError e)
        {
            if ( stopWatch.getTime() > maxWait_ms )
                throw e;
            Thread.sleep(100);
        }
    }
}

/** Returns a path to a directory in the temp dir with
 * class. This is useful for temporary test files.
 * @param aClass test class for which to create dir
 * @return the path
 */
static public String getTestDirPathForTestClass(Object
{

    String filename = object instanceof Class ?
        ((Class)object).getName() :
        object.getClass().getName();
    return DRFileUtils.getTempDir() + File.separator +
        filename;
}

static public byte[] createRandomByteArray(int bytesLe
{
    byte[] sourceBytes = new byte[bytesLength];
    random.nextBytes(sourceBytes);
    return sourceBytes;
}

/** Returns <code>true</code> if the given object is a
 */
static public boolean isEasyMockMock(Object object) {
    try {
        InvocationHandler invocationHandler = Proxy
                .getInvocationHandler(object);
        return invocationHandler.getClass().getName().
    } catch (IllegalArgumentException e) {
            return false;
        }
    }
}
```

Listing 3:

```
@Test
public void testSomething() {
    final AtomicBoolean called = new AtomicBoolean(fal
    subject.setCallback(new SomeCallback() {
        public void callback(Object arg) {
            // check arg here
            called.set(true);
        }
    });
    subject.run();
    assertTrue(called.get());
}
```

Share Improve this answer

Follow

edited Dec 23, 2019 at 1:15

ron190
1,102 ● 1 ● 20 ● 30

answered Sep 24, 2008 at 4:58

Kevin Wong
14.9k ● 11 ● 44 ● 52

2 A timeout is a good idea, but if a test times out, any later results in that run are suspect. The timed out test may still have some threads running that can mess you up.
– Don Kirkby Oct 20, 2010 at 5:59

I handle unit tests of threaded components the same way I handle any unit test, that is, with inversion of control and

7

isolation frameworks. I develop in the .Net-arena and, out of the box, the threading (among other things) is very hard (I'd say nearly impossible) to fully isolate.

Therefore, I've written wrappers that looks something like this (simplified):

```
public interface IThread
{
    void Start();
    ...
}

public class ThreadWrapper : IThread
{
    private readonly Thread _thread;

    public ThreadWrapper(ThreadStart threadStart)
    {
        _thread = new Thread(threadStart);
    }

    public Start()
    {
        _thread.Start();
    }
}

public interface IThreadingManager
{
    IThread CreateThread(ThreadStart threadStart);
}

public class ThreadingManager : IThreadingManager
{
    public IThread CreateThread(ThreadStart threadStar
    {
         return new ThreadWrapper(threadStart)
    }
}
```

From there, I can easily inject the IThreadingManager into my components and use my isolation framework of choice to make the thread behave as I expect during the test.

That has so far worked great for me, and I use the same approach for the thread pool, things in System.Environment, Sleep etc. etc.

Share Improve this answer Follow

edited Oct 27, 2020 at 0:52
Pang
10.1k ● 146 ● 85 ● 124

answered Feb 26, 2010 at 23:38
scim
203 ● 3 ● 8

+1. Its a shame that dotnet *still* has such poor support for this approach. Having to write wrappers for mundane things like Task.Delay – Yarek T Jan 8, 2021 at 13:20

6

Pete Goodliffe has a series on the unit testing of threaded code.

It's hard. I take the easier way out and try to keep the threading code abstracted from the actual test. Pete does mention that the way I do it is wrong but I've either got the separation right or I've just been lucky.

Share Improve this answer Follow

answered Aug 15, 2008 at 20:59

graham.reeds
16.5k ● 17 ● 75 ● 138

7 I read the two articles published so far, and I didn't find them very helpful. He just talks about the difficulties without giving much concrete advice. Maybe future articles will improve.
– Don Kirkby Sep 18, 2008 at 20:19

6

For Java, check out chapter 12 of [JCIP](). There are some concrete examples of writing deterministic, multi-threaded unit tests to at least test the correctness and invariants of concurrent code.

"Proving" thread-safety with unit tests is much dicier. My belief is that this is better served by automated integration testing on a variety of platforms/configurations.

Share Improve this answer Follow

answered Sep 17, 2008 at 16:35
Scott Bale
10.8k ● 5 ● 35 ● 36

6

Have a look at my related answer at

[Designing a Test class for a custom Barrier]()

It's biased towards Java but has a reasonable summary of the options.

In summary though (IMO) its not the use of some fancy framework that will ensure correctness but how you go about designing you multithreaded code. Splitting the concerns (concurrency and functionality) goes a huge way towards raising confidence. Growing Object Orientated Software Guided By Tests explains some options better than I can.

Static analysis and formal methods (see, Concurrency: State Models and Java Programs) is an option but I've found them to be of limited use in commercial development.

Don't forget that any load/soak style tests are rarely guaranteed to highlight problems.

Good luck!

Share Improve this answer Follow

edited May 23, 2017 at 12:10
Community Bot
1 ● 1

answered Jan 6, 2011 at 13:56
Toby
9,763 ● 8 ● 39 ● 60

You should also mention your `tempus-fugit` library here, which `helps write and test concurrent code` ;)
– Volo Sep 17, 2013 at 9:13

6

I like to write two or more test methods to execute on parallel threads, and each of them make calls into the object under test. I've been using Sleep() calls to coordinate the order of the calls from the different threads, but that's not really reliable. It's also a lot slower because you have to sleep long enough that the timing usually works.

I found the Multithreaded TC Java library from the same group that wrote FindBugs. It lets you specify the order of events without using Sleep(), and it's reliable. I haven't tried it yet.

The biggest limitation to this approach is that it only lets you test the scenarios you suspect will cause trouble. As others have said, you really need to isolate your multithreaded code into a small number of simple classes to have any hope of thoroughly testing them.

Once you've carefully tested the scenarios you expect to cause trouble, an unscientific test that throws a bunch of simultaneous requests at the class for a while is a good way to look for unexpected trouble.

Update: I've played a bit with the Multithreaded TC Java library, and it works well. I've also ported some of its features to a .NET version I call TickingTest.

Share Improve this answer Follow

edited Feb 22, 2017 at 20:24

answered Sep 8, 2008 at 5:34

Don Kirkby
56.5k ● 27 ● 219 ● 301

4

I just recently discovered (for Java) a tool called Threadsafe. It is a static analysis tool much like findbugs but specifically to spot multi-threading issues. It is not a replacement for testing but I can recommend it as part of writing reliable multi-threaded Java.

It even catches some very subtle potential issues around things like class subsumption, accessing unsafe objects through concurrent classes and spotting missing volatile modifiers when using the double checked locking paradigm.

If you write multithreaded Java give it a shot.

Share Improve this answer Follow

answered May 24, 2014 at 22:09

feldoh
658 ● 2 ● 6 ● 19

3

The following article suggests 2 solutions. Wrapping a semaphore (CountDownLatch) and adds functionality like externalize data from internal thread. Another way of achieving this purpose is to use Thread Pool (see Points of Interest).

Sprinkler - Advanced synchronization object

Share Improve this answer Follow

edited Sep 11, 2013 at 16:12

answered Jul 31, 2013 at 14:40

Effi Bar-She'an
792 ● 7 ● 18

3 Please explain the approaches here, external links might be dead in the future. – Uooo Aug 1, 2013 at 8:10

3

I have had the unfortunate task of testing threaded code and they are definitely the hardest tests I have ever written.

When writing my tests, I used a combination of delegates and events. Basically it is all about using `PropertyNotifyChanged` events with a `WaitCallback` or some kind of `ConditionalWaiter` that polls.

I am not sure if this was the best approach, but it has worked out for me.

Share Improve this answer
Follow

edited Jun 19, 2016 at 22:22
erip
16.9k ● 11 ● 72 ● 128

answered Aug 15, 2008 at 13:15
Dale Ragan
18.3k ● 3 ● 55 ● 71

2

I spent most of last week at a university library studying debugging of concurrent code. The central problem is concurrent code is non-deterministic. Typically, academic debugging has fallen into one of three camps here:

1. Event-trace/replay. This requires an event monitor and then reviewing the events that were sent. In a UT framework, this would involve manually sending the events as part of a test, and then doing post-mortem reviews.
2. Scriptable. This is where you interact with the running code with a set of triggers. "On x > foo, baz()". This could be interpreted into a UT framework where you have a run-time system triggering a given test on a certain condition.
3. Interactive. This obviously won't work in an automatic testing situation. ;)

Now, as above commentators have noticed, you can design your concurrent system into a more deterministic state. However, if you don't do that properly, you're just back to designing a sequential system again.

My suggestion would be to focus on having a very strict design protocol about what gets threaded and what doesn't get threaded. If you constrain your interface so that there is minimal dependancies between elements, it is much easier.

Good luck, and keep working on the problem.

Share Improve this answer Follow

answered Mar 22, 2009 at 22:45

Paul Nathan
40.2k ● 30 ● 120 ● 215

1

Assuming under "multi-threaded" code was meant something that is

- stateful and mutable
- AND accessed/modified by multiple threads concurrently

In other words we are talking about testing **custom stateful thread-safe class/method/unit** - which should be a very rare beast nowadays.

Because this beast is rare, first of all we need to make sure that there are all valid excuses to write it.

Step 1. Consider modifying state in same synchronization context.

Today it is easy to write compose-able concurrent and asynchronous code where IO or other slow operations offloaded to background but shared state is updated and queried in one synchronization context. e.g. async/await tasks and Rx in .NET etc. - they are all testable by design, "real" Tasks and schedulers can be substituted to make testing deterministic (however this is out of scope of the question).

It may sound very constrained but this approach works surprisingly well. It is possible to write whole apps in this style without need to make any state thread-safe (I do).

Step 2. If manipulating of shared state on single synchronization context is absolutely not possible.

Make sure the wheel is not being reinvented / there's definitely no standard alternative that can be adapted for the job. It should be likely that code is very cohesive and contained within one unit e.g. with a good chance it is a special case of some standard thread-safe data structure like hash map or collection or whatever.

Note: if code is large / spans across multiple classes AND needs multi-thread state manipulation then there's a very high chance that design is not good, reconsider Step 1

Step 3. If this step is reached then we need to test **our own custom stateful thread-safe class/method/unit**.

I'll be dead honest : I never had to write proper tests for such code. Most of the time I get away at Step 1, sometimes at Step 2. Last time I had to write custom thread-safe code was so many years ago that it was before I adopted unit testing / probably I wouldn't have to write it with the current knowledge anyway.

If I really had to test such code (**finally, actual answer**) then I would try couple of things below

1. Non-deterministic stress testing. e.g. run 100 threads simultaneously and check that end result is consistent. This is more typical for higher level / integration testing of multiple users scenarios but also can be used at the unit level.

2. Expose some test 'hooks' where test can inject some code to help make deterministic scenarios where one thread must perform operation before the other. As ugly as it is, I can't think of anything better.
3. Delay-driven testing to make threads run and perform operations in particular order. Strictly speaking such tests are non-deterministic too (there's a chance of system freeze / stop-the-world GC collection which can distort otherwise orchestrated delays), also it is ugly but allows to avoid hooks.

Share Improve this answer Follow

edited Mar 21, 2019 at 9:24

answered Mar 21, 2019 at 9:10

KolA
747 ● 7 ● 16

1

Running multiple threads is not difficult; it is piece of cake. Unfortunately, threads usually need to communicate with each other; that's what's difficult.

The mechanism that was originally invented to allow communication between modules was function calls; when module A wants to communicate with module B, it just invokes a function in module B. Unfortunately, this does not work with threads, because when you call a function, that function still runs in the current thread.

To overcome this problem, people decided to fall back to an even more primitive mechanism of communication: just declare a certain variable, and let both threads have access to that variable. In other words, allow the threads to share data. Sharing data is literally the first thing that naturally comes to mind, and it appears like a good choice because it seems very simple. I mean, how hard can it be, right? What could possibly go wrong?

Race conditions. That's what can, and will, go wrong.

When people realized their software was suffering from random, non-reproducible catastrophic failures due to race conditions, they started inventing elaborate mechanisms such as locks and compare-and-swap, aiming to protect against such things happening. These mechanisms fall under the broad category of "synchronization". Unfortunately, synchronization has two problems:

1. It is very difficult to get it right, so it is very prone to bugs.
2. It is completely untestable, because you cannot test for a race condition.

The astute reader might notice that "Very prone to bugs" and "Completely untestable" is a ***deadly combination***.

Now, the mechanisms I mentioned above were being invented and adopted by large parts of the industry before the concept of automated software testing became prevalent; So, nobody could see how deadly the problem was; they just regarded it as a difficult topic which requires guru programmers, and everyone was okay with that.

Nowadays, whatever we do, we put testing first. So, if some mechanism is untestable, then the use of that mechanism is just out of the question, period. Thus, synchronization has fallen out of grace; very few people still practice it, and they are becoming fewer and fewer every day.

Without synchronization threads cannot share data; however, the original requirement was not to share data; it was to allow threads to communicate with each other. Besides sharing data, there exist other, more elegant mechanisms for inter-thread communication.

One such mechanism is message-passing.

With message passing, there is only one place in the entire software system which utilizes synchronization: the implementation of the concurrent blocking queue collection class that we use for storing messages. (The idea is that we should be able to get at least that little part right.)

The great thing about message passing is that it does not suffer from race conditions and is fully testable.

Share Improve this answer Follow

edited Aug 13 at 9:11

answered Nov 25, 2022 at 14:39

Mike Nakis
61.8k ● 11 ● 124 ● 163

1 "The great thing about message passing is that it does not suffer from race conditions and is fully testable." This is patently untrue, though. Events are still, at their core, race conditions. The answer to the question "in what order are events read" is "no order at all". Putting a a giant lock around the language is helpful, but doesn't solve this out of order issue. – smaudet Jan 22 at 18:57

Yeah, I love using the actor model, but we definitively met race conditions in the form in multi-threaded code. We did avoid mutating the same objects concurrently, of course, as guaranteed by the ordering in the message queue, but response messages could still come in an unexpected order. Whether or not it is a bug that your code depends on ordering is another matter ;) – oligofren Dec 12 at 19:04

0

For J2E code, I've used SilkPerformer, LoadRunner and JMeter for concurrency testing of threads. They all do the same thing. Basically, they give you a relatively simple interface for administrating their version of the proxy server, required, in order to analyze the TCP/IP data stream, and simulate multiple users making simultaneous requests to your app server. The proxy server can give you the ability to do things like analyze the requests made, by presenting the whole page and URL sent to the server, as well as the response from the server, after processing the request.

You can find some bugs in insecure http mode, where you can at least analyze the form data that is being sent, and systematically alter that for each user. But the true tests are when you run in https (Secured Socket Layers). Then, you also have to contend with systematically altering the session and cookie data, which can be a little more convoluted.

The best bug I ever found, while testing concurrency, was when I discovered that the developer had relied upon Java garbage collection to close the connection request that was established at login, to the LDAP server, when logging in. This resulted in users being exposed to other users' sessions and very confusing results, when trying to analyze what happened when the server was brought to it's knees, barely able to complete one transaction, every few seconds.

In the end, you or someone will probably have to buckle down and analyze the code for blunders like the one I just mentioned. And an open discussion across departments, like the one that occurred, when we unfolded the problem described above, are most useful. But these tools are the best solution to testing multi-threaded code. JMeter is open source. SilkPerformer and LoadRunner are proprietary. If you really want to know whether your app is thread safe, that's how the big boys do it. I've done this for very large companies professionally, so I'm not guessing. I'm speaking from personal experience.

A word of caution: it does take some time to understand these tools. It will not be a matter of simply installing the software and firing up the GUI, unless you've already had some exposure to multi-threaded programming. I've tried to identify the 3 critical categories of areas to understand (forms, session and cookie data), with the hope that at least starting with understanding these topics will help you focus on quick results, as opposed to having to read through the entire documentation.

Share Improve this answer Follow

edited Sep 25, 2017 at 23:17

answered Sep 25, 2017 at 21:14

Red Rooster
329 ● 3 ● 4

0

Concurrency is a complex interplay between the memory model, hardware, caches and our code. In the case of Java at least such tests have been partly addressed mainly by jcstress. The creators of that library are known to be authors of many JVM, GC and Java concurrency features.

But even this library needs good knowledge of the Java Memory Model specification so that we know exactly what we are testing. But I think the focus of this effort is mircobenchmarks. Not huge business applications.

Share Improve this answer

edited Jun 20, 2018 at 8:15

Follow

answered Jun 20, 2018 at 8:06

Mohan Radhakrishnan
3,187 ● 5 ● 30 ● 44

0

There is an article on the topic, using Rust as the language in the example code:

https://medium.com/@polyglot_factotum/rust-concurrency-five-easy-pieces-871f1c62906a

In summary, the trick is to write your concurrent logic so that it is robust to the non-determinism involved with multiple threads of execution, using tools like channels and condvars.

Then, if that is how you've structured your "components", the easiest way to test them is by using channels to send messages to them, and then block on other channels to assert that the component sends certain expected messages.

The linked-to article is fully written using unit-tests.

Share Improve this answer Follow

edited Apr 28, 2020 at 4:56

answered Apr 28, 2020 at 4:14

gterzian
533 ● 6 ● 5

0

It's not perfect, but I wrote this helper for my tests in C#:

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

namespace Proto.Promises.Tests.Threading
{
    public class ThreadHelper
    {
        public static readonly int
multiThreadCount = Environment.ProcessorCount *
100;
        private static readonly int[] offsets =
new int[] { 0, 10, 100, 1000 };

        private readonly Stack<Task>
_executingTasks = new Stack<Task>
(multiThreadCount);
        private readonly Barrier _barrier = new
Barrier(1);
        private int _currentParticipants = 0;
        private readonly TimeSpan _timeout;

        public ThreadHelper() :
this(TimeSpan.FromSeconds(10)) { } // 10 second
timeout should be enough for most cases.

        public ThreadHelper(TimeSpan timeout)
        {
            _timeout = timeout;
        }

        /// <summary>
        /// Execute the action multiple times in
parallel threads.
        /// </summary>
        public void
```

Example usage:

```
[Test]
public void
DeferredMayBeBeResolvedAndPromiseAwaitedConcurrently
{
    Promise.Deferred deferred =
default(Promise.Deferred);
```

```
    Promise promise = default(Promise);

    int invokedCount = 0;

    var threadHelper = new ThreadHelper();

threadHelper.ExecuteParallelActionsWithOffsets(false
        // Setup
        () =>
        {
            invokedCount = 0;
            deferred = Promise.NewDeferred();
            promise = deferred.Promise;
        },
        // Teardown
        () => Assert.AreEqual(1, invokedCount),
        // Parallel Actions
        () => deferred.Resolve(),
        () => promise.Then(() => {
Interlocked.Increment(ref invokedCount);
}).Forget()
    );
}
```

Share Improve this answer

Follow

answered Jan 14, 2021 at 21:27

Tim

101 ● 10

0

It seems there is a framework from Microsoft for testing in a multithreaded manner:

https://github.com/microsoft/coyote/

From the project's GitHub description (excerpt):

Coyote is a cross-platform library and tool for testing concurrent C# code and deterministically reproducing bugs.

Using Coyote, you can easily test the concurrency and other nondeterminism in your C# code, by writing what we call a concurrency unit test. These look like your regular unit tests, but can reliably test concurrent workloads (such as actors, tasks, or concurrent requests to ASP.NET controllers). In regular unit tests, you would typically avoid concurrency due to flakiness, but with Coyote you are encouraged to embrace concurrency in your tests to find bugs. Coyote is used by many teams in Azure to test their distributed systems and services, and has found hundreds of concurrency-related bugs before deploying code in production and affecting users. In the words of an Azure service architect: Coyote found several issues early in the dev process, this sort of issues that would usually bleed through into production and become very expensive to fix later. Coyote is made with ❤️ by Microsoft Research. How it works Consider the following simple test:

```
[Fact]
public async Task TestTask()
{
  int value = 0;
  Task task = Task.Run(() =>
  {
    value = 1;
  });

  Assert.Equal(0, value);
  await task;
}
```

This test will pass most of the time because the assertion will typically execute before the task starts, but there is one schedule where the task starts fast enough to set value to 1 causing the assertion to fail. Of course, this is a very naive example and the bug is obvious, but you could imagine much more complicated race conditions that are hidden in complex execution paths. The way Coyote works, is that you first convert the above test to a concurrency unit test using the Coyote TestingEngine API:

```
using Microsoft.Coyote.SystematicTesting;

[Fact]
public async Task CoyoteTestTask()
{
  var configuration =
Configuration.Create().WithTestingIterations(10);
  var engine = TestingEngine.Create(configuration,
TestTask);
  engine.Run();
}
```

Next, you run the coyote rewrite command from the CLI (typically as a post-build task) to automatically rewrite the IL of your test and production binaries. This allows Coyote to inject hooks that take control of the concurrent execution during testing. You can then run the concurrent unit test from your favorite unit testing framework (such as xUnit). Coyote will take over and repeatedly execute the test from beginning to the end for N iterations (in the above example N was configured to 10). Under the hood, Coyote uses intelligent search strategies to explore all *kinds of execution paths that might hide a bug in each iteration. The awesome thing is that once a bug is found, Coyote gives you a trace through the engine.TestReport API that you can use to reliably reproduce the bug as many times as you want, making debugging and fixing the issue significantly easier.*

I hope that helps.

Share Improve this answer

Follow

answered Aug 22 at 8:34

holbizmetrics

101 ● 1 ● 3

0

One simple test pattern that can work for **some (not all!)** cases is to repeat the same test many times. For example, suppose you have a method:

```
def process(input):
    # Spawns several threads to do the job
    # ...
    return output
```

Create a bunch of tests:

```
process(input1) -> expect to return output1
process(input2) -> expect to return output2
...
```

Now run each of those tests many times.

If the implementation of `process` contains a subtle bug (e.g. deadlock, race condition, etc.) that has 0.1% chance to emerge, running the test 1000 times gives ~63% probability for the bug to emerge at least once. Running the test 10000 times gives >99% probability.

Share Improve this answer Follow

edited Nov 30 at 23:09

answered Aug 3, 2022 at 15:41

mercury0114
1,449 ● 2 ● 20 ● 33

Could you please share the math behind the part starting with `0.1% chance ...`? – Harry Nov 29 at 15:20

2 Assume a single run has a probability `p` to fail, that means it has a probability `1-p` to succeed. What's the probability that all `N` runs will succeed? That's `(1-p)^N`. What's the probability that at least one of `N` runs will fail? That's `1 - (1-p)^N`. Now plug in `p = 0.001` and `N=10000` into this last equation. – mercury0114 Nov 30 at 23:08

Math maestro 👌💛 – Harry Dec 1 at 3:06

-1

If you are testing simple *new Thread(runnable).run()* You can mock Thread to run the runnable sequentially

For instance, if the code of the tested object invokes a new thread like this

```
Class TestedClass {
    public void doAsychOp() {
       new Thread(new myRunnable()).start();
    }
}
```

Then mocking new Threads and run the runnable argument sequentially can help

```
@Mock
private Thread threadMock;

@Test
public void myTest() throws Exception {
    PowerMockito.mockStatic(Thread.class);
    //when new thread is created execute runnable
immediately

PowerMockito.whenNew(Thread.class).withAnyArguments(
Answer<Thread>() {
        @Override
        public Thread answer(InvocationOnMock
invocation) throws Throwable {
            // immediately run the runnable
            Runnable runnable =
invocation.getArgumentAt(0, Runnable.class);
            if(runnable != null) {
                runnable.run();
            }
            return threadMock;//return a mock so
Thread.start() will do nothing
        }
    });
    TestedClass testcls = new TestedClass()
    testcls.doAsychOp(); //will invoke
myRunnable.run in current thread
    //.... check expected
}
```

Share Improve this answer Follow

edited Dec 27, 2018 at 15:23

answered Apr 9, 2018 at 15:36

Avraham Shalev
110 ● 1 ● 3

-5

(if possible) don't use threads, use actors / active objects. Easy to test.

Share Improve this answer Follow

answered Jul 4, 2013 at 8:11

Dill
2,025 ● 4 ● 21 ● 33

2 @OMTheEternity maybe but its still the best answer imo. – Dill Mar 11, 2015 at 19:16

-7

You may use EasyMock.makeThreadSafe to make testing instance threadsafe

Share Improve this answer Follow

answered Jan 13, 2014 at 11:48

user590444
4,322 ● 9 ● 37 ● 44

1 This is not at all a possible way of testing multithreaded code. The problem is not that the test code runs multi threaded but that you test code that usually runs multi-threaded. And you can not synchronise everything away because then you

actually don't test for data races anymore. – bennidi Sep 18, 2014 at 10:19