Optimistic and Pessimistic Concurrency Control with Shared Memory Models

DRAFT-14092012

A look at modern concurrency control mechanisms in Java

Toby Weston

June 2011

Table of Contents

1 Introduction

1.1 Introduction

1.1.1 Abstract

In shared memory models, multiple concurrent processes may compete for access to common memory and so must provide ways to protect the integrity of that shared memory. The concurrency control mechanisms to achieve this can be categorised as either optimistic in nature or pessimistic (*Concurrency Control*, Anon., 2011). Languages such as Java have typically offered pessimistic approaches such as guarded memory. Guarded memory requires a lot of effort from the developer to get right, is difficult to prove correct and is often difficult to implement whilst maintaining good object oriented practices. Optimistic mechanisms, specifically Software Transactional Memory, purport to simplify the development process but as a relatively new approach has had little mainstream adoption.

As the acceleration of processor power predicted by Moore's Law reaches its peak, the utilisation of multi-core processors predicted by Amdahl's Law becomes more and more important (Moore, 1975; Amdahl 1967). With the current trend towards functional / object oriented hybrid languages and their impact on concurrent programming, it seems obvious that concurrency is set to be an even bigger part of modern software development.

Concurrent programming has always been difficult, mostly because of the shared memory model and traditional approaches guarding it. This paper aims to explore the problems, describing characteristics of concurrency control in shared memory systems, comparing optimistic and pessimistic approaches using a real world example and comment on the current state and appropriateness of technology choices.

Distributed models avoid contention as they don't actually share memory, each process works on its own local heap. Techniques such as the actor model or distributed message passing effectively simulate a distributed model and are out of scope for this discussion.

1.1.2 Goals

- Describe the shared memory model and appropriate concurrency control mechanisms.
- Present alternative implementations of a common concurrency problems; typical pessimistic, lock based synchronisation solutions, modern (non-blocking) optimistic based solutions and optimistic, software transactional memory based solution.
- Demonstrate a real-world usage examples, to help better understand the concurrency control mechanisms and provide a reference to interested readers.
- Present conclusions / experience report.

2 Shared Memory

2.1 Shared Memory Model

Sharing common memory allows us to build software that works on common data structures, it allows us to utilise modern architectures to solve common problems without having to copy common data between processes (Christopher and Thiruvathukal 2001, p.3).

2.1.1 The Java Memory Model

The part of the Java Specification (Gosling, et al. 2005) concerned with JVM implementations of shared memory is refereed to as the Java Memory Model. It basically describes how any JVM implementation should behave under certain conditions. Interestingly for us, it's particularly concerned with describing behaviour under multi-threaded conditions.

In modern systems, the order in which instructions are actually executed isn't necessarily the same order that they are arranged at source. The compiler, processors and memory subsystems may reorder execution for best performance. In fact, on multi-core platforms, the processors will likely have their own local cache which may or may not be in-sync with main memory. Without some synchronising mechanism, when the data in main memory is shared, there is no guarantee that each processor will see an up-to-date value. This turns out to be a good example of why we need the Java Memory Model. This part of the specification defines the behaviour of such synchronisation mechanisms and behaviour. For example, it defines that the `volatile` keyword should indicate to the JVM that some shared state is not eligible for caching in processor-local caches and so ensure inter-thread visibility.

Another important part of the Java Memory Model defines *as-if-serial* semantics. Here, the JVM is required to produce the same results as if serial execution were observed, regardless of the actual optimisations and re-ordering performed, at least within a single thread. The *as-if-serial* semantics however, don't prevent this guaranteed accuracy between threads and so the Java Memory Model has to prescribe alternative guarantees. These guarantees allow us to reason about concurrent program execution and underpin Java's concurrency control mechanisms. It's what enforces consistent behaviour across threads when entering or leaving a `synchronized` block for example.

It's interesting to note that like any specification, vendors are free to ignore the Java Memory Model. There are certainly JVM implementations that may not respect the volatile keyword for example.

2.1.2 Pessimistic Concurrency Control

Being pessimistic about how to control access to shared memory means assuming the worse. It assumes that access to shared memory will be contended and so above all else, access must be serialised in some way so that only one access is allowed at any given time. Java provides plenty of mechanisms to achieve this such as the `synchronized` keyword, locks and other high level mechanisms such as barriers and semaphores. They usually rely on co-operation within the code to work correctly. For example, all potential accessors must *agree* to participate in the specific control mechanism used. Failing to spot the need to participate in a given control mechanism is often the cause of correctness problems in concurrent systems.

For the purposes of this discussion, we can summarise pessimistic control as lock based. Locking usually implies blocking behaviour when waiting for a lock to become free.

2.1.3 Optimistic Concurrency Control

An optimistic approach to concurrency control on the other hand takes a more liberal view on things. How likely is it that shared memory will actually be contented really? What if we don't assume the worse but instead assume that conflicts are relatively rare? In this case we can essentially leave shared memory unguarded but provide mechanisms to spot collisions and provide failure and recovery semantics. Database systems have provided these mechanisms for some time (Kung and Robinson, 1981) with most popular ORM mapping tools including Hibernate offering implementations.

Software Transaction Memory is an optimistic alternative to lock based control to shared memory. It provides atomicity and isolation schematics similar to database transactions. Consistency is maintained by the developer just as in the pessimistic world but by providing the building blocks, consistency is supported (if not guaranteed). Durability however, can not be supported as ultimately, any successful transaction's results are stored in volatile memory the JVM can not ensure they are preserved.

2.1.4 Non-Blocking Algorithms; The Grey Area

As mentioned, locking usually implies blocking behaviour which can have a knock-on affect to performance and overall progress. However, non-blocking algorithms are available as an alternative to strict (mutually exclusive) locking when accessing shared memory. Non-blocking algorithms guarantee either per-thread progress (wait-free) or system wide progress (lock-free) (Goetz et al, 2006, p. 329; Non-blocking algorithm, Anon., 2011) and are often cited as offering better scalability than lock based equivalents (Goetz 2006, pp. 326-329, 336).

Usually, non-blocking algorithms require low level support for atomic read-modify-writes (such as *compare-and-swap* or *load-link/store-conditional*). The performance of equivalent implementations not using these primitives has traditionally been poor. More recently however, Software Transactional Memory offers a similar yet higher level abstraction when building non-blocking code whilst anecdotally offering good performance. Low level non-blocking constructs are usually used to build performant data structures (queues, stacks, hash tables etc) as found in the `java.util.concurrent` package.

As Goetz (2006, p. 321) points out, *compare-and-swap* is an optimistic technique so for the purpose of this discussion, where do the Java classes using *compare-and-swap* fit in? In terms of classifying as either pessimistic or optimistic, traditional control structures providing serial access (such as `synchronized` and `wait`/ `notify`) are certainly pessimitic. Emerging techniques such as Software Transactional Memory are clearly optimistic which just leaves the newer (post 1.5) constructs available in the `java.util.concurrent` package. Those that use *compare-and-swap* (for example, `AtomicLong`) or similar have to be classified as optimistic whereas the implementations of common concurrent blocking abstractions such as the semaphore are pessimistic. Some classes such as `Lock` implementations can even be seen as both (see Appendix A for details). However, constructs using *compare-and-swap*, although offering collision detection (`compareAndSet` returns a boolean indicating success), still require the developer to implement any recovery strategy.

2.1.5 Alternative Access Mechanisms

Distributed Memory is the idea that in multi-core processors (or in single-core multiple processor systems), each core/processor has local memory and works on it with exclusivity. If a task is required to collaborate with other core's memory, it must communicate with them as an external resource. The actor model or distributed message passing are examples.

3 Problem Definition

3.1 Example Problem

In order to better contrast the concurrency control mechanisms available, I had to come up with a concurrency problem that was meaty enough to represent real-world experiences. I ended up choosing a problem around request statistics.

3.1.1 Request Statistics

In typical client-server software, multiple clients make requests to a centralised server to achieve some business goal. This gives a good opportunity to encounter concurrency issues as multiple clients might be interested in the same kinds of things represented on the server. In our case, we're interested in recording statistics around generic requests that clients make. In a web application, this might represent the request-response cycle and in our case, we're interested in recording how long requests take to complete and respond to the client. We're interested in performance monitoring our client-server application.

Specifically then, the problem is given a new web-application, we would like to record request statistics against specific services so that we can understand typical response times and set achievable service level agreements with customers.

3.1.2 The Ping Server

The web-application that we're interested in is called *PingPong*. It is a server that responds to HTTP `GET` requests to the URL `/ping` with a HTTP message of 200 OK.

Expanding the problem description, we'd like to record the following information about requests

- Total, communicative number of `ping` requests
- Total, communicative number of failed `ping` requests (those that response with HTTP 5xx)
- Total, communicative number of successful `ping` requests
- The mean response time for `ping` requests
- Throughput of `ping` requests in requests per second
- Longest response time of `ping` requests
- The most recent response time for a `ping` request (to highlight the variance that would be smoothed by showing mean response times above)
- Total, communicative response time for all `ping` requests

In addition, we would also like to

- Reset the counters at any time
- Allow counter retrieval and reset from distributed machine
- Allow a sliding window implementation to better highlight rapid trend changes

For the purpose of this exercise, we're not interested in measuring requests statistics from the client or non-server side generated errors (errors that can not be represented by the server with HTTP 5xx error codes). For example, we're not interested in timeout of connection failures.

4 Solutions

4.1 Example Solutions

This section presents various solutions to recording statistics of our HTTP server. Essentially, I went about implementing basic performance monitoring abstractions and embedding these into the server. By implementing various versions of these components (optimistic and pessimistic), we're able to compare their characteristics. As the embedded components are associated with the request-response cycle, we can also expect them to see lots of concurrent access when the server is under heavy simulated load which is just what we're interested in.

This section includes a write up of thread safe components used to

- capture thread usage, the number of created, active and terminated threads in the system.
- thread timings, number of threads executed, total time and average time to execute.
- throughput of any request showing total number and a mean requests per second.
- Instrument contention of guarded segments as a ration of collisions against successful acquisition.

discussing the testing strategies to each and how that influenced the implementations.

4.1.1 The Software Transaction Memory Library Used

The specific Software Transaction Memory library used for this discussion is the Multiverse STM. I used some syntactic sugar available as part of Akka (akka-stm) but the core STM is Multiverse.

Multiverse version 0.6 is based around the Multi Version Concurrency Control (MVCC) idea used by popular database implementations. As it's name suggests it revolves around the idea of keeping versions of data or *snapshots* and detecting if a conflict has occurred when working with a particular version. Multiverse implements this idea using a central `AtomicLong` to increment version numbers associated with shared memory writes (Veentjer, 2011, section 10.1) and it's underlying `compareAndSet` for conflict detection. This can be seen as an implementation of the Transaction Locking II (TL2) algorithm (Dice et al, 2006). The central engine of Multiverse is called AlphaSTM.

Multiverse version 0.7 shifted away from the central clock towards the idea of a conflict counter (comparable to SkySTM by Lev et al) which offers less contention, greater scalability and may prove key in providing distributed transactional memory in the future. The implementation has various strategies which should offer improvements over vanilla SkySTM. The improved core engine pools more objects and is described by its author as faster than previous versions. Lev (2009) notes that SkySTM offers more scalable STM than previous approaches such as TL2. The central engine in 0.7 started live as BetaSTM but has since been deprecated and a newer version called GammaSTM introduced. At the time of writing, GammaSTM is the engine used in 0.7.

4.2 Source Code

All source code is available from Subversion. To checkout and recreate this document, run the following commands (*nix platforms).

```
svn checkout http://badrobot.googlecode.com/svn/trunk/bad.robot/conconc badrobot-read-only
mvn pdf:pdf
open target/pdf/concurrency-control-1.0-SNAPSHOT.pdf
```

or the following on Windows platforms

```
svn checkout http://badrobot.googlecode.com/svn/trunk/bad.robot/conconc badrobot-read-only
mvn pdf:pdf
start target\pdf\concurrency-control-1.0-SNAPSHOT.pdf
```

5 Instrumenting Thread Usage using ThreadCounter

5.1 Instrumenting Thread Usage

One area that was identified as an opportunity to explore shared access was around instrumenting thread usage within the system. The requirement being around understanding the cumulative number of threads created and the currently active threads (threads started but not yet terminated).

5.1.1 Shared Infrastructure

The ultimate goal was to create alternative implementations of something that can be used to instrument thread usage within the system, one pessimistic / lock based implementation and an alternative optimistic implementation. It makes sense if these competing implementations follow a similar approach so that they can be swapped easily for comparison.

The role of collecting or processing this information can be seen in terms of an *observer*, for example,

```
public interface ThreadObserver {
    void threadCreated();
    void threadStarted();
    void threadTerminated();
}
```

Fig 5.1. The basic observer interface

Java's `ThreadFactory` is a natural place to make observations about thread activity. All that would be required is for the application to be wired up to use the following thread factory and we can start our instrumentation.

```
public class ObservableThreadFactory implements ThreadFactory {
    private final ThreadObserver observer;
    public ObservableThreadFactory(ThreadObserver observer) {
        this.observer = observer;
    }
    @Override
    public Thread newThread(final Runnable runnable) {
        Thread thread = new Thread(new Runnable() {
            public void run() {
                try {
                     observer.threadStarted();
                    runnable.run();
                } finally {
                     observer.threadTerminated();
                }
            }
        });
         observer.threadCreated();
        return thread;
    }
}
```

Fig 5.2. Example use of the observer in a ThreadFactory

Note that the class is easily tested using mock objects.

5.1.2 Pessimistic / Lock Based Synchronisation

This section talks about the lock based implementation an observer called `ThreadCounter` and its evolution.

5.1.2.1 Basic Implementation

A naive implementation of the `ThreadCounter` class might look like this

```
@Not(ThreadSafe.class)
public class ThreadCounter implements ThreadObserver {
    private long activeThreads;
    private long createdThreads;
    @Override
    public void threadCreated() {
        createdThreads++;
    }
    @Override
    public void threadStarted() {
        activeThreads++;
    }
    @Override
    public void threadTerminated() {
        activeThreads--;
    }
    @Override
    public long getActiveCount() {
        return activeThreads;
    }
    @Override
    public long getCreatedCount() {
        return createdThreads;
    }
    @Override
    public void reset() {
        activeThreads = 0;
        createdThreads = 0;
    }
}
```

Fig 5.3. Naive implementation of ThreadCounter

The basic test below shows the implementation to be correct (at least in a non-concurrent context).

```
public class ThreadCounterTest {
    private final ThreadCounter counter = new ThreadCounter();
    @Test
    public void shouldInitialiseCounts() {
        assertThat(counter.getActiveCount(), is(0L));
        assertThat(counter.getCreatedCount(), is(0L));
    }
    @Test
    public void shouldIncrementActiveCount() {
        incrementActiveThreadsBy(3);
        assertThat(counter.getActiveCount(), is(3L));
    }
    @Test
    public void shouldDecrementActiveThreadCount() {
        incrementActiveThreadsBy(5);
        assertThat(counter.getActiveCount(), is(5L));
        decrementActiveThreadsBy(5);
        assertThat(counter.getActiveCount(), is(0L));
    }
    @Test
    public void shouldIncrementCreatedCount() {
        incrementThreadsBy(6);
        assertThat(counter.getCreatedCount(), is(6L));
    }
    @Test
    public void shouldResetCounts() {
        incrementActiveThreadsBy(8);
        incrementThreadsBy(5);
        counter.reset();
        assertThat(counter.getActiveCount(), is(0L));
        assertThat(counter.getCreatedCount(), is(0L));
    }
    private void incrementActiveThreadsBy(int amount) {
        for (int i = 0; i < amount; i++)
            counter.threadStarted();
    }
    private void decrementActiveThreadsBy(int amount)  {
        for (int i = 0; i < amount; i++)
            counter.threadTerminated();
    }
    private void incrementThreadsBy(int amount)  {
        for (int i = 0; i < amount; i++)
            counter.threadCreated();
    }
}
```

Fig 5.4. Basic single-thread behavioural unit test

5.1.3.1 Testing Thread Safety

The next test shows that it isn't correct from a concurrent context. Here, the tempus-fugit micro-library is used to run each test method repeatedly over several threads. Specifically, each of the test methods are run one hundred times (thanks to the `RepeatingRule` rule) in fifty threads (thanks to the `ConcurrentRule` rule along with the `count` variable). To kick this off for each test method

at the *same time*, the `ConcurrentTestRunner` is used. Otherwise, each of the test methods will run in their own threads, repeatedly but in sequence (ie, `notifyThreadStarted` would run, then `notifyThreadTerminated` and so on). Adding the `@RunWith` means that each test method is kicked off in its own thread at roughly the same time.

Separating a functionally correctness test from a thread-safety style test meant that the two concerns could stay separate during testing and development.

```
@RunWith( ConcurrentTestRunner.class)
public class ThreadCounterIntegrationTest {
    private static final ThreadCounter counter = new ThreadCounter();
    @Rule public  ConcurrentRule concurrent = new ConcurrentRule();
    @Rule public  RepeatingRule repeating = new RepeatingRule();
    @Test
    @Repeating
    @Concurrent(count = 50)
    public void notifyThreadStarted() {
        counter.threadStarted();
        Introduce.jitter();
    }
    @Test
    @Repeating
    @Concurrent(count = 10)
    public void notifyThreadTerminated() {
        counter.threadTerminated();
        Introduce.jitter();
    }
    @Test
    @Repeating
    @Concurrent(count = 50)
    public void notifyThreadCreated() {
        counter.threadCreated();
        Introduce.jitter();
    }
    @AfterClass
    public static void verifyCounter() {
        assertThat(counter.getCreatedCount(), is(5000L));
        assertThat(counter.getActiveCount(), is(4000L));
    }
}
```

Fig 5.5. Multi-threaded test highlighting concurrency problems

The call to `Introduce.jitter()` introduces a pseudo-random delay of up to five milliseconds. This is designed to try and avoid deterministic behaviour and exaggerate the affect of the tests.

The first assertion is expecting a created count of five thousand (having called `threadCreated` one hundred times over fifty threads). The second assertion also ensures that `threadTerminated` affects the active count (it will be run one hundred times over ten threads). The default number of repetitions from the `RepeatingRule` is one hundred.

5.1.3.2 Testing Invariants

The above test is designed to load the class under test so heavily that it is *reasonably likely* to fail the assertions (in `verifyCounter()`). It doesn't however test the invariant around the `reset` method. It could be argued that if `reset` is called, it should reset both `activeThreads` and `createdThreads`

atomically. That is to say, no additional updates should be allowed to either variable until both have been set to zero.

Testing the invariant directly proved too difficult to do, it was just too hard to simulate the race condition between resetting and setting multiple variables. However, as well see, the natural progression of the implementation lead to an alternative strategy which ensures the invariant is maintained.

5.1.3.3 Making `ThreadCounter` Thread Safe

Making the class thread safe (and passing the previous tests) was pretty straight forward using the `AtomicLong` class.

```
@ThreadSafe
public class ThreadCounter implements ThreadObserver {
    private final  AtomicLong activeThreads = new AtomicLong();
    private final  AtomicLong createdThreads = new AtomicLong();
    @Override
    public void threadCreated() {
        createdThreads.getAndIncrement();
    }
    @Override
    public void threadStarted() {
        activeThreads.getAndIncrement();
    }
    @Override
    public void threadTerminated() {
        activeThreads.getAndDecrement();
    }
    @Override
    public long getActiveCount() {
        return activeThreads.get();
    }
    @Override
    public long getCreatedCount() {
        return createdThreads.get();
    }
    @Override
    public void reset() {
        activeThreads.set(0);
        createdThreads.set(0);
    }
}
```

Fig 5.6. Thread safe version of the ThreadCounter

At this point, the class is thread safe but the invariant around the `reset` method is still not maintained (or tested). A simple fix might be to use the `synchronized` keyword on *all* of the methods (at which point, we'd no longer need to the `AtomicLong`s). This is explored in the below.

5.1.3.4 Maintaining the Invariant

An initial revision to guarding access to the state to maintain the invariant is shown below.

```
@ThreadSafe
public class ThreadCounter implements ThreadObserver {
    private final AtomicLong activeThreads = new AtomicLong();
    private final AtomicLong createdThreads = new AtomicLong();
    private final ReentrantLock lock = new ReentrantLock();
    @Override
    public void threadCreated() {
        execute(threadCreated).using(lock);
    }
    @Override
    public void threadStarted() {
        execute(threadStarted).using(lock);
    }
    @Override
    public void threadTerminated() {
        execute(threadTerminated).using(lock);
    }
    @Override
    public long getActiveCount() {
        return activeThreads.get();
    }
    @Override
    public long getCreatedCount() {
        return createdThreads.get();
    }
    @Override
    public void reset() {
        if (acquired(lock))
            execute(reset).using(lock);
    }
    private Callable<Void, RuntimeException> threadCreated = new Callable<Void, RuntimeException>() {
        @Override
        public Void call() throws RuntimeException {
            createdThreads.getAndIncrement();
            return null;
        }
    };
    private Callable<Void, RuntimeException> threadStarted = new Callable<Void, RuntimeException>() {
        @Override
        public Void call() throws RuntimeException {
            activeThreads.getAndIncrement();
            return null;
        }
    };
    private Callable<Void, RuntimeException> threadTerminated = new Callable<Void, RuntimeException>() {
        @Override
        public Void call() throws RuntimeException {
            activeThreads.getAndDecrement();
            return null;
        }
    };
    private Callable<Void, RuntimeException> reset = new Callable<Void, RuntimeException>() {
        @Override
        public Void call() throws RuntimeException {
            activeThreads.set(0);
            createdThreads.set(0);
            return null;
        }
    };
    private static Boolean acquired(final Lock lock) {
        return resetInterruptFlagWhen(new Interruptible<Boolean>() {
            @Override
```

Fig 5.7. More elaborate version ensuring consistency during reset

This revision attempts to maintain the invariant using Java `Lock`s and at the same encapsulate the use of the locks in a separate class (`ExecuteUsingLock`) to ensure consistent behaviour. The unfortunate verbosity of using `Callable` objects to achieve this is addressed later. For now, the helper class looks like this

```
public class ExecuteUsingLock<T, E extends Exception> {
    private final Callable<T, E> callable;
    private ExecuteUsingLock(Callable<T, E> callable) {
        this.callable = callable;
    }
    public static <T, E extends Exception> ExecuteUsingLock<T, E> execute(Callable<T, E> callable) {
        return new ExecuteUsingLock<T, E>(callable);
    }
    public T using(Lock lock) throws E {
        try {
            lock.lock();
            return callable.call();
        } finally {
            lock.unlock();
        }
    }
}
```

Fig 5.8. Execute Callables ensuring lock and unlock semantics

By using the same lock to guard all the write methods, we're effectively implementing a class equivilant to one that synchronises on all the write methods. It becomes more serial than previous revisions (ie, you can't call `threadStarted` at the same time as `threadTerminated`). It's up to you, to decide if that's a big deal or not.

The `reset` method has been implemented to try and acquire the lock before actually executing the reset functionality. This is an attempt to optimise the reset and isn't really necessary unless you've tested and identified it as a bottleneck. It's here really as part of the academic exercise.

As the same lock is used when writing (including the reset), there's no need to lock on the read as the underlying `AtomicLong` will ensure visibility of any successful writes. As discussed, we could avoid the use of locks completely by synchronising all the methods and if we dropped the `AtomicLong`s in favour of `long`s, we could make the variables `volatile` to ensure visibility. These alternatives are roughly equivalent but by exposing the lock in this revision we can create a test using mock objects that separates the synchronisation policy from the functionality of the class.

5.1.3.5 Tidying Up

A quick tidy up saw me push the anonymous `Callable` objects into their own classes and reduce the noice in the `ThreadCounter`.

```
@ThreadSafe
public class ThreadCounter implements ThreadObserver {
    private final AtomicLong activeThreads = new AtomicLong();
    private final AtomicLong createdThreads = new AtomicLong();
    private final ReentrantLock lock = new ReentrantLock();
    @Override
    public void threadCreated() {
        execute(increment(createdThreads)).using(lock);
    }
    @Override
    public void threadStarted() {
        execute(increment(activeThreads)).using(lock);
    }
    @Override
    public void threadTerminated() {
        execute(decrement(activeThreads)).using(lock);
    }
    @Override
    public long getActiveCount() {
        return activeThreads.get();
    }
    @Override
    public long getCreatedCount() {
        return createdThreads.get();
    }
    @Override
    public void reset() {
        if (acquired(lock))
            execute(resetOf(activeThreads, createdThreads)).using(lock);
    }
}
```

Fig 5.9. Tidied version of the ThreadCounter

```
public class Increment implements Callable<Void, RuntimeException> {
    private final AtomicLong counter;
    public static Increment increment(AtomicLong counter) {
        return new Increment(counter);
    }
    private Increment(AtomicLong counter) {
        this.counter = counter;
    }
    @Override
    public Void call() throws RuntimeException {
        counter.getAndIncrement();
        return null;
    }
}
```

Fig 5.10. Pushing the incrementing Callable into its own class

```
public class AcquireLock {
    public static Boolean acquired(final Lock lock) {
        return resetInterruptFlagWhen(new Interruptible<Boolean>() {
            @Override
            public Boolean call() throws InterruptedException {
                return lock.tryLock(10, MILLISECONDS);
            }
        });
    }
}
```

Fig 5.11. Pushing tryLock semantics in its own class

```
public class Reset implements Callable<Void, RuntimeException> {
    private final List<AtomicLong> counters;
    public static Reset resetOf(AtomicLong... counters) {
        return new Reset(counters);
    }
    private Reset(AtomicLong... counters) {
        this.counters = asList(counters);
    }
    @Override
    public Void call() throws RuntimeException {
        for (AtomicLong counter : counters)
            counter.set(0);
        return null;
    }
}
```

Fig 5.12. Pushing the reset Callable into its own class

5.1.3.6 Building out the `Guard` Interface

There are some limitations with the current implementation, notably the inability to test the invariant. We also know that we ultimately want to create a counter that isn't limited to a pessimistic locking strategy. The idea of *guarding* shared memory seems to be abstract enough to imply non-lock based solutions, so I created a basic `Gaurd` interface.

```
public interface Guard {
    <R, E extends Exception> R execute(Callable<R, E> callable) throws E;
    Boolean guarding();
}
```

Fig 5.13. The Guard class

The basic lock based implementation of which is shown below

```
public class LockingGuard implements Guard {
    private final Lock lock;
    public LockingGuard(Lock lock) {
        this.lock = lock;
    }
    @Override
    public <R, E extends Exception> R execute(Callable<R, E> callable) throws E {
        try {
            lock.lock();
            return callable.call();
        } finally {
            lock.unlock();
        }
    }
    @Override
    public Boolean guarding() {
        return acquired(lock);
    }
}
```

Fig 5.14. The Lock based guard

This opens several opportunities for the improving the current implementation, specifically around testing individual components in isolation and finally creating a test to ensure the invariant in maintained. The first step is to refactor the `ThreadCounter` to use the `Guard`.

```
@ThreadSafe
public class ThreadCounter implements ThreadObserver {
    private final AtomicLong activeThreads = new AtomicLong();
    private final AtomicLong createdThreads = new AtomicLong();
    private final Guard guard;
    public ThreadCounter(Guard guard) {
        this.guard = guard;
    }
    @Override
    public void threadCreated() {
        guard.execute(increment(createdThreads));
    }
    @Override
    public void threadStarted() {
        guard.execute(increment(activeThreads));
    }
    @Override
    public void threadTerminated() {
        guard.execute(decrement(activeThreads));
    }
    @Override
    public long getActiveCount() {
        return activeThreads.get();
    }
    @Override
    public long getCreatedCount() {
        return createdThreads.get();
    }
    @Override
    public void reset() {
        if (guard.guarding())
            guard.execute(resetOf(activeThreads, createdThreads));
    }
}
```

and in order to test the original `ThreadCounterTest` (which shouldn't be concerned with thread safety), a dummy `Guard` is implemented. Notice this just delegates to the `Callable` object.

```
public class Unguarded implements Guard {
    public static Guard unguarded() {
        return new Unguarded();
    }
    @Override
    public <R, E extends Exception> R execute(Callable<R, E> callable) throws E {
        return callable.call();
    }
    @Override
    public Boolean guarding() {
        return true;
    }
}
```

This allows the original test to be unaffected

```
public class ThreadCounterTest {

    private final ThreadCounter counter = new ThreadCounter(unguarded());
    ...
}
```

We can also change tact when it comes to testing the invariant. Rather than try and reproduce the race condition, it's sufficient to ensure that the same `Guard` is used for write and reset methods. Assuming the runtime guard implementation has been tested, a test to ensure the same guard is used for all of the methods ensures the invariant will be maintained. This is very implementation specific but given the race condition proved too difficult to reproduce, it's a sensible compromise.

```
@RunWith(JMock.class)
public class ThreadCounterInvariantTest {
    private final Mockery context = new Mockery();
    private final Guard guard = context.mock(Guard.class);
    private final ThreadCounter counter = new ThreadCounter(guard, new AtomicLongCounter(), new AtomicLon
    @Test
    public void shouldMakeGuardedCallOnWrites() throws Exception {
        context.checking(new Expectations() {{
            exactly(3).of(guard).execute(with(any(Callable.class)));
        }});
        counter.threadCreated();
        counter.threadStarted();
        counter.threadTerminated();
    }
    @Test
    public void shouldMakeGuardedCallForResetAndSoMaintainInvariant() throws Exception {
        context.checking(new Expectations() {{
            one(guard).guarding(); will(returnValue(true));
            one(guard).execute(with(any(Callable.class)));
        }});
        counter.reset();
    }
    @Test
    public void shouldNotAttemptGuardedCall() throws Exception {
        context.checking(new Expectations() {{
            one(guard).guarding(); will(returnValue(false));
            never(guard).execute(with(any(Callable.class)));
        }});
        counter.reset();
    }
}
```

5.1.3.7 Taking it Further with `Counters`

Using the `Guard` abstraction is interesting but doesn't give us very much other than allowing alternative guard implementations. The current `LockGuard` uses a `Lock` and we could supply an alternative that uses `synchronized` but we don't gain much. As long as we use the same guard instance, we'll get consistent behaviour between implementations. For example,

```
public class SynchronisingGuard implements Guard {
    public static Guard synchronised() {
        return new SynchronisingGuard();
    }
    @Override
    public synchronized <R, E extends Exception> R execute(Callable<R, E> callable) throws E {
        return callable.call();
    }
    @Override
    public Boolean guarding() {
        return true;
    }
}
```

What might be more useful is to come up with a `Counter` abstract to separate the increment and decrement functions from the classes that use them from the guarding policy that controls their access. For example,

```
public interface Counter {
    void increment();
    void decrement();
    Long get();
    void reset();
}
```

With an example implementation of

```
public class LongCounter implements Counter {
    private Long count = new Long(0);
    @Override
    public void increment() {
        count++;
    }
    @Override
    public void decrement() {
        count--;
    }
    @Override
    public Long get() {
        return count;
    }
    @Override
    public void reset() {
        count = new Long(0);
    }
}
```

or an atomic version

```
public class AtomicLongCounter implements Counter {
    private volatile final AtomicLong count = new AtomicLong();
    @Override
    public void increment() {
        count.getAndIncrement();
    }
    @Override
    public void decrement() {
        count.getAndDecrement();
    }
    @Override
    public void reset() {
        count.set(0);
    }
    public Long get() {
        return count.get();
    }
}
```

Which means we can change the `ThreadCounter` to take the dependencies on construction.

```
@ThreadSafe
public class ThreadCounter implements ThreadObserver {
    private final Counter activeThreads;
    private final Counter createdThreads;
    private final Guard guard;
    public ThreadCounter(Guard guard, Counter activeThreads, Counter createdThreads) {
        this.guard = guard;
        this.activeThreads = activeThreads;
        this.createdThreads = createdThreads;
    }
    ...
}
```

We can therefore construct a `ThreadCounter` with different semantics when it comes to use from a concurrent context. For example,

```
public static ThreadObserver createLockBasedThreadSafeCounter() {
    return new ThreadCounter(new LockingGuard(new ReentrantLock()), new LongCounter(), new LongCounter())
}
```

creates a thread safe thread counter using the reentrant lock to ensure the invariant around reset is maintained. The visibility of writes against the (non-thread safe) `LongCounter` can be ensured by the `volatile` keyword in `LongCounter`.

```
public static ThreadObserver createThreadSafeCounterWithoutMaintainingResetInvariant() {
    return new ThreadCounter(unguarded(), new AtomicLongCounter(), new AtomicLongCounter());
}
```

The above statement creates a thread counter which is thread safe (by virtue of the `AtomicLongCounters`) but doesn't maintain the invariant. The unguarded `Guard` doesn't include any synchronisation.

```
public static ThreadObserver createSynchronisedThreadSafeCounter() {
    return new ThreadCounter(synchronised(), new LongCounter(), new LongCounter());
}
```

The above statement is roughly equivalent, it creates a thread safe thread counter but uses a Java monitor rather than a `Lock` for its guarding policy. It maintains the invariant as before. The `synchronised` method is the static creation method for `SynchronisingGuard` class.

```
public static ThreadObserver createNonThreadSafeCounter() {
    return new ThreadCounter(unguarded(), new LongCounter(), new LongCounter());
}
```

The above statement creates a non-thread safe version.

The original functionality test can also be updated to use mocks (as the increment functionality can be tested separately in the `Counter` implementations. For example, using JMock, we can ensure increment and decrement functionality is called against a `Counter` instance.

```
@RunWith(JMock.class)
public class ThreadCounterTest {
    private final Mockery context = new Mockery();
    private final Counter activeThreads = context.mock(Counter.class, "active");
    private final Counter createdThreads = context.mock(Counter.class, "created");
    private ThreadCounter counter;
    @Before
    public void setupCounter() {
        counter = new ThreadCounter(unguarded(), activeThreads, createdThreads);
    }
    @Test
    public void shouldIncrementActiveCount() {
        context.checking(new Expectations() {{
            one(activeThreads).increment();
        }});
        counter.threadStarted();
    }
    @Test
    public void shouldDecrementActiveThreadCount() {
        context.checking(new Expectations() {{
            one(activeThreads).decrement();
        }});
        counter.threadTerminated();
    }
    @Test
    public void shouldIncrementCreatedCount() {
        context.checking(new Expectations(){{
            one(createdThreads).increment();
        }});
        counter.threadCreated();
    }
    @Test
    public void shouldResetCounts() {
        context.checking(new Expectations(){{
            one(activeThreads).reset();
            one(createdThreads).reset();
        }});
        counter.reset();
    }
}
```

5.1.4 Optimistic / Software Transaction Memory

In principle, a Software Transactional Memory (STM) version of the `ThreadCounter` should isolate access to the shared memory (the `Counters` in our case) such that the integrity of that access is maintained even from a concurrent context. The existing concurrent tests should all pass.

Using an STM based implementation of the `ThreadCounter`'s `Guard`, we can slot straight into the unit of work that the guard defines. In this way, the gaurd is no longer protecting or synchronising access but instead defining actions that will form an atomic unit of work.

For example, the snippet below shows how the `guard` co-ordinates access to decrement the counter.

```
public void threadTerminated() {
    guard.execute( decrement(activeThreads));
}
```

When an STM implementation of the `guard` is used, we can define our unit of work. For example,

```
public class StmGuard implements Guard {
    public <R, E extends Exception> R execute(Callable<R, E> callable) throws E {
        return  runAtomically(callable);
    }
}
```

Where the `runAtomically` method delegates to the underlying STM library. In this case, we're using the Multiverse STM library as the underlying library and Akka STM to provide some neater abstractions.

```
    public class RunAtomically<R, E extends Exception> extends Atomic<R> {
        private final Callable<R, E> callable;
        public static <R, E extends Exception> R runAtomically(Callable<R, E> callable) {
            return new RunAtomically<R, E>(callable).execute();
        }
        public RunAtomically(Callable<R, E> callable) {
            this.callable = callable;
        }
        @Override
        public R atomically() {
            try {
                return callable.call();
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
```

A call to the is basically equivalent to the following (minus the exception handling).

```
new Atomic<R>() {
    return callable.call();
}.execute();
```

So the above defines the unit of work but any shared memory to be included in the transaction has to be defined as a *transactional reference*. This is done by defining them as a `Ref` instance. For example, to make use of the new `StmGuard` we would have to combine them with a `Counter` as a *transactional reference.*

```
@Not(ThreadSafe.class)
public class TransactionalReferenceCounter implements Counter {
    private final Ref<Long> count = new Ref<Long>(0L);
    @Override
    public void increment() {
        count.set(count.get() + 1L);
    }
    @Override
    public void decrement() {
        count.set(count.get() - 1);
    }
    @Override
    public Long get() {
        return count.get();
    }
    @Override
    public void reset() {
        count.set(0L);
    }
}
```

As you'll notice, this class in itself isn't thread safe, nor is the `StmGuard`. However, when they're combined with the `ThreadCounter` they'll pass all our previously defined tests (including the concurrent ones). Therefore, the class construction below represents a thread safe `ThreadCounter` that will also maintain the invariant.

```
static ThreadObserver createThreadSafeCounterMaintainingInvariant() {
    return new ThreadCounter(new StmGuard(), new TransactionalReferenceCounter(), new TransactionalRefere
}
```

If we consider an implementation similar to the pessimistic `createThreadSafeCounterWithoutMaintainingResetInvariant` we can bypass the guard and implement atomicity around the individual mutators as below.

```
@ThreadSafe
public class StmAtomicLongCounter implements Counter {
    private final Ref<Long> count = new Ref<Long>(0L);
    @Override
    public void increment() {
        new Atomic<Long>() {
            @Override
            public Long atomically() {
                return count.set(count.get() + 1L);
            }
        }.execute();
    }
    @Override
    public void decrement() {
        new Atomic<Long>() {
            @Override
            public Long atomically() {
                return count.set(count.get() - 1L);
            }
        }.execute();
    }
    @Override
    public Long get() {
        return count.get();
    }
    @Override
    public void reset() {
        new Atomic<Long>() {
            @Override
            public Long atomically() {
                return count.set(0L);
            }
        }.execute();
    }
}
```

Which in context of the `ThreadCounter` class would be used as follows.

```
static ThreadObserver createThreadSafeCounterWithoutMaintainingResetInvariant() {
    return new ThreadCounter(unguarded(), new StmAtomicLongCounter(), new StmAtomicLongCounter());
}
```

6 Instrumenting Thread Timings using ThreadPoolTimer

6.1 Instrumenting Thread Pool Timings

6.1.1 Testing using Time

Testing something that involves time can be tricky as we don't want to introduce non-determinism by using the real system clock. Instead, we generally have to try and control time using collaborators such as mock system clock like the `Clock` interface. For example, a `StopWatch` class night maintain an internal time which it can use to compare with the current time in order to work out the elapsed time. A straight forward implementation might look like the following (taken from tempus-fugit).

```
public class BadStopWatch {
    private Date startDate;
    private long elapsedTime;
    public BadStopWatch() {
        this.startDate = new Date();
    }
    public Duration elapsedTime() {
        return millis(new Date().getTime() - startDate.getTime());
    }
}
```

Writing the (rather silly) test below highlights a problem using real time in the class

```
public class BadStopWatchTest {
    @Test
    public void elapsedTime() throws InterruptedException{
        BadStopWatch watch = new BadStopWatch();
        Thread.sleep(millis(100));
        assertThat(watch.elapsedTime(), is(millis(100)));
    }
}
```

We've introduced non-determinism by using real time, there's no guarantee that we can accurately delay the execution between object constructions and evaluation of the assertions for precisely 100 milliseconds. Unsurprisingly, the test is unlikely to pass consistently.

```
java.lang.AssertionError:
Expected: is <Duration 100 MILLISECONDS>
     got: <Duration 103 MILLISECONDS>
at org.junit.Assert.assertThat(Assert.java:778)
at org.junit.Assert.assertThat(Assert.java:736)
at com.google.code.tempusfugit.temporal.BadStopWatchTest.elapsedTime(BadStopWatchTest.java:32)
```

We clearly need a way to inject a clock that we can control. We can improve the implementation above by introducing the `Clock` interface and injecting a mock instance using jmock.

```
public class BetterStopWatch {
    private Date startDate;
    private long elapsedTime;
    private Clock clock;
    public BetterStopWatch(Clock clock) {
        this.clock = clock;
        this.startDate = clock.time();
    }
    public Duration elapsedTime() {
        return millis(clock.time() - startDate.getTime());
    }
}
```

```
@Test
public void elapsedTimeFromBetterStopWatch() {
    context.checking(new Expectations() {{
        one(clock).time(); will(returnValue(new Date(0)));
        one(clock).time(); will(returnValue(new Date(100)));
    }});
    BetterStopWatch watch = new BetterStopWatch(clock);
    assertThat(watch.elapsedTime(), is(millis(100)));
}
```

Alternatively, we could create our own mock to encapsulate this kind of behaviour like the following.

```
public final class MovableClock implements Clock {
    private final Date now;
    public MovableClock() {
        now = new Date(0);
    }
    public MovableClock(Date date) {
        now = new Date(date.getTime());
    }
    public Date time() {
        return new Date(now.getTime());
    }
    public void incrementBy(Duration time) {
        now.setTime(now.getTime() + time.inMillis());
    }
}
```

With the following test

```
@Test
public void elapsedTimeFromBetterStopWatch() {
    BetterStopWatch watch = new BetterStopWatch(clock);
    clock.incrementBy(millis(100));
    assertThat(watch.elapsedTime(), is(millis(100)));
}
```

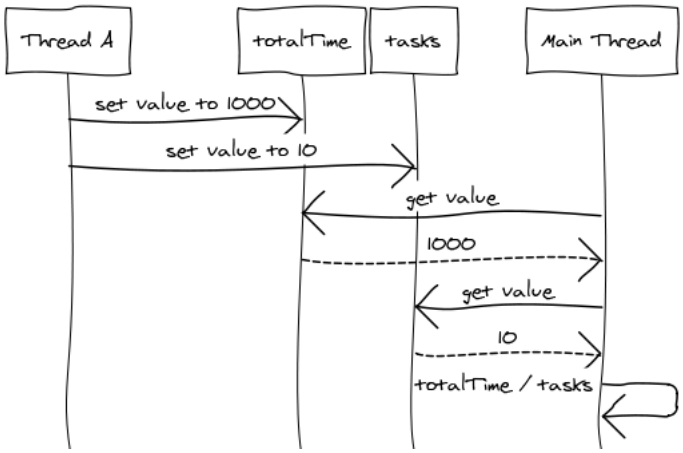
6.1.2 The Race Condition Involving Time

Applying the principle above to the `ThreadPoolTimer` proved a little more involved. There is a race condition when calculating the mean execution time. Here, we have time being stored along with

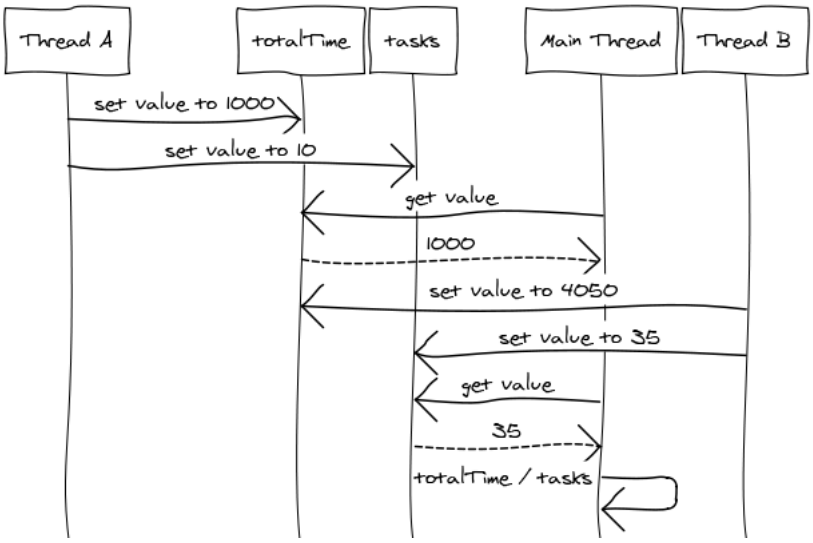
the total number of executions. Divide one by the other to get the mean, but these are two operations which if uncontrolled, introduce the possibility of unlucky timing in terms of the interleaving with other threads affecting those numbers.

```
@Override
public Long getMeanExecutionTime() {
    return totalTime / tasks;
}
```

The sequence of events is shown below when access to the two variables is uncoordinated.



Uncoordinated access works fine when `Thread A` is the only one writing to the variables, the result will be `1000 / 10 = 100`. However, with the introduction of another thread and different interleaving, the result can be skewed. For example,



With this interleaving, the `totalTime` is updated to `4050` but *after* the main thread has read the value for `totalTime` and *before* the related operation to set the task count to `35` could be completed. This update hasn't been able to complete before part of another operation has begun and so the consistency of the later will be compromised. The values the main thread will use can be seen by the dotted return value above; `1000 / 35 = 28.6`, it should either be `1000 / 10` or `4050 / 35` but not part of each as in the example above.

Testing for consistency of the mean execution time (`getMeanExecutionTime`) is difficult. Initially, a basic approach for the pessimistic version of the `ThreadPoolTimer` could use real time but introduce uniform distribution of wait time for each timed thread. We can force an execution time of between zero and five milliseconds and rely on the uniformity of the pseudo-random number generator to make assertions. The theory being that any failures in the assertion are down to bugs in the concurrent usage of the timer rather than natural variation in the wait time.

```
public class ThreadPoolTimerIntegrationTest {
    private static final ThreadPoolTimer timer = new ThreadPoolTimer(...);
    @Rule public ConcurrentRule concurrent = new ConcurrentRule();
    @Rule public RepeatingRule repeating = new RepeatingRule();
    @Concurrent (count = 50)
    @Repeating (repetition = 100)
    @Test
    public void executeTask() {
        Runnable task = newRunnable();
        timer.beforeExecute(currentThread(), task);
        Introduce.jitter(upTo(millis(5)));
        timer.afterExecute(task, NO_EXCEPTION);
    }
    @AfterClass
    public static void verifyCounters() {
        assertThat(timer.getNumberOfExecutions(), is(5000L));
        assertThat(timer.getMeanExecutionTime(), is(2L));
        assertThat(timer.getTerminated(), is(0L));
    }
}
```

Here, the test is using the same approach to spawning multiple threads as the `ThreadCounterIntegrationTest`. Using real time here, we're attempting to introduce a delay of 2.5 milliseconds (on average) between `timer.beforeExecute` and `timer.afterExecute`. The assertions ensure that the number of executions is correct (as we know the expected count ahead of time) and make a best guess on the mean execution time. Mostly, this will pass but it still represents an intermittently failing test. Not good.

In the optimistic version, the actual time to execute outweighed the artificial delay skewing the assumption that on average the time taken would be 2.5 milliseconds. We can't rely on the forced delay being the dominant contributor as the actual execution time took proportionally longer than the simulated delay. Anecdotally, this is likely to be caused by contention and retries in the STM but we'll revisit that later. The affect was a good reminder why using real time can be problematic and forced a rethink.

Somehow, we really want to control time but with more complex semantics than the simple `StopWatch` example above. In the case of the `ThreadPoolTimer` it effectively has to maintain multiple stopwatches, so we'd need to control multiple instances of time!

6.1.3 The Use of ThreadLocal

The `ThreadLocal` class allows us to associate a variable with a particular thread, it basically maintains a map of threads to instance variables. Because only a single thread can access a variable, it is inherently thread-safe and doesn't really fall into a classification as either optimistic or pessimistic control; it's just enforcing serial, single-threaded behaviour.

In testing for the race condition when calculating mean execution time, we make life easier for ourselves if we can control the stopwatch used to time the execution from within the test. Usually, we

can just use something like the `MovableClock` class to do this, effectively mocking the stopwatch within the test. However, the `ThreadPoolTimer` is required to use multiple stopwatches, one for each thread it's responsible for timing.

For example, when the timer starts the stopwatch, it must do so for the current thread. The class requires that the current thread be passed into the `beforeExecute` method in order to ensure this. When another timer is started by calling the same method (lets say, at the same time, but from a different thread), the class should start a new stopwatch tied to the current thread. This requirement is largely influenced by the semantics of the `java.util.concurrent.ThreadPoolExecutor` which provides the before and after extension points that we'll be using.

A basic implementation would be to associate a thread with a map, start a stopwatch and associate it with that thread. Fortunately for us, that's exactly what `ThreadLocal` provides. So, assuming the method is called with the correct parameters (see the `assert` below), we can just use a `ThreadLocalStopWatch` for the `timer` instance below.

```
@Override
public void beforeExecute(Thread thread, Runnable task) {
    assert(Thread.currentThread().equals(thread));
    timer.start();
    // increment the total task count
}
@Override
public void afterExecute(Runnable task, Throwable throwable) {
    timer.stop();
    totalTime.add(timer.elapsedTime());
}
```

This still leaves the question of testing it *and* controlling the time. The test above uses a static `ThreadPoolTimer` and so we need to be able to share an instance of a `Clock` between the threads (as it will also need to be static) but maintain per-thread semantics. Sounds like a job for `ThreadLocal` again.

```
public class ThreadLocalMovableClock implements Clock {
    private final ThreadLocal<Date> now;
    public ThreadLocalMovableClock() {
        now = new ThreadLocal<Date>() {
            @Override
            protected Date initialValue() {
                return new Date(0);
            }
        };
    }
    public ThreadLocalMovableClock(final Date date) {
        now = new ThreadLocal<Date>() {
            @Override
            protected Date initialValue() {
                return new Date(date.getTime());
            }
        };
    }
    @Override
    public Date time() {
        return new Date(now.get().getTime());
    }
    public void incrementBy(Duration time) {
        now.get().setTime(now.get().getTime() + time.inMillis());
    }
}
```

We can now use this to make our test more deterministic and not reliant on real time.

```
public class ThreadPoolTimerIntegrationTest {
    private static final ThreadLocalMovableClock clock = new ThreadLocalMovableClock();
    private static final ThreadPoolTimer timer = new ThreadPoolTimer(...);
    private static final Throwable NO_EXCEPTION = null;

    @Rule public ConcurrentRule concurrent = new ConcurrentRule();
    @Rule public RepeatingRule repeating = new RepeatingRule();
    @Concurrent (count = 50)
    @Repeating (repetition = 100)
    @Test
    public void executeTask() {
        Runnable task = newRunnable();
        timer.beforeExecute(currentThread(), task);
        clock.incrementBy(millis(400));
        timer.afterExecute(task, NO_EXCEPTION);
        Introduce.jitter();
    }

    @AfterClass
    public static void verifyCounters() {
        assertThat(timer.getNumberOfExecutions(), is(5000L));
        assertThat(timer.getMeanExecutionTime(), is(400L));
        assertThat(timer.getTerminated(), is(0L));
    }
}
```

To specifically test for the race condition, we should be able check the consistency of the date during multiple updates. A test similar to the above but with assertions immediately after the updates would be better but the tricky part is moving time forward a different amount for each thread and be able to make meaningful assertions. The test below achieves this by manually controlling the number of threads and iterations per thread (rather than use the `ConcurrentRule` and `RepeatingRule` and then using the thread count to create a multiplier for the delay in the stopwatch.

```
public class ThreadPoolTimerRaceConditionIntegrationTest {
    private static final int threadCount = 50;
    private static final int repetitions = 100;
    private static final ThreadLocalMovableClock clock = new ThreadLocalMovableClock();
    private static final ThreadPoolTimer timer = new ThreadPoolTimer(new SynchronisingGuard(), new Thread
    private static final Throwable NO_EXCEPTION = null;
    @Test
    public void executeTask() throws InterruptedException, ExecutionException {
        List<Future<?>> futures = new ArrayList<Future<?>>();
        ExecutorService pool = newFixedThreadPool(threadCount);
        for (int i = 1; i <= threadCount; i++)
            futures.add(pool.submit(newTestThread(millis(threadCount * 10))));
        for (Future<?> future : futures)
            future.get();
        shutdown(pool).waitingForCompletion(seconds(5));
    }
    private static Callable<Void> newTestThread(final Duration delay) {
        return new Callable<Void>() {
            @Override
            public Void call() throws RuntimeException {
                for (int count = 1; count <= repetitions; count++) {
                    Runnable task = newRunnable();
                    timer.beforeExecute(currentThread(), task);
                    clock.incrementBy(delay);
                    timer.afterExecute(task, NO_EXCEPTION);
                    assertThat(timer.getMeanExecutionTime(), is(delay.inMillis()));
                    Introduce.jitter();
                }
                return null;
            }
        };
    }
}
```

6.1.4 What to Guard

Having gone some way to creating tests to uncover concurrency problems, a basic implementation of the `ThreadPoolTimer` might look like the following

```
public class ThreadPoolTimer implements ThreadPoolObserver, ThreadPoolTimerMBean {
    private final Guard guard;
    private final StopWatch timer;
    private final Counter tasks;
    private final Counter terminated;
    private final AccumulatingCounter<Duration> totalTime;
    public ThreadPoolTimer(Guard guard, StopWatch timer, Counter tasks, Counter terminated, AccumulatingC
        this.timer = timer;
        this.tasks = tasks;
        this.terminated = terminated;
        this.totalTime = totalTime;
        this.guard = guard;
    }
    @Override
    public void beforeExecute(Thread thread, Runnable task) {
        assert(Thread.currentThread().equals(thread));
        timer.start();
    }
    @Override
    public void afterExecute(Runnable task, Throwable throwable) {
        timer.stop();
        tasks.increment();
        totalTime.add(timer.elapsedTime());
    }
    @Override
    public void terminated() {
        terminated.increment();
    }
    @Override
    public Long getNumberOfExecutions() {
        return tasks.get();
    }
    @Override
    public Long getTotalTime() {
        return totalTime.get();
    }
    @Override
    public Long getMeanExecutionTime() {
        return guard.execute(divide(totalTime, by(tasks)));
    }
    @Override
    public Long getTerminated() {
        return terminated.get();
    }
    @Override
    public void reset() {
        totalTime.reset();
        tasks.reset();
        terminated.reset();
    }
}
```

This implementation fails the above race condition test (`ThreadPoolTimerRaceConditionIntegrationTest`). The use of the `guard` in the `getMeanExecutionTime` and variable write methods ensure that writes to individual variables are coordinated with the reads. Updates during the read are prevented but our test still fails. It turns out we've missed something fundamental here, the guard doesn't actually protect us from specific interleaving when the protected methods have themselves completed. The interleaving showing diagram in Fig.XXX above is still very much possible. We'd need to coordinate read and writes access of multiple variables in order to preserve consistent behaviour under concurrent usage.

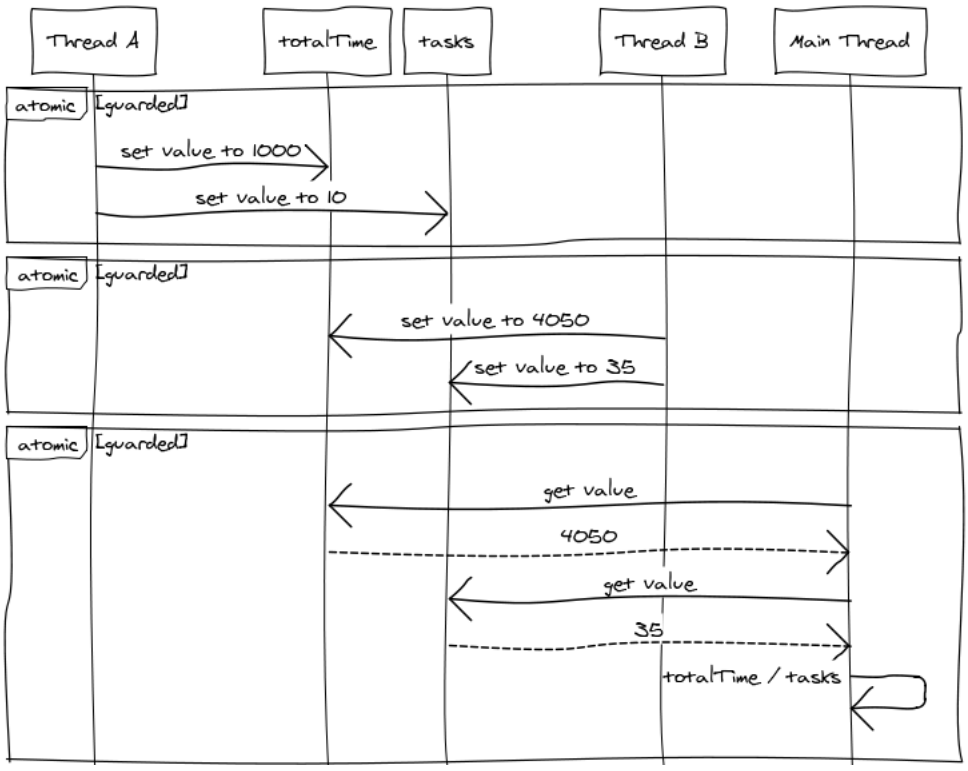
The offending methods from the the original implementation are shown below.

```
public void afterExecute(Runnable task, Throwable throwable) {
    timer.stop();
    tasks.increment();
    totalTime.add(timer.elapsedTime());
}
public Long getMeanExecutionTime() {
    return guard.execute(divide(totalTime, by(tasks)));
}
```

This shows that although the three lines in the `afterExecute` method are individually thread safe, they do not preserve any aromaticity with respect to each other. The task counter and total time can be updated independently as in diagram Fig.XXX above. We can fix this by employing the same gaurd the `getMeanExecutionTime` method uses as below.

```
public void beforeExecute(Thread thread, Runnable task) {
    assert (Thread.currentThread().equals(thread));
    timer.start();
}
public void afterExecute(Runnable task, Throwable throwable) {
    timer.stop();
     guard.execute(new Callable<Void, RuntimeException>() {
        @Override
        public Void call() throws RuntimeException {
            tasks.increment();
            totalTime.add(timer.elapsedTime());
            return null;
        }
    });
}
public Long getMeanExecutionTime() {
    return guard.execute(divide(totalTime, by(tasks)));
}
```

The implementation now passes our test. Access to individual variables is now coordinated using the guard as shown in the gorilla UML below. The diagram is trying to indicate that the updates and the reads (along with the calculation) are now all protected.



We can further tidy up the code by using instances of `Callable` classes rather than the anonymous version like this.

```
public void afterExecute(Runnable task, Throwable throwable) {
    timer.stop();
    guard.execute(
         allOf(
             increment(tasks),
             add(timer.elapsedTime(), to(totalTime))
        )
    );
}
```

6.1.5 Summary

The development steps for the `ThreadPoolTimer` followed the now familiar steps.

1. Develop unit test to drive out the behaviour of the timer in a **non-concurrent usage** (`ThreadPoolTimerTest`)
2. Develop the basic timer class to pass the test
3. Develop the a integration test running in a concurrent context (`ThreadPoolTimerIntegrationTest`) to identify individual elements that require concurrency control.
4. Extend the class, make the test pass. Our example forced the use of `Counter` variables which could be made thread safe.
5. Identify any collaborating elements that require concurrency control, formulate as a test (`ThreadPoolTimerRaceConditionIntegrationTest`)

6 Extend the class, make the test pass. Our example forced guarding collaborating variables rather than making individual variables thread safe.

Interestingly, to some degree steps 3. and 5. require some analysis from the developer *before* the test can be written. For step 1., the test can genuinely be written first to *drive* out behaviour but for the other testing steps, an understanding of the concurrency loop holes is required so that a test can be tailored to expose them. This is an uncomfortable situation for the TDD practitioner to be in but with concurrency tests, it often boils down to spotting the holes first then filling them rather than letting the tests expose the holes for you.

7 Instrumenting Throughput

7.1 Throughput

A general purpose interface to expose throughput might look like the following.

```
public interface ThroughputMBean {
    Double getRequestsPerSecond();
    Long getTotalRequests();
}
```

Any implementation would require some form of timer to record elapsed time and the ability to record the number of requests made. If we think about the thing that we want to observe as a *request*, we can phrase an interface to record the start and finish points as the following.

```
public interface RequestObserver {
    Request started();
    public interface Request {
        Duration finished();
    }
}
```

With the intention of using an instance to instrument before / after points something like the following.

```
public void doGet(HttpServletRequest request, HttpServletResponse response) {
    Request request = throughput.started();
    try {
        doSomeWork();
    } finally {
        request.finished();
    }
}
```

In this way, we're able to indicate to some component a request has started and provide a callback object to indicate when its finished. No mention of a timer and lots of "tell, don't ask" which means the component is free to decide what to do with the information. A basic implementation might look like this.

```
public class Throughput implements RequestObserver, ThroughputMBean {
    private final StopWatch timer;
    private final Counter count;
    private final AccumulatingCounter<Duration> totalTime;
    public Throughput(StopWatch timer, Counter count, AccumulatingCounter<Duration> totalTime) {
        this.timer = timer;
        this.count = count;
        this.totalTime = totalTime;
    }
    @Override
    public Request started() {
        timer.start();
        return new Request() {
            @Override
            public Duration finished() {
                count.increment();
                timer.stop();
                totalTime.add(timer.elapsedTime());
                return timer.elapsedTime();
            }
        };
    }

    @Override
    public Double getRequestsPerSecond() {
        return (double) count.get() / ((double) totalTime.get() / 1000);
    }
    @Override
    public Long getTotalRequests() {
        return count.get();
    }
}
```

A unit style test, ensuring just the behaviour and not thread safety, might look like the following.

```
public class ThroughputTest {
    private final StopWatch timer = new StopWatchStub();
    private final Throughput throughput = new Throughput(timer, new LongCounter(), new AtomicMillisecondC
    @Test
    public void calculateThroughputWithNoRequests() {
        throughput.started();
        timer.setElapsedTime(millis(355));
        assertThat(throughput.getRequestsPerSecond(), is( NaN));
    }
    @Test
    public void calculateThroughput() throws Exception {
        makeRequestLasting(millis(250));
        makeRequestLasting(millis(150));
        makeRequestLasting(millis(50));
        makeRequestLasting(millis(300));
        assertThat(throughput.getRequestsPerSecond(), is(5.333333333333333));
    }
    private void makeRequestLasting(Duration duration) {
        RequestObserver.Request request = throughput.started();
        timer.setElapsedTime(duration);
        request.finished();
    }
}
```

In terms of thread safety, the class depends on a `StopWatch` and two `Counter`s. If these are themselves thread safe implementations, then the following test will pass. This highlights that a class can be thread safe in two ways, the first is that its composite variables are thread safe in isolation and the second is that the class itself is free from race conditions and is consistent with any invariants and so on (see the Conclusions section). The following test only tests that the variables are thread safe in isolation is so much as any race condition possible in the `getRequestsPerSecond` isn't exercised.

```
public class ThroughputIntegrationTest {
    private static final ThreadLocalMovableClock clock = new ThreadLocalMovableClock();
    private static final StopWatch timer = new ThreadLocalStopWatch(clock);
    private static final Throughput throughput = new Throughput(timer, new AtomicLongCounter(), new Atomi
    @Rule public ConcurrentRule concurrent = new ConcurrentRule();
    @Rule public RepeatingRule repeating = new RepeatingRule();
    @Concurrent (count = 10)
    @Repeating (repetition = 100)
    @Test
    public void recordThroughput() {
        RequestObserver.Request request = throughput.started();
        clock.incrementBy(millis(250));
        request.finished();
    }
    @AfterClass
    public static void verify() {
        Long requests = throughput.getTotalRequests();
        Double requestsPerSecond = throughput.getRequestsPerSecond();
        assertThat(requests, is(1000L));
        assertThat(requestsPerSecond, is(4D));
    }
}
```

8 Instrumenting Blocking Ratio

8.1 Contention Monitoring and Block/Wait Counts

When making comparisons between the implementations, it may be useful to understand any contention between competing threads on resources. For us, we're interested in contention when accessing shared memory. When shared memory is protected (or serialised) by a single guard, several concurrent requests will *contend* for exclusive access to the guard. A *contention ratio* then, may be useful in understanding the potential strain a protected resources could come under.

A simple contention ratio then can be seen as

```
number of concurrent calls to a shared memory resource : number of guards
```

For example, twenty concurrent calls to a single `synchronized` block would result in a ratio of

```
20:1
```

in other words, the single lock could service only 5% of the time.

However, this is not an easy thing to predict or measure ahead of time. We're not going to be able to tell how many calls to a shared resource are going to be made at precisely the same time. We can however, measure the *affect* of some implied contention. For example, we can measure the number of failed guarded calls and compare these to the number of success calls under different loads. The implication here being that the throughput (the number of requests the system is going to be able to make) will be affected by the contention ratio and by extension the effectiveness of the guard implementation. A guard that processes multiple requests quickly will positively affect throughput compared to a guard that blocks excessively, which will slow things down. At least, that's the theory.

However, because of the way in which our guards protect a shared resource, we can't just monitor the number of failed guarded calls and use it as an indicator of contention. For example, a Software Transaction Memory guard might retry dozens of time before failing, whereas a `Lock` based solution may fail on the first attempt and never retry. Both were asked just once to attempt the call, but the behaviour of the implementation dictates the number of retries. What number represents the request count, the original request (one) of the number of retries (dozens)?

So, to paint a fuller picture, we need to understand our system in terms of contention (how likely is it that a service is going to be heavily contended) as well as how quickly the service can process the contention (both in terms of processing time and time spent co-ordinating access to shared memory) and the observable outcome to the throughput of the service. In summary, if shared memory isn't contended (under load), it's likely that co-ordinating efforts are having little affect on the throughput. We want to be able to stretch the system to simulate contention in order to observe the affect our guards have on throughput.

Lets paraphrase some of these assumptions to make things clearer

- we want to **exaggerate contention in order to exaggerate the affect of a `Guard`** implementation (so that we can evaluate their use)
- we can **increase the contention by increasing the load** (number of concurrent requests) made to a service
- **we can measure a block/wait count** for our `Guard` implementations
- if we **tweak the load so that block/wait counts are similar, a measure of throughput indicates effectiveness** of the `Guard` (as well as the intermediate code on the critical path)

8.1.1 The Blocking Coefficient

We introduced the idea of measuring blocked calls or waits above in order to better interpret various measures to evaluate our `Guard`s. We can formalise the idea into talking about a *blocking coefficient*. A coefficient is some multiplicative factor, a constant term affecting a calculation. In our case, the blocking coefficient is amount of time not spent servicing a task (blocking or waiting) expressed as a percentage (or fraction) of the total time.

The blocking coefficient as a fraction is a number between 0 and 1 to indicate how much a particular task is blocking (or waiting). Zero indicates CPU intensive work (no blocking) and a number close to one represents a heavily blocked task. A fully blocked task would have 1 as a coefficient.

For example, if a task is idling (waiting or blocked) 80% of the time and so actually processing 20% of the time, the blocking coefficient is 0.8 (80 / 100).

In terms of our `Guard` implementations, we're able to monitor;

1. **Blocking with Pessimistic (`synchronized`) Control** - the number of requests (or time) blocked whilst attempting to acquire an object monitor.
2. **Waiting with Optimistic (`Lock`s) Control** - the number of requests (or time) made to wait whilst attempting to acquire a lock.
3. **Contention with Optimistic (Software Transactional Memory) Control** - the number of aborted atomic updates (this assumes an abort is the result of an attempt to access a transactional reference which has already been accessed and not some other runtime exception).

In all cases, we can use these values along with total request counts to give us our coefficient. For example, the waiting or blocked count divided by the sample count where the sample count is equal to the number of requests made. This gives the ratio of failed requests (due to blocked or waiting). For example, if 10 calls out of 100 failed, the ratio would be 0.1 (10 / 100).

We can also use the waiting or blocked *time* divided by the CPU (service) time. This produces a similar indication as above but this time in terms of time. For example, if thread A was busy for 1000 milliseconds and waiting for 100 milliseconds, the result would be 0.1 milliseconds (100 microseconds).

See Appendix A for some additional background around why and how threads will enter blocked or waiting states.

8.1.2 Blocking in Pessimistic Concurrency Control

To begin with we're interested in measuring the blocking coefficient caused by the waiting on monitor acquisition. We can use the Java class `ThreadInfo` to get information about a particular thread including the block count (the number of times the thread has been in the `BLOCKED` state) and total elapsed time a thread has been blocked (again, the total time spent in the `BLOCKED` state). The state transition to `BLOCKED` is only possible when a thread is waiting to acquire (or re-acquire) an object's monitor.

Unfortunately, the `ThreadInfo` class doesn't distinguish between the specific monitor a thread is blocked waiting to acquire but we can make some assuptions and provide an approximation using the following code.

```
public class BlockingRatio {
    private final Counter count = new AtomicLongCounter();
    private final Map<Long, Long> blocked = new ConcurrentHashMap<Long, Long>();
    private final ThreadMXBean jvm;
    public BlockingRatio(Factory<ThreadMXBean> factory) {
        jvm = factory.create();
        if (jvm.isThreadContentionMonitoringSupported())
            jvm.setThreadContentionMonitoringEnabled(true);
    }
    public void sample() {
        count.increment();
        blocked.put(currentThread().getId(), getBlockedCount(currentThread()));
    }
    public Double get() {
        double ratio = 0;
        for (Long blocked : this.blocked.values())
            ratio += (double) blocked / (double) count.get();
        return ratio;
    }
    private long getBlockedCount(Thread thread) {
        return jvm.getThreadInfo(thread.getId()).getBlockedCount();
    }
}
```

Here, we assume that client will "sample" blocked calls at appropriate times which is basically during load. We also brush over the fact that the instrumentation itself may influence the results. We use an `AtomicLongCounter` and `ConcurrentHashMap` as they both offer optimistic thread safety (with the implication being that they are fast). We also defer maintaining the consistency of updating `count` and `blocked` together (for example, by using a `Guard`) for the same reason; namely we're favouring performance over accuracy. We use a factory to create the `ThreadMXBean` in order to be able to write unit style tests without using real JVM thread metadata.

As we are interested in monitoring contention around locks, we can conveniently use the `BlockingRatio` class from within a custom `Guard` implementation. The guard is our abstraction for protecting resources and we're interested in understanding contention at this point. As we're also interested in the total number of requests, we can employ the `Throughput` class defined previously in the same place. For example,

```
public class ContentionMonitoringGuard implements Guard, ContentionMonitoringGuardMBean {
    private final BlockingRatio contention = new BlockingRatio(new JmxThreadMxBean());
    private final Throughput throughput;
    public ContentionMonitoringGuard(Throughput throughput) {
        this.throughput = throughput;
    }
    @Override
    public <R, E extends Exception> R execute(Callable<R, E> callable) throws E {
        synchronized (this) {
            RequestObserver.Request request = throughput.started();
            try {
                return callable.call();
            } finally {
                request.finished();
                 contention.sample();
            }
        }
    }
    @Override
    public Boolean guarding() {
        return true;
    }
    @Override
    public Double getContentionRatio() {
        return contention.get();
    }
    @Override
    public Double getRequestsPerSecond() {
        return throughput.getRequestsPerSecond();
    }
    @Override
    public Long getTotalRequests() {
        return throughput.getTotalRequests();
    }
}
```

We include the throughput as well as blocking coefficient so that we can adjust the load later.

An instance of this guard will be used to protect some shared resource and as such will sample the current thread's block count just before releasing it's monitor. A thread which manages to execute the guarded section (the call to `callable.call()`) without being blocked will record no contention. If however, whilst that thread is executing the guarded section, another attempts to do the same, it will block until the first has released the monitor. In this case, the second thread will record a blocked attempt when executing the `contention.sample()` method. A thread dump showing the kind of blocking behaviour that `ContentionMonitoringGuard` would capture as contention is shown below.

```
Thread Thread-0@9: (state = RUNNABLE)
 ...
 - bad.robot.pessimistic.ContentionMonitoringGuardTest$1.call(ContentionMonitoringGuardTest.java:14)
 - bad.robot.pessimistic.ContentionMonitoringGuard.execute(ContentionMonitoringGuard.java:36)
 - bad.robot.pessimistic.ContentionMonitoringGuardTest$3.run(ContentionMonitoringGuardTest.java:36)
 - java.lang.Thread.run(Thread.java:722)
Thread Thread-1@10: (state = BLOCKED)
 - bad.robot.pessimistic.ContentionMonitoringGuard.execute(ContentionMonitoringGuard.java:34)
 - bad.robot.pessimistic.ContentionMonitoringGuardTest$3.run(ContentionMonitoringGuardTest.java:36)
 - java.lang.Thread.run(Thread.java:722)
```

`Thread-0` acquired the guard's monitor and is executing (it's in the `RUNNABLE` state) whilst `Thread-1` is `BLOCKED` at the `execute` call. When `Thread-1` finally continues the `ContentionMonitoringGuard` would indicate a contention ratio of 0.5. Half the requests were contended.

8.1.3 Waiting in Locks

...

8.1.4 Contention in Software Transaction Memory

Using hooks into the Multiverse STM library, we can observe the number of aborts vs the number of successful commits giving us the contention ratio. Multiverse allows us to add a *deferred task* to execute on successful commit and a *compensating task* on aborts. Implementing basic tasks using our existing `Counters`, we can wire up a basic contention monitoring `Guard`. For example, we can re-use the `Increment` class shown below to increment `Counters` *on abort* or *on commit* events.

```
public class Increment<T extends Counter> implements Callable<Void, RuntimeException> {
    private final Counter counter;
    public static <T extends Counter> Increment<T> increment(T counter) {
        return new Increment<T>(counter);
    }
    private Increment(T counter) {
        this.counter = counter;
    }
    @Override
    public Void call() throws RuntimeException {
        counter.increment();
        return null;
    }
}
```

The increment functionality is a `Callable`, so if we adapt it to either a `CompensatingTask` or `DeferredTask`,

```
public final class CallableAdaptors {
    public static CompensatingTask onAbort(final Callable<?, RuntimeException> callable) {
        return new CompensatingTask() {
            @Override
            public void run() {
                callable.call();
            }
        };
    }
    public static DeferredTask onCommit(final Callable<?, RuntimeException> callable) {
        return new DeferredTask() {
            @Override
            public void run() {
                callable.call();
            }
        };
    }
}
```

we can then schedule increment behaviour on the events using our "runner" (and the infrastructure supplied by Multiverse in the `STMUtils` class) below.

```
public class RunAtomically<R, E extends Exception> extends Atomic<R> {
    private final Callable<R, E> callable;
    private final DeferredTask onCommit;
    private final CompensatingTask onAbort;
    public static <R, E extends Exception> R runAtomically(Callable<R, E> callable) {
        return new RunAtomically<R, E>(callable, new DoNothingDeferredTask(), new DoNothingCompensatingTa
    }
    public static <R, E extends Exception> R runAtomically(Callable<R, E> callable, DeferredTask onCommit
        return new RunAtomically<R, E>(callable, onCommit, onAbort).execute();
    }
    RunAtomically(Callable<R, E> callable, DeferredTask onCommit, CompensatingTask onAbort) {
        this.callable = callable;
        this.onCommit = onCommit;
        this.onAbort = onAbort;
    }
    @Override
    public R atomically() {
        try {
             StmUtils.scheduleDeferredTask(onCommit);
             StmUtils.scheduleCompensatingTask(onAbort);
            return callable.call();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Finally, the `Guard` can use the counters to work out contention when using the "runner" to `runAtomically`. For example,

```
public class ContentionMonitoringStmGuard implements Guard, ContentionMonitoringStmGuardMBean {
    private final Counter aborts = new AtomicLongCounter();
    private final Counter commits = new AtomicLongCounter();
    @Override
    public <R, E extends Exception> R execute(Callable<R, E> callable) throws E {
        return  runAtomically(callable, onCommit(increment(commits)), onAbort(increment(aborts)));
    }
    @Override
    public Boolean guarding() {
        return true;
    }
    @Override
    public Double getContentionRatio() {
        return (double) aborts.get() / (double) commits.get();
    }
}
```

8.1.5 Choosing the Optimal Number of Threads

As we saw earlier, if we can keep the blocking coefficient in a similar range by adjusting the load, we can make stronger inferences against the resulting throughput. However, this isn't the whole story. We also want to try and tune the system so that it's able to utilise the system processors efficiently. It may therefore make sense to have a guide when deciding how many threads to use when testing.

8.1.5.1 CPU Bound Tasks

For CPU bound tasks, Goetz (2002, 2006.) recommends

```
threads = number of CPUs + 1
```

Which is intuitive as if a CPU is being kept busy, we can't do more work than the number of CPUs. Goetz purports that the additional CPU has been shown as an improvement over omitting it (2006. pp.XXX), presumably helping with thread context switching.

8.1.5.2 IO Bound Tasks

Working out the optimal number for IO bound tasks is less obvious. During an IO bound task, a CPU will be left idle (waiting or blocking). This idle time can be better used in initiating another IO bound request.

Subramaniam (2011, p.31) describes the optimal number of threads in terms of the following formula.

```
threads = number of cores /  (1 – blocking coefficient)
```

$$t = \frac{c}{1-w}$$

And Goetz (2002) describes the optimal number of threads in terms of the following.

```
threads = number of cores * (1 + wait time / service time)
```

$$t = c\left(1 + \frac{w}{s}\right)$$

Where we can think of `wait time / service` time as a measure of how contended the task is.

When we use equivalent terms in Subramaniam‘s expression we can begin to form the proposition that both formulas are equivalent. Starting with Goetz’s formula, we assert that w+s=1 and remove the service time (s) from Goetz’s formula giving the following

$$t = c\left(1 + \frac{w}{1-w}\right)$$

We can continue by multiplying both sides by 1-w reducing the right hand side to c before reversing the operation and revealing Subramaniam’s expression.

$$t\,(1-w) = c\,(1-w+w)$$

$$t\,(1-w) = c$$

$$t = \frac{c}{1-w}$$

As we were able to show that Subramaniam and Goetz agree on the number of threads to use for IO bound tasks, we'll be confident in our choices of thread pool sizes when it comes to performance testing later.

9 Conclusions

9.1 Conclusions

...

9.1.1 General Comments / Observations

In the pessimistic world, contended locks obviously block. The misconception that locking is expensive comes from this but in fact, uncontended locks add only tens of nanoseconds (biased locking at around 2-4 clock cycles and fast user-model locks otherwise). This is all highly optimised by the JVM as synchronisation has been so common in development. Things like lock elision, escape analysis, adaptive locking and lock coarsening all aim to correct the oversights of the developer. These optimisations and the support for the pessimistic model from the language have set its place and we won't be seeing it go away for some time yet.

From some guy

9.1.2 Fine Grained Concurrent Components

Stressing components like the instrumentation classes in a concurrent context demonstrates typical concurrency control of fine-grained shared memory. It does not, however, demonstrate coarse grained concurrency like the type you might expect when using *business components*. What I mean here is that when two or more *business processes* are run in parallel, the same kind of problems may appear as when we access fine-grained shared memory. The consistency of critical sections may still need to be preserved and race conditions between processes may still exist.

This discussion hasn't focused on this at all.

9.1.3 Pessimistic / Lock Based Synchronisation

As we noted in the Shared Memory Model section, pessimistic solutions revolve around using intrinsic locks, essentially using the `synchronized` keyword. In looking at solutions using this approach, it's important to be aware of some drawbacks associated with it, namely;

- Lock's can block indefinitely, causing non-recoverable liveliness problems such as deadlock.
- Various up-front strategies must be employed to avoid situations above (for example, ensuring consistent lock acquisition ordering to avoid deadlock.

9.1.4 Making Classes Thread Safe

As we've seen from different styles of tests in the Solution section, we can look at class level thread safety as having two dimensions. A class can be thread safe, in terms of

1 its composite variables. Variables which are available for read and write access from multiple threads need to be protected against lost updates (visibility) and write consistency. The `Counter` implementations when used in isolation are a good example here.

   We can protect these at the class level or at the client level. In our examples, it's interesting to note that we haven't needed to implement a `synchronized` version of a `Counter` instead favouring client `Guard` implementations to protect access. We have implemented optimistic versions (for example, `AtomicLongCounter` and `StmAtomicLongCounter`).

2 the relationships between composite variables. Any shared variables from 1. above that collaborate with others should be considered in terms of atomicity. It's likely that any interaction should be executed under a `Guard`. In the same way as a *check then set* operation is subject to

race conditions, any collaboration with shared variables is also open to modifications during the execution of that collaboration which may subvert the outcome.

It may be worth considering *accuracy vs performance* here though as sometimes the consistency isn't always required. An example might be the `reset` method of `ThreadCounter` where we've chosen to reset the active and create threads together preventing modifications to either until the reset is complete but in `ThreadPoolTimer` we've chosen to reset variables independently and allow modifications.

In building out the implementations, I naturally feel into a rhythm that fits into this way of looking at it. This is outlined below.

1. Develop a non-threaded behavioural unit test to help drive out the behaviour of your class.
2. Build the class to pass the test.
3. Develop a threaded integration style test to identify the composite variables of the class that represent shared state (point 1. above).
4. Implement basic protection for the composite variables.
5. Develop a threaded integration style test to highlight collaborating elements that require additional protection to ensure consistency of behaviour (maintaining invariants for example) and avoid race conditions.
6. Implement addition protection against the relationships of the elements, this is a good candidate for using a `Guard`.

10 References

10.1 References

1 Christopher, T. W. and Thiruvathukal, G. K., 2001. *High Performance Java Platform Computing*. Sun Microsystems/Prentice Hall.

2 Kung, H. T. and John T. Robinson. June 1981. *On optimistic methods for concurrency control*. ACM Transactions on Database Systems, Vol. 6, No. 2.

3 Anon. 2011. Concurrency Control. *Wikipedia*, [online] Available at: http://en.wikipedia.org/wiki/Concurrency_control

4 Anon. 2011. Non-blocking algorithm. *Wikipedia*, [online] Available at: http://en.wikipedia.org/wiki/Non-blocking_synchronization

5 Moore, G. 1965. *Cramming More Components Onto Integrated Circuits.* Electronics Magazine.

6 Moore, G. 1975. *Progress in Digital Integrated Electronics.* IEEE, IEDM Tech Digest., pp. 11-13

7 Amdahl, G.M. 1967, *Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities.* Proc. Am. Federation of Information Processing Societies Conf., AFIPS Press., pp. 483-485.

8 Gustafson, J. L. 1988. *Reevaluating Amdahl's Law*. Comm. ACM., pp. 532-533.

9 Gosling, J. Joy, B. Steele, G. and Bracha, G. 2005. *Java Langauge Specification* 3rd Edition. Addison Wesley.

10 Goetz, B. 2002. Java theory and practice: Thread pools and work queues. *IBM DeveloperWorks*, [online] Available at http://www.ibm.com/developerworks/java/library/j-jtp0730/index.html

11 Goetz, B. Peierls, T. Bloch, J. Bowbeer, J. Holmes, D. and Lea, D. 2006. *Java Concurrency in Practice*. 1st Edition. Addison Wesley

12 Dice, D. Shalev, O and shavit, N. 2006. *Transactional Locking II*. 20th International Symposium on Distributed Computing (DISC). pp. 194-208.

13 Lev, Y. Luchangco, V. Marathe, V. J. Moir, M. Nussbaum, D. and Olszewski, M. 2009. *Anatomy of a Scalable Software Transactional Memory* In the proceedings of TRANSACT`09: The 4th ACM SIGPLAN Workshop on Transactional Computing Raleigh.

14 Veentjer, P. 2011. *Multiverse (0.7) Reference Manual*. [online] Available at: http://multiverse.codehaus.org/manual/index.html

11 Appendices

11.1 Appendices

12 Appendix A

12.1 Classification of Java Concurrency Control Mechanisms

For the purposes of this discussion, I've classified various options as either optimistic or pessimistic.

12.1.1 Pessimistic

Class or keyword	Notes
`synchronized`	Exclusive lock is inherently pessimistic. Client threads unable to acquire a object's monitor will enter the BLOCKED state.
`ReentrantLock`	Exclusive locks but with additional functionality meaning they can offer non-blocking semantics (see below). When a client thread in unable to acquire a lock, it will enter the WAITING or TIMED_WAITING state rather than BLOCKED.
`ThreadLocal`	Although avoiding contention, when using `ThreadLocal`, it can be argued that we're expecting the potential for contention and electing to side-step conflicts. As such, it offers no explicit collision detection or recovery as described in the Optimistic Concurrency Control section.

Pessimistic

12.1.2 Optimistic

Technique or keyword	Notes
Software Transactional Memory	Often STM offers automatic retry semantics.
`volatile`	Atomic read and write (Gosling, et al. 2005. Section. 17.7). A write to a volatile field *happens-before* every subsequent read of that field (Gosling, et al. 2005. Section. 17.4.5).
`AtomicInteger` and others	Based on CAS, lock-free algorithm although on some platforms may involve some form of internal locking.

`ReentrantLock`	Non-blocking when used to attempt to acquire a lock (using `tryLock`) preceded with a conditional or allowing the lock to be interrupted (using `lockInterruptibly`) or with a timeout (using `tryLock(long, TimeUnit)`). In this mode, `Lock`s will not block in so much as client threads will not enter the BLOCKED state if unable to acquire the lock but instead will go into WAITING or TIMED_WAITING states. This is the difference between waiting to acquire a lock due to `synchronized` or `wait` as apposed to something that ends up calling a `park` method. See footnote for more details. `ReentrantLock` (and `ReentrantReadWriteLock`) is a type of *ownable syncrhoniser* implying that they will not force client threads to be BLOCKED but will force waiting instead. This can have all the same detrimental affects as blocking in terms of liveliness and performance. See `ThreadMXBean`.

Optimistic

12.1.3 When Threads can be Blocked Waiting

With reference to non-blocking algorithms, a blocked thread is one that to some degree can not progress when waiting for some other thread to release a mutex that it would like to acquire. Java's documentation is reasonable consistent with this definition but it doesn't imply that a thread that is "blocked" is actually in the state BLOCKED. Java itself describes situations where a thread can be *blocked waiting*, meaning the following (taken from the JavaDoc).

A thread can be blocked waiting for one of the following:

- an object monitor to be acquired for entering or reentering a synchronization block/method. The thread is in the BLOCKED state waiting to enter the synchronized statement or method.
- an object monitor to be notified by another thread. The thread is in the WAITING or TIMED_WAITING state due to a call to the `Object.wait` method.
- a synchronization object responsible for the thread parking. The thread is in the WAITING or TIMED_WAITING state due to a call to the `LockSupport.park` method. The synchronization object is the object returned from `LockSupport.getBlocker` method. Typically it is an *ownable synchronizer* or a `Condition`.

12.1.4 How Threads become blocked

A summary of how threads can enter the various states is offered below.

State	As a result of calling
`BLOCKED`	`synchronized` (when monitor is already owned)

TIMED_WAITING	Thread.sleep(duration) Object.wait(timeout) Thread.join(timeout) LockSupport.parkNanos(timeout) LockSupport.parkUntil(timeout)
WAITING	Object.wait() Thread.join() LockSupport.park()

12.1.5 Thread Pool Tuning

T