Final Last Updated January 24, 2015

Java Concurrency in Practice Reading Notes

Ilya Nepomnyashchiy

I received this book at my current job when I started, but hadn't had a chance to get around to it between slacking off and doing other projects. Concurrency is an issue that I think any developer could benefit from having a more solid grasp of (no matter how solid their grasp already), so I figured it'd be a good read before I moved on to some more academic reading for the year.

These are notes for myself. If you have any questions, the book itself is much more detailed That being said, I'd be happy to clarify, debate, or correct any observations that I make here if you e-mail me from my website.

Contents

Part 0

Chapter 1: Introduction

Concurrency is hard, so why do we bother? This chapter gives an overview of the history of concurrency and the benefits it gives to us. In short, threads let us turn complicated asynchronous code into smaller, simpler pieces that look like serial code, and threads let us take advantage of the full power of multiprocessor systems.

Back in the day, computers only ran one program at a time. This was bad for several reasons: sometimes programs would block on resources (such as IO or network) and the CPU would go unused, one program could hog a very large amount of time on a computer, and programs were forced to be giant monolithic pieces of code rather than smaller programs that did one thing very well.

In order to solve this problem, Operating Systems introduced the concept of processes, which could execute simultaneously and communicate with each other through certain methods. Most processes were written sequentially, which models the way most people think about executing tasks, with the exception of some asynchronicity (such as "doing something while the kettle boils").

The advent of threads came for much the same reasons as processes, but threads were allowed to share much of the program state within a process. For this reason, threads are referred to as "lightweight processes". Most Operating Systems treat threads as the main unit of scheduling.

As discussed before, threads confer many benefits upon developers when used correctly. They allow programs to make full use of all processors. They allow developers to model all of their asynchronous tasks sequentially. They simplify handling of events (for example, a server can spawn a new thread for every client request that comes in, rather than forcing a developer to write a non-blocking request handler). They also allow for more responsive UIs, by having a thread that is devoted to responding to UI events and offloading any processing elsewhere.

As powerful as threads are, they also come with some risks. First of all, threads come with safety hazards. A subtle change in the ordering of thread scheduling can cause a poorly written program to behave incorrectly, and this may be hard to uncover during testing. Second of all, threads can have liveness hazards, which is a situation where multiple threads try to access the same set of resources, but are locked out by other threads, thus causing them to permanently be unable to do anything. Finally, threads can have performance hazards, for various reasons. In short, the three goals here are "nothing bad ever happens", "something good eventually happens", and "something good happens quickly", respectively.

Even if you never write a multithreaded program yourself, frameworks can introduce multithreadedness to your program. Some examples of this are servlets, UI libraries and timed executions. Any state that is accessed by multiple threads, and any state that that state depends on, etc. must be thread-safe. The book demonstrates how this causes a ripple effect through your program.

Part I: Fundamentals

Chapter 2: Thread safety

Thread-safe code is a matter of managing access to shared, mutable state. Developers must make sure that if they manage access to any shared state through synchronization, then they manage **all** accesses to that shared state through synchronization. Furthermore, all shared state must have synchronization, and there are no exceptions to this rule. There are three ways to deal with shared state: don't share it across threads, make it immutable, or use synchronization methods such as locks and mutexes. It is much easier to design a class to be thread-safe from the beginning than to add it in later. Good object-oriented design helps with this a great deal.

Most definitions of thread-safety are either extremely involved or circular. The book winds up with the definition of "A class is thread-safe if it behaves correctly when accessed from multiple threads...with no additional synchronization or other coordination on the part of calling code." In other words, thread-safe classes encapsulate their own synchronization.

One easy way to be thread-safe is to be stateless. Obviously this doesn't work for all cases. If a class does have state, one thing that the class writer can do is to always use thread-safe classes (such as `AtomicLong` and `AtomicReference`) to manage state. However, this is not always sufficient.

If your code uses what the book calls "compound actions", then those actions often need to be made atomic. The book's definition of atomic is that another thread can never see the middle of the operation. Some actions that may need to be atomic but aren't on their own are check-then-act (e.g. "if this class is not initialized, initialize it") and read-modify-write (e.g. `count++`, which is deceiving because it is actually three actions, but is written so compactly).

The way to make actions such as this atomic is by using locks. Java objects all carry an intrinsic lock which can be used by applying the `synchronized` keyword to either a method (which forces the acquisition of a lock for the entire method block) or for a block of code. Methods that are synchronized use their object's intrinsic lock, or in the case of static methods, the intrinsic lock of the class object.

These locks are re-entrant, which means that a thread can acquire the same lock multiple times without blocking. This allows developers to call a synchronized block from a synchronized block. Every shared, mutable variable should be guarded by only one lock, and any set of variables that are involved in the same invariant should be guarded by the same lock. Otherwise, it is as if they are not guarded at all!

For performance considerations, the developer should make sure that they put as little code as possible into `synchronized` blocks. For example, synchronizing the entire method of a servlet defeats the purpose, as only one client can have its request serviced simultaneously. It is better to only put the parts that access shared state into these blocks, allowing the CPU-intensive pieces to operate in parallel. On the other hand, acquiring and releasing locks has some overhead, so it is best not to go crazy in making these blocks too small. Obviously, thread-safety and correctness should never be sacrificed for performance. Furthermore, simplicity should not be prematurely sacrificed for performance. All of these considerations create a trade-off.

Chapter 3: Sharing Objects

It turns out that `synchronized` blocks serve two purposes: they allow for mutual exclusion, as we saw in the last chapter, and they make sure that updates to variables are actually seen by other threads. The JVM is given a lot of freedom to re-order memory accesses (for performance reasons), so without proper synchronization, your other threads may wind up seeing updates to variables either in the wrong order or, even worse, never. Moreover, while the stale data you see when this problem occurs tends to at least be the data resulting from one write, the stale data in 64-bit variables can be the result of two different writes, because the JVM is allowed to turn 64-bit writes into two 32-bit writes.

Once again, the general solution to this problem is to always use synchronization when accessing shared variables, no matter what the type of operation is. When thread B acquires a lock that was just released by thread A, the JVM ensures that thread B can see all of the updates that thread A could see.

Another potential solution to variable visibility is the `volatile` keyword. This keyword provides visibility but not mutual exclusion – it tells the compiler that the variable is shared and thus that updates should not be cached or hidden from other processors. Furthermore, a write to a volatile variable one one thread ensures that another thread which reads that variable can also see all updates that the first thread made to any other variables before that write. However, it is not usually recommended to heavily use this fact to ensure correctness of your program, as this behavior can be rather difficult to reason about. Volatile variables are best used as state variables. The conditions for best use of them safely are that locking is not required for interaction with them, they are not involved in any invariants with other variables, and that writes to this variable do not depend on its value.

The next section discusses publishing variables and the concept of "escaping references," or references that are published accidentally. For example, returning a list from a public method on a class publishes references to all objects inside the list, and returning a private array publishes the array unless you make a copy. These are unsafe because another thread could do something unsafe or malicious with this state that it has access to (the book makes the analogy of getting your password stolen: it does not matter if someone has already used it maliciously, as the fact that it was stolen is already bad). The most surprising example of escaping references is that of an escaping `this` reference when anonymous inner classes are written in constructors (this includes `Callable`s, or starting threads, for example). If you need to do this, the book suggests making them in a static factory method, as otherwise this anonymous class will have a reference to `this`, which may be undesired. Calling an overridable method can also cause the reference to `this` to escape.

One obvious way to avoid state publishing issues is to confine the state to the current thread. There are three ways of doing this. The first is ad-hoc, where the implementation is entirely responsible for respecting the confinement. This is incredibly fragile and not recommended, but many GUI frameworks work this way. They merely expect you to not use their state from the non-GUI thread. The second way is to confine the state to the stack. This is easiest with primitive variables, as it is impossible to obtain a reference to them. For object references, one needs to merely make sure to never publish a reference to the object and to keep only one reference to it in the current method. The final way to confine state to a thread is to use the `ThreadLocal` object, which keeps one value per thread. One should be careful with this object, though, as it tends to encourage keeping global state.

Another easy way to avoid state publishing issues is to publish immutable objects. An object is immutable if its state cannot be modified after construction, all its fields are `final`, and it is properly constructed (does not allow the `this` reference to escape). Some immutable classes, such as `String`, actually have non-final fields but are still immutable; however, this involves intricate knowledge of the JVM memory model. Immutable objects may still use mutable objects internally, as long as their references to them are `final`. These objects are much easier to reason about, since they cannot change, and are safer from a multithreaded perspective, since there is no worry of visibility of changes. For example, putting an immutable variable into a `volatile` field to represent state can provide a thread-safe way to see certain state, such as the one value cache of the factorizer servlet we saw earlier.

Even for mutable classes, it is recommended that as many fields as possible be final, as each such field makes it easier to reason about the class' behavior.

Poor use of publication techniques can result in various non-intuitive behaviors. Other threads can see an out-of-date reference to an object, or they could see an up-to-date reference to a partially constructed object, or they could see an up-to-date object to an object whose fields are out-of-date. Moreover, a thread could read the same value twice in a row and see different values (thus causing hilarious results, such as `n != n`). On the other hand, the JVM guarantees that any immutable objects which meet the above requirements can be safely accessed even if they are not safely published.

There are four ways to safely publish a reference to an object, which all share the property that they publish the reference to the object and the object's state simultaneously. The first is initializing the reference to the object from a static initializer. The second is storing a reference to the object into a `volatile` field or `AtomicReference`. The third is to store the reference into a `final` field of a properly constructed object. The final way is to store a

reference to the object into a field that is properly guarded by a lock. Note that placing a reference into a thread-safe collection satisfies this final method.

Some objects are "effectively immutable", as they do not strictly meet the definition above, but their state will not change over time. One example of this is the `Date` class. These objects can be used safely by any thread provided that they are safely published.

In short, immutable objects can be published through any mechanism, effectively immutable objects must be safely published, and mutable objects must be safely published *and* have their state accessed in a thread-safe manner (either the object is thread-safe or it is guarded by a lock).

Chapter 4: Composing Objects

This chapter discusses how to compose classes in a thread-safe manner.

When designing a thread-safe class, it is important to know what state the class has and what invariants its state variables participate in. The state of an object includes the state of all objects in its fields. A thread-safe class should have a well-documented set of synchronization policies (what combination of locking, immutability, and thread confinement is used), both for maintainers and for people who use the class. Using encapsulation makes it much easier to reason about state and synchronization.

Many objects have conditions on the state as well as pre- and post-conditions on the state transitions (an example of the latter is that a counter can only transition from its count being n to its count being $n + 1$). In the case of state transition conditions, the check and then the transition need to be atomic. Sometimes in concurrent code, threads can wait for a pre-condition to become true. While it is possible to do this with `wait` and `notify`, it is better to use built-in library classes.

Sometimes objects are not considered part of a class' state because they are not "owned" by that class. For example, "ServletContext" might be storing objects for a thread that it doesn't own.

By encapsulating an object within another object and not publishing any references to it, we can confine access to the data within that object. This is another method we can use to access non-thread-safe objects in a thread-safe way. For example, one might wrap a `HashSet` in a class that only adds and checks containinment in synchronized blocks. One example of this in the Java libraries is `Collections.SynchronizedList`

The act of encapsulating all mutable state and guarding it with the object's intrinsic lock is known as the Java Monitor Pattern. This pattern has the side effect of exposing the lock used by the object to clients of the object, which can be either a disadvantage or an advantage depending on both the intended use of the object and the clients. This exposure can be avoided by using an internal lock.

Another way to create thread safety in a composite class is to delegate it to the published objects. For example, a class whose state consists of a concurrent hash map of immutable objects can publish an unmodifiable map view of that map (the fact that the objects in the map are immutable is crucial here). This is okay when each of your pieces of state is independent of the others (for example, several independent lists of listeners), but it fails if there are multiple pieces of state involved in certain invariants or that have invalid state transitions.

State variables can be safely published if they do not participate in any invariants that constrain their value, are thread-safe, and have no prohibited state transitions.

Adding functionality to already-existing classes can be tricky. The safest way to do so is to modify the original class (since the synchronization policy can remain in one place), but this isn't always possible. One way to do this is to extend the class, assuming it was designed for extension (see Effective Java). In doing so, one must make sure to acquire the correct lock when performing operations. The main downside of this method is that the synchronization policy is now spread over multiple classes. Furthermore, the original class might change its synchronization policy, which could affect the correctness of the extending class in subtle and bizarre ways.

Sometimes this doesn't work because we are trying to do things with a wrapper class (such as `SynchronizedList`). In this case, we can make a "helper" class, but we have to make sure that we acquire the correct lock on the object, rather than synchronizing on the helper class' intrinsic lock. This is the most fragile method so far, because the synchronization policy is now spread to a completely unrelated class.

A better way to do all of this is composition. When we compose a class, we make no assumptions on the thread-safety of our state variables and just delegate all of the methods that we might want, taking it on ourselves to make the class thread-safe.

The chapter then entreats the reader to always document synchronization policies and thread-safety. One good way to do the former is to use the `@GuardedBy` annotation. It also offers some advice to reading vague documentation, of which there is a lot. For example, sometimes it really would be absurd if a particular class were not thread-safe, so it may be a reasonable assumption that it is.

Chapter 5: Building Blocks

This chapter covers the various concurrency building blocks that are present in the Java libraries since Java 5.0 and Java 6, as well as best practices in using them.

We've already discussed the synchronized collections such as `Vector`, `Hashtable` and the `Collections.synchronizedXXX` classes that have been around in Java forever (since the beginning or since 1.2). These classes are all thread-safe because they synchronize every public method. These sorts of collections have several problems. The first problem is that compound operations require client-side locking in order to act as expected, even though they can't leave the underlying collection in an inconsistent state. For example, getting the size and then removing the last element, or iterating a collection, may cause the collection to actually be smaller than we expect by the time we perform the second operation. In the case of iteration, client-side locking isn't even that preferable, because it locks out the entire collection while we go through the whole collection, potentially doing an expensive operation. This leads to the second problem with the synchronized collections: lack of concurrency.

Many classes, including the synchronized classes, return iterators that are fail-fast if they notice that the underlying collection has been modified since iteration started (this is done by keeping a change count in the iterator and the collection). They are not *guaranteed* to fail if the collection is modified (for performance reasons), but when they do, they will throw a `ConcurrentModificationException`. The way to prevent this is to lock the collection while iterating and to only use `Iterator.remove()` to delete elements, or to copy tho collection and iterate the copy, since the copy will be thread-confined.

One pernicious thing about iterators is that many operations use "hidden" iterators. For example, `toString` on collections will often iterate over all of the elements. Thus, much like encapsulating state, it is often beneficial to encapsulate the synchronization for a class as well.

As an improvement on the synchronized collections, Java 5.0 added the concurrent collections. These collections often improve performance over their synchronized brethren due to the fact that they don't lock the entire collection for every operation. They also provide more support for compound operations.

`ConcurrentHashMap` uses a technique called lock striping to allow multiple readers to access the map at the same time, while a limited number of writers can access the map simultaneously. This allows for much better parallelization. The iterators from this class are "weakly consistent", which means that they traverse the elements of the collection as they were when the iterator was created and may reflect modifications to the collection since then, but are not guaranteed to. There are a few disadvantages of these classes. The first is that global calculations like `isEmpty` and `size` are not guaranteed to be correct soon after they returned, so they just function as estimates. This is okay since such collections are often changing quickly in multithreaded environments anyway, so these methods are not very useful. Furthermore, the concurrent collections do not support operations that need to lock the entire array. This is okay, however, because these collections support many of the compound operations we might care about anyway. In general, there are more advantages to these collections than disadvantages in a multithreaded environment.

`CopyOnWriteArrayList` and `CopyOnWriteArraySet` do what you might expect from the name, thus removing the need to lock the entire collection on iteration (among other things). Due to the copying on write, any references published by these collections are effectively immutable, so as long as they are properly published, they are thread-safe (as per the previous chapters). Iterators only need to synchronize to ensure visibility. There is clearly some cost with doing the copy, but these objects are very useful for cases where iteration is much more common than writes (such as iterating the list of registered listeners in an event handler).

Blocking queues are objects that allow threads to block on `put` operations (if the queue is bounded and full) or `take` operations (if the queue is empty). This enables a very powerful pattern known as the producer-consumer pattern. In this pattern, producer threads can create work and put it into a blocking queue that's monitored by a consumer thread. The advantage of this pattern is that neither producers nor consumers need to know anything about the other threads running. In some cases, a thread can be both a producer and a consumer, if there are multiple stages to an operation and multiple blocking queues. When using this pattern, it is important to be mindful of resource management to make sure that queues don't overfill, etc.

The Java library offers several implementations of `BlockingQueue`, each of which can be more or less helpful for various operations. One of them is `SynchronousQueue`, which never actually keeps a queue but instead blocks all threads until it can directly connect a putter and a taker.

The concept of serial thread confinement is that threads can pass ownership of mutable objects, thus ensuring that those objects are essentially thread-confined. An example of this is a connection pool, which hands a connection to a thread and expects that only that thread will use it.

Java 6 also adds an object called a `Deque`, which is essentially a double-sided queue. This object enables the practice of "work stealing," in which each thread has a work queue of its own, but threads that have an empty work queue can "steal" a task from the other end of another thread. This is a useful pattern in tasks where the consumers can potentially identify further work that needs to be done (in which case, they can add it to the end of other threads' work queues).

Some blocking methods can throw `InterruptedException`, which signals that they can be interrupted by other threads or the VM. This exception allows the thread to decide what it wants to do. In the vast majority of cases, your code will want to either propagate the exception or restore the interrupt by calling `interrupt` on the current thread. It is never acceptable to swallow the interrupt unless the code is extending `Thread` and thus control all of the code higher up on the call stack.

Blocking queues not only contain objects, they also control the flow of any threads using it. The Java class library contains other objects that exhibit this flow control property. These are known as "synchronizers". In general, these are objects whose state determines whether threads arriving at that object should be blocked or let through, as well as methods to manipulate that state.

Latches are synchronizers that delay threads reaching them until their terminal state is met. One example is a countdown latch, which starts at a positive number and counts down every time a thread reaches it. When it reaches zero, all threads are let through. Some examples of uses for latches are making sure that resources have been initialized before another service initializes and waiting for all players to be ready in a multi-player game.

`FutureTask` also acts as a latch. Each one contains a `Callable` and can be waiting to run, running or completed. When a thread asks for the result of a future task that is waiting to run or running, the thread blocks. When a thread asks for a completed one, it receives the answer immediately. These are very useful to represent tasks that will be run in parallel (the chapter has a rather lengthy example of how to use these to build a scalable and thread-safe result cache from a lengthy computation).

Semaphores are similar to latches except that a thread can also increase the count (the metaphor is that each thread takes a "permit" from the semaphore). Barriers are similar to latches except that all threads must be present for the barrier to release (as the book puts it, latches are waiting for events, whereas barriers are waiting for other threads). A `CyclicBarrier` allows a fixed number of threads to repeatedly wait on the same barrier over multiple steps. This is very useful for making sure an entire step of a parallel computation completes before the next step.

Part II: Structuring Concurrent Applications

Chapter 6: Task Execution

It is valuable to structure an application in terms of tasks, which are self-contained units of work that have their own state and can be kept separate from others. These are useful because we can have threads execute tasks and thus require minimal sharing of state, as well as providing for a very intuitive structure for an application.

As such, the best thing to think about when structuring concurrent applications is sensible task boundaries. In the case of a server, each task might be servicing a client connection. In the case of rendering a webpage, each image that needs to be downloaded can be its own tasks. Each task should represent a small amount of the capability of the machine that is running the server. Good choices of both tasks and task execution policies can lead to graceful running and degradation in the case of overload.

The first step to concurrency with tasks (from single-threaded execution) is to create a new thread for each task and run it immediately. This has some disadvantages when there are many tasks. First of all, each thread that is created and destroyed invokes some amount of overhead in the JVM and the OS. Second of all, each thread uses some amount of resources (such as memory), and too many threads can run out of memory. Finally, there are often limits on the number of threads that can be created, imposed by both the OS and the JVM, so the program might crash if this limit is hit. Thus, while threads give us a huge advantage, we'd like to limit the number of threads if the number of tasks becomes unbounded.

Enter the `Executor` framework. The interface for an Executor is really simple:

```
public interface Executor {
  void execute(Runnable command);
}
```

One executor may simply run all tasks in the same thread. Another one may simply create a new thread and run each task immediately. The most common executor is one that creates a fixed size thread pool and schedules tasks to available threads in the pool (and otherwise waits). The implementation of an executor is basically the implementation of a thread execution policy (what threads run a task, what order do they run in, how many tasks run at a time, when to queue tasks, when to cancel tasks, what to do if tasks are rejected, etc.)

This last executor is used in an extended example for writing a web server in the book. It provides all of the advantages of using threads without the disadvantages of having an unbounded number of them. The summary here is that code of the form `new Thread(runnable).start()` should alert you to the potential of using an executor instead.

Java also has an interface called `ExecutorService` which provides lifecycle methods on an executor, allowing us to cancel all waiting and running tasks and query whether the executor has shutdown or been terminated.

For periodic or deferred tasks, new code should use `ScheduledThreadPoolExecutor` instead of `Timer`, primarily because Timers behave poorly if a task runs too long or throws an unchecked exception. The book provides an example of code that exhibits this problem.

The next section describes how to find exploitable parallelization, and does a quick review of `Callable` and `Future`. The latter allows get to have a timeout, allowing one to call out to external services but produce a default value from any that take too long to return, for example. If they take too long, the calling thread gets a `TimeoutException`.

`CompletionService` is the combination of an Executor and a Blocking Queue, allowing one to block on the completion of threads in the executor.

Chapter 7: Cancellation and Shutdown

Sometimes we need or want a task or thread to complete before it is done with its job. However, we rarely want this to happen *immediately*, and in most cases this would be fairly disastrous since the task or thread could hold resources or locks that need to be cleaned up or made consistent. Therefore, the JVM provides the cooperative mechanism of interruption. Thus, tasks can be coded so that when interrupted, they clean up and exit.

There are several reasons why you'd want to cancel a task, including: user-requested cancellation, time-limited activities, application events, errors, and service shutdown.

A trivial way to request a thread be cancelled is by having a `volatile boolean` flag that can be set by other threads (through a method on the class that does the running) and have the main task loop check for this cancellation flag. This qualifies as a cancellation policy, or a description of how a thread must be cancelled and what it promises to do (and how quickly) in the event that it is. The problem with this policy is that the loop could be calling a long-running or blocking operation (such as taking from a `BlockingQueue`), so the time to cancellation may be longer than inteded or never! Thus, this policy is only good for tasks where each loop runs relatively quickly and doesn't block.

Enter interruption. Threads have a boolean interrupted status which can be set and signals to anything running in the thread that it should probably stop what it's doing whenever it can. There is nothing that ties using interruption to cancellation, but in practice it is extremely fragile to use it for anything else. Library methods that block will try to detect thread interruption and throw an `InterruptedException` if they do. There is no guarantee on how quickly this will happen, but it happens quickly in practice. Interruption is the best way to implement cancellation.

The static method `interrupted` should be used with caution, as it clears the current interrupted status of the thread. If this method returns true, the calling code should either throw `InterruptedException` or restore the interrupted state of the thread, unless it was planning to swallow the interruption.

In addition to the notion of a cancellation policy, threads should also have an interruption policy. This policy determines what the thread actually does in case of an interruption, what work will be completed, and how quickly it will react. In most cases, the best solution is to exit as soon as possible, cleaning up aloqng the way. If a non-standard policy is used, the entire service needs to be written with awareness of this policy.

The reason tasks shouldn't swallow interruptions or mess with the thread is that they may not be the only thing running on that thread. It is up to the task that owns the thread to determine whether to kill a thread, for example, and that involves the interrupted status reaching the right places.

For a timed run, don't schedule an interrupt on a borrowed thread! This could result in cancelling the wrong task. Instead, it's best to use the `Future` framework.

Some blocking tasks are not interruptible, but there are ways to get around it, which the book goes into. Fortunately, for non-standard cancellation methods, the `ThreadPoolExecutor` class offers a `newTaskFor` hook, which allows you to define a custom thread interrupt method for the executor.

Thread-owning services should provide lifecycle methods if they will live longer than the method which created them. One way to do this is with the `ExecutorService` shutdown. Another way to do this (for e.g. services that have threads reading from a work queue) is to create a "poison pill". This is a sentinel value that signals to the thread that its work is done and it should shut down.

Threads may sometimes exit abnormally (e.g. with a `RuntimeException`. In long-running applications, there should be an `UncaughtExceptionHandler` that at least logs the exception. Such a handler is called, as you might expect, when there is an uncaught exception.

The JVM can shutdown in an orderly or abrupt manner. In an orderly shutdown, the JVM calls all registered shutdown hooks (registered with `Runtime.addShutdownHook`), which are usually used to clean up resources. There is no guarantee on which order the hooks are called and they must be thread safe.

Some threads mark themselves as daemon threads. These are threads whose continuing to run does not prevent the JVM from shutting down. These should be used sparringly, but are sometimes very useful.

Avoid finalizers (see Effective Java).

Chapter 8: Applying Thread Pools

This chapter discusses how to write programs that utilize thread pools for various classes of tasks and certain issues to watch out for.

Sometimes the layout of tasks has to dictate certain things about the execution policy of a thread pool. Tasks can depend on each other, which places constraints on the execution policy to ensure there aren't liveness problems. Tasks can exploit thread confinement, which can mean that the executor should be single-threaded. Tasks can be sensitive to response time, which would dictate the size of the thread pool and number of long-running tasks that can go into it. Finally, tasks can use `ThreadLocal`, which can be problematic if threads are re-used (so tasks should never count on using anything inside the `ThreadLocal` beyond the boundaries of the task).

As stated before, the best case for thread pools is when tasks are homogenous and independent. Usually requests in web servers tend to meet these guidelines, but not all sets of tasks work like this, which means we need to make sure to document all requirements and make sure our thread pools are designed well for said tasks.

Dependent tasks can deadlock if the depending task gets scheduled first and the thread pool is too small. This can happen if tasks can submit further tasks to the pool and then wait on them. In general, it's best to avoid this sort of workflow, but if it's necessary then any pool sizing or configuration constraints should be clearly documented wherever the Executor is configured. There may also be implicit bounds on the size of thread pools if other limited pools are used, like a JDBC connection pool.

Long-running tasks can cause thread starvation if they "clog up" the thread pool. One way to mitigate this is to use timed resource waits instead of unbounded waits (something that is available in most of the Java concurrency classes) and then fail if the wait times out.

Sizing thread pools is a little bit of an art, since one wants to avoid making them too big (too much overhead in scheduling threads, too much memory usage) or too small (thread starvation, concurrency suffers). The book offers the following as a nice formula for how many threads to run:

$$N_{threads} = N_{cpu} * U_{cpu} * \left(1 + \frac{W}{C}\right)$$

where N_{cpu} is the number of CPUs available, U_{cpu} is the target CPU utilization (between 0 and 1) and $\frac{W}{C}$ is the ratio of wait time to compute time.

`ThreadPoolExecutor` is the base implementation for many of the executor factories found in `Executors`. It provides a constructor with various configuration objects (which the factories set with a particular value). One major configuration point is their saturation policy, or what they do when they get too many tasks. Unbounded task creation can cause memory issues just like (albeit slower than) unbounded thread creation. One part of the way this policy is specified is by giving the `ThreadPoolExecutor` constructor a `BlockingQueue` that implements the policy (having a FIFO queue, a priority queue, or a synchronous queue).

The other part of this is to set the `RejectedExecutionHandler`. Some default ones are `AbortPolicy` (makes the execute method throw `RejectedExecutionException`), `CallerRunsPolicy` (the executor, when full, tries to run the task in the calling thread), `DiscardPolicy` (tasks that exceed the limit are just thrown away), and `DiscardOldestPolicy` (the next task to be executed is thrown away). Some policies should not be combined with some types of blocking queues (e.g. the `DiscardOldestPolicy` with a priority queue). There is no way to make

execute block when the queue is full, but the same effect can be had with a semaphore.

Executors can also specify a thread factory, which can do certain things upon thread creation like giving the thread a name.

Executors have methods to configure them after they are created, but if a non-configurable pool is desired, one can use the `unconfigurableExecutorService` method to get one that does not make these methods available.

`ThreadPoolExecutor` was designed for extension, so it can be subclassed to extend. For example, special functionality can be made to run before execution, after execution, and upon termination (such as logging).

If a loop has iterations that don't depend on each other, then it is easy to use a thread pool to paralellize that loop. If an algorithm is recursive but each recursive call does not require the results of its children, it can submit the calls to the executor. The book demonstrates doing this to solve a puzzle by submitting each mave sequence to an executor and stopping the pool when a solution is found.

Chapter 9: GUI Applications

GUI applications tend to be single-threaded. The reason for this is that events typically bubble up from the OS, through the GUI layer, and into the application, and then responses come back the other way. Furthermore, most front-end applications are structured as models, views and controllers. This makes it very difficult to avoid deadlock. Therefore, most GUI frameworks rely on thread confinement for correctness.

In Swing, for example, pretty much the only things that are allowed to be touched from threads outside the event dispatch thread are methods that schedule tasks into the event dispatch thread. Short running GUI tasks can run in this thread. Long-running ones can make the UI unresponsive and should dispatch out to executors that are using other threads. The various lifecycle methods we've learned about can be used to get the results or to cancel these tasks.

State can be shared in several ways. It can be kept entirely in the GUI's model objects in-thread. It can be kept in thread-safe containers. Or it can be split, with the GUI models acting as a view into the state.

Other subsystems may work single-threadedly and look a lot like GUI frameworks.

Part III: Liveness, Performance, and Testing

Chapter 10: Avoiding Liveness Hazards

This chapter discusses the various ways in which concurrent programs can suffer from liveness issues.

The first way is deadlock. The book discusses the Dining Philosophers Problem, in which a group of five philosophers around a table suffer from deadlock. In programs, this can happen when two threads need the same two locks but acquire them in a different order. This can cause both threads to stay waiting forever. Since Java applications have no built-in way to recover from deadlock, this can cause programs to become permanently unresponsive. Unlike Java, database systems will detect deadlocks and abort an arbitrary transaction to fix a deadlock. Deadlocks rarely manifest deterministically or quickly, so your application may have hidden ones.

Lock-ordering deadlocks are those that occur due to threads acquiring locks in different orders, as above. These can be avoided by setting a prescribed order on all locks and making sure threads acquire them in the same order. In order to do this, one usually has to globally find everywhere that multiple locks are acquired, as locks don't always have a natural ordering. Sometimes locking is based on arguments that are passed in, so an ordering must

be induced. For example, if your method transfers money between bank accounts, it is not sufficient to always acquire the lock on the "from" account first. Inducing an ordering can be done by locking on things that implement `Comparable<T>` or by using `System.identityHashCode`.

Multiple lock acquisition can sometimes happen across two objects (when one object calls into another). In this case, the best one can do is to avoid calling into another object with a lock held (as the book puts it, that is "asking for liveness trouble"). Calling into another object with a lock held can violate several encapsulation principles of object oriented programming, since it requires you to know which locks are obtained by the method you're calling. A call into another object with no locks held is called an "open lock" and is to be preferred, as it simplifies the liveness analysis of the program.

Sometimes this is not possible, though, as it turns a previously atomic operation into a non-atomic one. In this case, the object should be restructured so that the code path following the open call can only be executed by one thread at a time. Usually this can be accomplished by setting some flag that other threads can check.

In addition to deadlock while waiting for locks, threads can deadlock on resources if they try to acquire multiple pooled resources that are limited, such as database connections. Applications can also experience thread starvation deadlock, as discussed in previous chapters.

The best way to avoid deadlock is to make the number of code paths that must acquire multiple locks as small as possible. Then, analyze the program to find anywhere where multiple locks must be held and make sure that locks are always acquired in the same order. With no non-open calls, there are automated tools that can analyze the program's source to find instances of multiple locks being acquired.

One way to recover from deadlock is to use timed lock attempts (on non-intrinsic locks) and allow the code path to fail if it cannot acquire the lock.

Deadlocks can be analyzed with thread dumps. A thread dump can be attained by sending the JVM a SIGQUIT signal or by using jstack. Java 6 and above have deadlock detection in the thread dump, which is very useful.

There are several other liveness hazards. The first is starvation, where a thread perpetually fails to get access to resources it needs, such as CPU cycles (which can be caused by manually setting thread priorities). It is not recommended in general to tweak thread priorities manually. The second is poor responsiveness, which can happen most commonly in GUI applications. It can also happen if an application has poor lock management. The third is livelock, where the program is still technically doing work, but no useful work is accomplished. An example of this is a work queue that replaces the task on the queue if it fails, which has one task that fails every time it is executed. Another example is two threads trying to send packets on a shared carrier simultaneously but perpetually trying to back out of the other's way. A fix for the latter case is to introduce some randomness into the retry time.

Chapter 11: Performance and Scalability

This chapter discusses how to analyze and improve the performance of concurrent programs. It cautions that safety should always come first and performance improvements should only be undertaken if they are absolutely necessary.

Performance means doing more work with fewer resources, for various definitions of resources. We often care about what the resource bottleneck is for a program (we say it is bound by that resource). Using threads introduces certain performance overhead (the cost of setting up threads, context switches, and using locks), but using them correctly should more than make up for it. However, not using them correctly can hurt performance a lot. The end goal is to make sure that all CPUs on a system are always busy with useful work (because this means that there is enough to go around).

Scalability is another side of this picture, and looks at how much throughput or capacity can be improved if additional resources are added. Instead of doing the same work with less effort, we are interested in doing more work with more resources. For example, a program that only ever runs four threads could be keeping four CPUs hot but would not get any better on an eight-core system (this being a contrived example, of course). Sometimes

performance and scalability are at odds with one another, as designing a scalable system may be less performant in the small N case.

Avoid premature optimizations and always use data when determining when to make something faster. Determine what your common conditions are, what is slow (and why) under those circumstances, and what good performance actually means.

Amdahl's law provides an equation for determining the maximum amount of concurrency speedup in a program:

$$S \leq \frac{1}{F + \frac{1 - F}{N}}$$

where S is the speedup, F is the fraction of the computation that is run serially, and N is the number of processors. Serial computation arises due to things like locks and putting work onto a work queue (as such, F is pretty much never 0). Sometimes the serial computation can be hidden in frameworks. Thinking about Amdahl's law qualitatively in the limit as N gets large can be useful for analyzing the scalability of programs. One way to do this is by plotting throughput against the number of threads used for various techniques of doing things.

Threads introduce certain costs into a computation. The first is context switching, which is incurred every time a thread is pre-empted or blocks due to waiting for a lock. Another is memory synchronization, which happens due to keywords like `synchronized` and `volatile`, as this requires doing certain flushes and invalidation of caches. Furthermore, memory barriers can prevent certain compiler optimizations.

Uncontended synchronization (where a lock is immediately taken instead of blocking the thread) is much faster than contended synchronization, because the mechanism is optimized for this case. Furthermore, uncontended synchronization does not require any OS-level context switching or management. Modern JVMs can actually optimize out synchronization that they can prove will never be contended. They can also change multiple consecutive lock acquisitions into one big block with one acquisition (for example, when adding multiple elements to a synchronized collection with one lock). As such, there's not much use in worrying too much about the cost of uncontended synchronization. However, it is very useful to optimize code paths where lots of lock contention occurs.

While locks are absolutely necessary for safety, exclusive resource locks are the biggest threat to scalability. There are three ways to reduce lock contention: reduce the duration for which locks are held (only do the operations you need the lock for while holding it), reduce how frequently locks are requested, or replace locks with other coordination mechanisms.

Locks can be split up so that each lock guards separate independent variables (for example, if there are two fields in a class that do not participate in any invariants, we can use each individual field's intrinsic lock instead of the whole class's lock). These theme can be expanded into a technique known as lock striping. An example of this is `ConcurrentHashMap`, which has 16 locks, each of which guard about 1/16th of the keys (and therefore allowing 16 concurrent accesses to the map rather than just one). One of the downsides to this method is that this makes operations which affect the whole collection more difficult.

Another technique to avoid lock contention is to avoid fields that multiple threads will want to access. For example, if our `ConcurrentHashMap` wanted to keep a count of elements so that other threads could get the size in $O(1)$, then this would defeat the purpose of lock striping. In this case, we could have each of the 16 buckets keep their own count. This is actually a good example of performance (wherein we keep the one count so that the size method is $O(1)$ instead of $O(n)$) is at odds with scalability (wherein a "hot" field affects concurrency).

Finally, lock contention can be avoided by avoiding exclusive locks altogether. This can be done by using concurrent collections, read-write locks, immutable objects, or atomic variables (the last of which are optimized to use processor primitives, so they are very fast).

There are performance monitoring tools (`vmstat` and `mpstat` on Unix or `perfmon` on Windows) that can show how hot the CPUs are running. If CPUs are not being fully utilized, there can be several reasons for this. There can be insufficient load, the application can be I/O-bound, the application can be externally bound (e.g. waiting on a

database server much of the time), or there might be lock contention. Java profiling tools can help with determining if there is lock contention.

In general and for various reasons, keeping object pools is bad. Objects are cheap and locally created objects improve cache hitting and reduce the need to take out locks on the object pool.

The book shows an example for how `ConcurrentHashMap` has much better throughput than `synchronized HashMap`s with large numbers of threads. It also walks through an example for how to reduce context switching overhead when coding a logging service. Offloading all logging to one thread that pulls from a queue reduces the number of threads that have to wait on I/O in an application.

Chapter 12: Testing Concurrent Programs

It is difficult to test concurrent programs, since many concurrency failures are probabilistic, depending on the timing of events. Most concurrency testing comes in one of two categories: safety and liveness. The former usually involves testing invariants. This is hard since the test code can have races or its own synchronization that affects timing. For liveness, the challenges involve both deciding what to test (throughput, responsiveness, scalability, something else?) and deciding on whether code is running slowly or permanently deadlocked.

Correctness tests can have some basic unit tests (like initializing a new Queue and checking that it reports that it is empty). These can, in most cases, be entirely sequential.

Tests that have to be concurrent are a bit more tricky, since most test frameworks don't provide an easy way to spawn threads and check up on them or have them report back when something failed. the JSR 166 Expert Group solved this problem by creating a base class that provided methods to do many of these things. Testing that a thread blocks should work like testing for an exception being thrown – the test should fail if the thread reaches the end. However, it is hard to keep a blocked thread from holding up the test forever. This can be done by writing code that responds to interruptions and interrupting the thread that should be blocking. A difficult decision is deciding how much time should pass before interrupting the thread. The main runner thread should use a timed join on any threads it spawns and joins to, in case a thread gets stuck on something.

Do not use `Thread.getState` to check if a thread blocked. The JVM is allowed to spin wait threads that block (which leaves them in the `RUNNABLE` state), and there are other cases where the thread might not change status.

Tests for concurrent correctness should be designed to unearth problems with high probability without themselves causing problems or changes in behavior. This means that tester threads should try to update as little global state (which requires synchronization) as possible. It's best to keep thread-local state and combine at the end (or at checkpoints for checking that the state of your class is correct).

Test data often needs to be generated randomly, but many RNGs are thread-safe, which introduces more synchronization. Instead, the book provides a medium-quality (not cryptographically secure) random method that does well for testing.

Depending on the platform, creating and starting threads may be a heavyweight operation, so the first threads might get a huge headstart on running the test, decreasing the amount of interleaving that occurs in the test. One fix for this is to use a `CyclicBarrier` to ensure all the threads are ready before starting. Tests should run on systems that have multiple CPUs and should run with more threads than CPUs.

Resource management (in particular, memory leaks) can be tested by using a heap profiler tool at certain parts during testing, such as taking a test that allocates a huge object and then clears it, then testing that the heap is not too much bigger than it was before the allocation.

Callbacks can be used to test certain aspects of thread pools and thread creation, by creating a custom Thread or ThreadFactory.

In general, we want to increase the interleavings of thread computation as much as possible. One way to do this is to call `Thread.yield` liberally. Since JVMs can optimize this call into a NOOP, another (albeit slower) method is to call `Thread.sleep` with a small, but nonzero number. Sometimes, we want to put these calls into the middle of a method in the code, in which case an AOP tool might help.

Performance testing is another thing that we often want to do, which comes with its own bag of techniques and caveats. One important aspect of performance testing is to choose a set of representative scenarios for your class and try to replicate them in the code, which is not always easy. This includes approximating the same amount of CPU, IO, and other resource usage in the thread.

The book has a running example of a concurrent test, which it extends in this section to add timing information to the `CyclicBarrier` that it uses. This gives us the amount of time all threads take to complete, which we can divide by the number of threads to find the average per-thread runtime. Trying to time each thread individually can be tricky, since threads can often complete in a faster time than the timer tick. This technique gives us a measure of throughput.

For responsiveness, we can change our test runner to batch a small number of threads and measure the batch (which can get around the timer tick issue) and histogram the approximate thread completion times. This gives us a good idea of the variance of our task completion time. Sometimes, we have to make decisions about accepting a higher mean response time for a lower variance.

There are a number of performance testing pitfalls. The first is garbage collection: Sometimes N iterations of a test do not trigger a garbage collection, while N+1 do. One solution is to call the JVM with `-verbose: gc` to see if Garbage Collection is called during the test and tweak the test to ensure it isn't. The downside to this approach is that removing garbage collection isn't a realistic condition for the program's actual run. Alternatively, the test writer can make sure that garbage collection runs a number of times during the test, by making the test longer.

Another pitfall is dynamic compilation. Sometimes the JVM will decide that a codepath is used often enough that it should compile the given code (instead of interpreting it on every run). It may take a variable amount of time for it to do this. Once again, the best way to get around this is by making the test long enough that compilation always occurs and that runs which happened the compilation (as well as time to compile the code) account for a small percentage of the runtime. Also, the test can perform a 'warmup' pass to make sure that all the code is compiled for the actual timed portion of the test. The JVM can be run with `-XX:+PrintCompilation` to see when compilation occurs to verify that the warmup run was long enough.

Due to optimizations in the JVM (like compilation and dead code elimination), it is important that the code paths used in the test accurately reflect those that would be used in real life, in the same amounts and orders. As stated before, resource usage should also be approximately the same to simulate the contention well (for example, a task that uses a synchronized collection and a lot of CPU intensive work may be okay with the amount of contention on the collection). To prevent dead code elimination, test code should always use results obtained (even if it is to do an empty print in rare cases).

Automated testing can't find *all* of the bugs present in your program. As such there are two complementary testing approaches that should be used: thorough code reviews from other members of your team (no matter how good at concurrency you are, you might miss something), and static analysis tools. The latter have evolved quite a bit to find some common sort of bugs just by examining the uncompiled code. In addition, Aspect-oriented programming (AOP) tools and profilers/monitoring tools can help quite a bit.

Chapter 13: Explicit Locks

Java before 5.0 only had the `synchronized` and `volatile` keywords for coordinating access to shared data. Java 5.0 added the `Lock` interface and `ReentrantLock` class. The latter is not a replacement for the `synchronized` keyword but rather an alternative that supports extra features and has its own drawbacks.

The `Lock` interface provides the ability to lock, lock interruptibly, or lock with a time-based time-out. Using a `Lock`

is supposed to provide the same memory semantics as intrinsic locks. `ReentrantLock` works in the same way as a `synchronized` block. When using an explicit lock, the coder must absolutely make sure to execute the protected code in a `try` block and put the unlock statement into an associated `finally` block. Code that does not do this is incorrent, since explicit locks do not automatically unlock. This is one reason why `synchronized` should be used when possible.

Polled and timed lock acquisition is useful for tasks that have a fixed time budget. Interruptible lock acquisition is useful for tasks that want to be interruptible, since acquiring an intrinsic lock is not interruptible. The downside to this is that two try blocks are required in this idiom.

Sometimes lock idioms may be desired that do not fit with block-based code structuring (for example, the "hand-over-hand locking" used when we assign each node of a linked list its own lock and acquire the next one before releasing the previous one).

In Java 5, `ReentrantLock` performed much better than `synchronized` blocks, but this is no longer true in Java 6. This illustrates the fact that benchmarks and performance may be a moving target.

Explicit locks may offer a choice of fairness options. Fair locks always schedule threads in the order that they request the locks. Unfair threads allow "barging", in which a thread can jump ahead of the queue if the lock is available when that thread requests it. Fair locks would just queue this thread. Unfair locks are much faster (since the cost of context switching and scheduling a new thread is pretty high), and typically offer a weaker guarantee that a thread which requests a lock will eventually obtain it (and this is good enough in practice). Fair locks work best if they are held for a long time or are requested infrequently.

In summary, `ReentrantLock` should pretty much only be used when its advanced features are needed, because it is harder to analyze programs that use explicit locks (for example, to make sure that all locks are released and that there are no deadlock risks). The compiler may also understand `synchronized` blocks better for optimizations.

Read-write locks are another type of lock that can provide performance benefits for "read mostly" data structures. These locks allow multiple readers at a time but only one writer, though they may differ in some of their semantics (such as whether or not threads are allowed to up- or downgrade their locks, whether or not the locks are reentrant, etc).

Chapter 14: Building Custom Synchronizers

Many library classes contain operations that operate with preconditions based on state, such as `FutureTask`, `Semaphore`, and `BlockingQueue`. It is usually best to use these library classes when such operations are needed, but sometimes special behavior is needed that cannot be fulfilled by library classes. This chapter covers how to build custom synchronizers to build such classes.

The book goes through several examples of implementing a bounded buffer (with blocking put and take operations) to illustrate the challenges of managing state. The first example propagates failure to callers through exceptions. The problem with this example is that using the buffer is very difficult, due to the exception handling that is necessary. The buffer could also return some error sentinel value, which would require the client to spin until the precondition was met.

The second example is to poll the buffer and sleep if the precondition isn't met. The problem here is that the thread could "oversleep" the condition becoming true.

The best solution to this problem is condition queues. Basically, a condition queue is a collection of threads that are all waiting for some condition to become true before they can continue executing. Much like locks, every Java object has an intrinsic condition queue, which is used by calling the methods `wait`, `notify`, and `notifyAll`. In order to call any of these methods, a thread must hold that object's intrinsic lock. Calling `wait` releases the lock and reacquires it upon waking. The book's final example of the queue uses condition queues.

When using condition queues, the predicate (the condition which threads are waiting on) should be very clearly documented. Furthermore, the lock that is associated with the queue must guard all variables involved in that predicate.

Threads calling `wait` should always test the predicate beforehand, and after waking. The call should be done in a loop. The reason for these is that `wait` may return without `notify` being called, or another thread may have made the predicate false after having been woken by the same `notify`. Not testing the predicate before waiting may result in a missed signal if the predicate is already true, causing the thread to potentially wait forever.

Program writers should make sure that a notification will actually occur anytime the predicate becomes true. Usually, notifiers should call `notifyAll`, but calling a single `notify` could have performance benefits. However, it is only correct to call a single `notify` when the following conditions are met: all waiters are waiting on the same predicate and a notification allows at most one thread to proceed. This is an optimization, and thus follows all the same rules as other optimizations.

Classes which are designed for subclassing need to explicitly document and expose their waiting predicates and behavior, so that subclasses may make modifications where necessary. Otherwise, condition queues should be encapsulated, which requires the use of a private lock.

Much like explicit locks, there are explicit condition queues, which can be nice when separating multiple predicates into multiple queues. These extend from the interface `Condition` and expose the methods `await`, `signal`, and `signalAll`. Note that, as objects, these also have the methods `wait`, `notify` and `notifyAll`, and that the proper methods should be used.

The book then introduces the `AbstractQueueSynchronizer`, on which many of the concurrency primitives in the concurrency package are based. In most cases, developers will never use the AQS directly, but they might if they wish to create their own primitives. The AQS generalizes the idea of having some state and how that state changes when threads "acquire" and "release". The book then shows how some common primitives are implemented in terms of AQS.

Chapter 15: Atomic Variables and Nonblocking Synchronization

Non-blocking algorithms are those that use low-level processor instructions like "compare-and-swap" in order to avoid needing to use locks. Such algorithms are difficult to design and reason about, but they can offer a large performance boost over locking in certain situations, due to decreased need for thread scheduling overhead. The disadvantages of locking have been discussed many times throughout the book, but a summary appears in this chapter.

Compare-and-swap is an operation that the book calls "optimistic" (in contrast to the "pessimistic" approach of locking), in that it assumes that the operation will work but is able to detect collisions. The general use case is to get the current value of the variable, compute an update, and try to write it (using the CAS operation) then see if the write proceeded without collision. If not, we loop and try the operation again.

The JVM exposes this functionality through the `Atomic*` classes, like `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, and `AtomicReference`. These classes can be used as "better volatiles" in that they have the same memory read/write semantics as volatile variables with additional atomic update guarantees.

The performance of using such variables is better than locks in cases where there is moderate contention (a fair amount of thread-local computation) but is worse than locks where there is high contention (a low amount of thread-local computation). The latter rarely happens in practice, though, so it is often useful to consider using these classes where possible.

Algorithms are called "nonblocking" if no thread's failure or suspension can cause another thread to fail or suspend. They are called "lock-free" if at any step, some thread can make progress. If a program uses CAS exclusively for coordination, it can be both, if it is structured correctly. The book goes through a few examples: nonblocking

stacks and linkedLists. The latter is harder, because each update requires updating two variables, so a more complex algorithm (that is more difficult to reason about) is put forth.

The book also mentions atomic field updaters, which can use CAS to set a field in a class through reflection. These offer the same guarantees as the CAS classes, but only if the updater is the only way used to update said field.

The way that threads know if there was contention in their update is that the CAS method returns the value that was in the variable when the update was requested. If the value is the same as it was when we first checked, it is assumed that no thread updated the value first. However, two threads may have updated the value, the second one updating it back to the original one, since we checked. In many algorithms, this doesn't matter, but sometimes it does. This is termed the "ABA problem". In such cases, it may become necessary to apply a "version number" on the value of the variable, which Java provides in `AtomicStampedReference`.

Chapter 16: The Java Memory Model

This chapter covers the low-level details of the Java Memory Model. The memory model is the thing that specifies the conditions under which threads see updates to variables that may have occurred in other threads. The reason this is a non-trivial question is that the compiler is allowed to re-order instructions when it compiles in order to increase performance (which it does by a great amount). Furthermore, memory reads and writes (and then local cache flushes and refreshes) can occur somewhat arbitrarily. In a single-threaded program, this isn't a problem, because the semantics of program execution as specified by the language specification require that everything occurs as if the instructions are executed in program order strictly serially if it's in the same thread.

Platforms provide their own guarantees about how caches and CPU-local stuff work, but the Java Memory Model is intended to abstract that away, so that Java programs behave the same regardless of platform. Earlier, we learned rules for how to make things correct given how the memory model works, but if we can get a good understanding of the model itself, then the rules can be more intuitive.

The Java Memory Model is defined in terms of actions (things like memory reads/writes, locking/unlocking, and others) and a partial ordering on those actions called "happens-before". The guarantee it makes is that if action A happens-before action B, then all memory effects that result from A can be seen when a thread executes action B.

The guarantees are the following:

- Every action in the same thread earlier in the program order happen-before any action that is later.
- Unlocks on monitor locks happen-before any subsequent locks (synchronization actions are totally ordered).
- A write to a `volatile` variable happens-before any subsequent read from it.
- A call to `Thread.start` happens-before any action in the thread.
- All actions in a thread happen-before any other thread detects that it has terminated.
- A call to interrupt a thread happen-before the thread detects the interrupt.
- All actions in a constructor happen-before any actions in the finalizer for that object.
- Happens-before is a transitive relation.

We can piggyback off of these guarantees to ensure execution order. For example, we can combine the program order rule with the volatile read rule to ensure that a method always happens-before another. This can save time spent on synchronization. It is used in FutureTask, for example, to ensure that `set` calls happen-before `get` calls by using existing guarantees from AQS. On the other hand, this is a fairly fragile technique and needs to be undertaken carefully. However, it is acceptable when using classes that commit to a happens-before ordering between methods.

We can also use the happens-before guarantees to see how safe publication works. For example, the unsafe lazy initialization that was visited earlier is unsafe not only because of its race condition, but because the publication to the new reference is not ensured to happen-before any writes to the fields of the object in the constructor. As such, other threads can see a partially constructed object. The safe publication techniques explored before are a way to ensure happens-before happens when we need it. In fact, we get the stronger guarantees given by happens-before of having visibility on all things that happened-before the object we published.

The chapter explores the double-checked locking anti-pattern and shows how it fails to solve the problem of seeing a partially constructed object.

The JVM also provides a guarantee of initialization safety which says that for properly constructed objects, other threads will always see the correct values of final fields, and any variables that can be reached only through final fields of a properly constructed objects are guaranteed to be visible to thore threads.

Appendix A: Annotations for concurrency

This short appendix describes some annotations for better thread-safety documentation of code. I believe many of these and more can be found in the JSR-305 package.