13 Concurrency

by Brett L. Schuchert

“Objects are abstractions of processing. Threads are abstractions of schedule.”

—James O. Coplien[1]

Writing clean concurrent programs is hard—very hard. It is much easier to write code that executes in a single thread. It is also easy to write multithreaded code that looks fine on the surface but is broken at a deeper level. Such code works fine until the system is placed under stress.

In this chapter we discuss the need for concurrent programming, and the difficulties it presents. We then present several recommendations for dealing with those difficulties, and writing clean concurrent code. Finally, we conclude with issues related to testing concurrent code.

Clean Concurrency is a complex topic, worthy of a book by itself. Our strategy in *this* book is to present an overview here and provide a more detailed tutorial in "Concurrency II" on page 317. If you are just curious about concurrency, then this chapter will suffice for you now. If you have a need to understand concurrency at a deeper level, then you should read through the tutorial as well.

Why Concurrency?

Concurrency is a decoupling strategy. It helps us decouple *what* gets done from *when* it gets done. In single-threaded applications *what* and *when* are so strongly coupled that the state of the entire application can often be determined by looking at the stack backtrace. A programmer who debugs such a system can set a breakpoint, or a sequence of breakpoints, and *know* the state of the system by which breakpoints are hit.

Decoupling *what* from *when* can dramatically improve both the throughput and structures of an application. From a structural point of view the application looks like many little collaborating computers rather than one big main loop. This can make the system easier to understand and offers some powerful ways to separate concerns.

Consider, for example, the standard "Servlet" model of Web applications. These systems run under the umbrella of a Web or EJB container that *partially* manages concurrency for you. The servlets are executed asynchronously whenever Web requests come in. The servlet programmer does not have to manage all the incoming requests. *In principle*, each servlet execution lives in its own little world and is decoupled from all the other servlet executions.

Of course if it were that easy, this chapter wouldn't be necessary. In fact, the decoupling provided by Web containers is far less than perfect. Servlet programmers have to be very aware, and very careful, to make sure their concurrent programs are correct. Still, the structural benefits of the servlet model are significant.

But structure is not the only motive for adopting concurrency. Some systems have response time and throughput constraints that require hand-coded concurrent solutions. For example, consider a single-threaded information aggregator that acquires information from many different

Web sites and merges that information into a daily summary. Because this system is single threaded, it hits each Web site in turn, always finishing one before starting the next. The daily run needs to execute in less than 24 hours. However, as more and more Web sites are added, the time grows until it takes more than 24 hours to gather all the data. The single-thread involves a lot of waiting at Web sockets for I/O to complete. We could improve the performance by using a multithreaded algorithm that hits more than one Web site at a time.

Or consider a system that handles one user at a time and requires only one second of time per user. This system is fairly responsive for a few users, but as the number of users increases, the system's response time increases. No user wants to get in line behind 150 others! We could improve the response time of this system by handling many users concurrently.

Or consider a system that interprets large data sets but can only give a complete solution after processing all of them. Perhaps each data set could be processed on a different computer, so that many data sets are being processed in parallel.

Myths and Misconceptions

And so there are compelling reasons to adopt concurrency. However, as we said before, concurrency is *hard*. If you aren't very careful, you can create some very nasty situations. Consider these common myths and misconceptions:

- *Concurrency always improves performance.*
  Concurrency can *sometimes* improve performance, but only when there is a lot of wait time that can be shared between multiple threads or multiple processors. Neither situation is trivial.

- *Design does not change when writing concurrent programs.*
  In fact, the design of a concurrent algorithm can be remarkably different from the design of a single-threaded system. The decoupling of *what* from *when* usually has a huge effect on the structure of the system.

- *Understanding concurrency issues is not important when working with a container such as a Web or EJB container.*

In fact, you'd better know just what your container is doing and how to guard against the issues of concurrent update and deadlock described later in this chapter.

Here are a few more balanced sound bites regarding writing concurrent software:

• *Concurrency incurs some overhead*, both in performance as well as writing additional code.

• *Correct concurrency is complex*, even for simple problems.

• *Concurrency bugs aren't usually repeatable,* so they are often ignored as one-offs[2] instead of the true defects they are.

• *Concurrency often requires a fundamental change in design strategy*.

Challenges

What makes concurrent programming so difficult? Consider the following trivial class:

```
public class X {
private int lastIdUsed;

public int getNextId() {
    return ++lastIdUsed;
 }
}
```

Let's say we create an instance of `X`, set the `lastIdUsed` field to 42, and then share the instance between two threads. Now suppose that both of those threads call the method `getNextId();` there are three possible outcomes:

• Thread one gets the value 43, thread two gets the value 44, `lastIdUsed` is 44.

• Thread one gets the value 44, thread two gets the value 43, `lastIdUsed` is 44.

- Thread one gets the value 43, thread two gets the value 43, `lastIdUsed` is 43.

The surprising third result[3] occurs when the two threads step on each other. This happens because there are many possible paths that the two threads can take through that one line of Java code, and some of those paths generate incorrect results. How many different paths are there? To really answer that question, we need to understand what the Just-In-Time Compiler does with the generated byte-code, and understand what the Java memory model considers to be atomic.

A quick answer, working with just the generated byte-code, is that there are 12,870 different possible execution paths[4] for those two threads executing within the `getNextId` method. If the type of `lastIdUsed` is changed from `int` to `long`, the number of possible paths increases to 2,704,156. Of course most of those paths generate valid results. The problem is that *some of them don't*.

Concurrency Defense Principles

What follows is a series of principles and techniques for defending your systems from the problems of concurrent code.

Single Responsibility Principle

The SRP[5] states that a given method/class/component should have a single reason to change. Concurrency design is complex enough to be a reason to change in it's own right and therefore deserves to be separated from the rest of the code. Unfortunately, it is all too common for concurrency implementation details to be embedded directly into other production code. Here are a few things to consider:

- *Concurrency-related code has its own life cycle of development*, change, and tuning.
- *Concurrency-related code has its own challenges*, which are different from and often more difficult than nonconcurrency-related code.

- The number of ways in which miswritten concurrency-based code can fail makes it challenging enough without the added burden of surrounding application code.

Recommendation: *Keep your concurrency-related code separate from other code.*[6]

Corollary: Limit the Scope of Data

As we saw, two threads modifying the same field of a shared object can interfere with each other, causing unexpected behavior. One solution is to use the `synchronized` keyword to protect a *critical section* in the code that uses the shared object. It is important to restrict the number of such critical sections. The more places shared data can get updated, the more likely:

- You will forget to protect one or more of those places—effectively breaking all code that modifies that shared data.
- There will be duplication of effort required to make sure everything is effectively guarded (violation of DRY[7]).
- It will be difficult to determine the source of failures, which are already hard enough to find.

Recommendation: *Take data encapsulation to heart; severely limit the access of any data that may be shared.*

Corollary: Use Copies of Data

A good way to avoid shared data is to avoid sharing the data in the first place. In some situations it is possible to copy objects and treat them as read-only. In other cases it might be possible to copy objects, collect results from multiple threads in these copies and then merge the results in a single thread.

If there is an easy way to avoid sharing objects, the resulting code will be far less likely to cause problems. You might be concerned about the cost of all the extra object creation. It is worth experimenting to find out if this is in fact a problem. However, if using copies of objects allows the code to avoid synchronizing, the savings in avoiding the intrinsic lock will

likely make up for the additional creation and garbage collection overhead.

Corollary: Threads Should Be as Independent as Possible

Consider writing your threaded code such that each thread exists in its own world, sharing no data with any other thread. Each thread processes one client request, with all of its required data coming from an unshared source and stored as local variables. This makes each of those threads behave as if it were the only thread in the world and there were no synchronization requirements.

For example, classes that subclass from `HttpServlet` receive all of their information as parameters passed in to the `doGet` and `doPost` methods. This makes each `Servlet` act as if it has its own machine. So long as the code in the `Servlet` uses only local variables, there is no chance that the `Servlet` will cause synchronization problems. Of course, most applications using `Servlets` eventually run into shared resources such as database connections.

Recommendation: *Attempt to partition data into independent subsets than can be operated on by independent threads, possibly in different processors.*

Know Your Library

Java 5 offers many improvements for concurrent development over previous versions. There are several things to consider when writing threaded code in Java 5:

- Use the provided thread-safe collections.
- Use the executor framework for executing unrelated tasks.
- Use nonblocking solutions when possible.
- Several library classes are not thread safe.

Thread-Safe Collections

When Java was young, Doug Lea wrote the seminal book[8] *Concurrent Programming in Java*. Along with the book he developed several thread-

safe collections, which later became part of the JDK in the `java.util.concurrent` package. The collections in that package are safe for multithreaded situations and they perform well. In fact, the `ConcurrentHashMap` implementation performs better than `HashMap` in nearly all situations. It also allows for simultaneous concurrent reads and writes, and it has methods supporting common composite operations that are otherwise not thread safe. If Java 5 is the deployment environment, start with `ConcurrentHashMap`.

There are several other kinds of classes added to support advanced concurrency design. Here are a few examples:

`ReentrantLock`	A lock that can be acquired in one method and released in another.
`Semaphore`	An implementation of the classic semaphore, a lock with a count.
`CountDownLatch`	A lock that waits for a number of events before releasing all threads waiting on it. This allows all threads to have a fair chance of starting at about the same time.

Recommendation: *Review the classes available to you. In the case of Java, become familiar with java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks.*

Know Your Execution Models

There are several different ways to partition behavior in a concurrent application. To discuss them we need to understand some basic definitions.

Bound Resources	Resources of a fixed size or number used in a concurrent environment. Examples include database connections and fixed-size read/write buffers.
Mutual Exclusion	Only one thread can access shared data or a shared resource at a time.
Starvation	One thread or a group of threads is prohibited from proceeding for an excessively long time or forever. For example, always letting fast-running threads through first could starve out longer running threads if there is no end to the fast-running threads.
Deadlock	Two or more threads waiting for each other to finish. Each thread has a resource that the other thread requires and neither can finish until it gets the other resource.
Livelock	Threads in lockstep, each trying to do work but finding another "in the way." Due to resonance, threads continue trying to make progress but are unable to for an excessively long time—or forever.

Given these definitions, we can now discuss the various execution models used in concurrent programming.

Producer-Consumer[9]

One or more producer threads create some work and place it in a buffer or queue. One or more consumer threads acquire that work from the queue and complete it. The queue between the producers and consumers is a *bound resource*. This means producers must wait for free space in the queue before writing and consumers must wait until there is something in the queue to consume. Coordination between the producers and consumers via the queue involves producers and consumers signaling each other. The producers write to the queue and signal that the queue is no longer empty. Consumers read from the queue and signal that the queue is no longer full. Both potentially wait to be notified when they can continue.

Readers-Writers[10]

When you have a shared resource that primarily serves as a source of information for readers, but which is occasionally updated by writers, throughput is an issue. Emphasizing throughput can cause starvation and the accumulation of stale information. Allowing updates can impact throughput. Coordinating readers so they do not read something a writer is

updating and vice versa is a tough balancing act. Writers tend to block many readers for a long period of time, thus causing throughput issues.

The challenge is to balance the needs of both readers and writers to satisfy correct operation, provide reasonable throughput and avoiding starvation. A simple strategy makes writers wait until there are no readers before allowing the writer to perform an update. If there are continuous readers, however, the writers will be starved. On the other hand, if there are frequent writers and they are given priority, throughput will suffer. Finding that balance and avoiding concurrent update issues is what the problem addresses.

Dining Philosophers[11]

Imagine a number of philosophers sitting around a circular table. A fork is placed to the left of each philosopher. There is a big bowl of spaghetti in the center of the table. The philosophers spend their time thinking unless they get hungry. Once hungry, they pick up the forks on either side of them and eat. A philosopher cannot eat unless he is holding two forks. If the philosopher to his right or left is already using one of the forks he needs, he must wait until that philosopher finishes eating and puts the forks back down. Once a philosopher eats, he puts both his forks back down on the table and waits until he is hungry again.

Replace philosophers with threads and forks with resources and this problem is similar to many enterprise applications in which processes compete for resources. Unless carefully designed, systems that compete in this way can experience deadlock, livelock, throughput, and efficiency degradation.

Most concurrent problems you will likely encounter will be some variation of these three problems. Study these algorithms and write solutions using them on your own so that when you come across concurrent problems, you'll be more prepared to solve the problem.

Recommendation: *Learn these basic algorithms and understand their solutions.*

Beware Dependencies Between Synchronized Methods

Dependencies between synchronized methods cause subtle bugs in concurrent code. The Java language has the notion of `synchronized`, which protects an individual method. However, if there is more than one synchronized method on the same shared class, then your system may be written incorrectly.[12]

Recommendation: *Avoid using more than one method on a shared object.*

There will be times when you must use more than one method on a shared object. When this is the case, there are three ways to make the code correct:

- **Client-Based Locking**—Have the client lock the server before calling the first method and make sure the lock's extent includes code calling the last method.
- **Server-Based Locking**—Within the server create a method that locks the server, calls all the methods, and then unlocks. Have the client call the new method.
- **Adapted Server**—create an intermediary that performs the locking. This is an example of server-based locking, where the original server cannot be changed.

Keep Synchronized Sections Small

The `synchronized` keyword introduces a lock. All sections of code guarded by the same lock are guaranteed to have only one thread executing through them at any given time. Locks are expensive because they create delays and add overhead. So we don't want to litter our code with `synchronized` statements. On the other hand, critical sections[13] must be guarded. So we want to design our code with as few critical sections as possible.

Some naive programmers try to achieve this by making their critical sections very large. However, extending synchronization beyond the

minimal critical section increases contention and degrades performance.[14]

Recommendation: *Keep your synchronized sections as small as possible.*

Writing Correct Shut-Down Code Is Hard

Writing a system that is meant to stay live and run forever is different from writing something that works for awhile and then shuts down gracefully.

Graceful shutdown can be hard to get correct. Common problems involve deadlock,[15] with threads waiting for a signal to continue that never comes.

For example, imagine a system with a parent thread that spawns several child threads and then waits for them all to finish before it releases its resources and shuts down. What if one of the spawned threads is deadlocked? The parent will wait forever, and the system will never shut down.

Or consider a similar system that has been *instructed* to shut down. The parent tells all the spawned children to abandon their tasks and finish. But what if two of the children were operating as a producer/consumer pair. Suppose the producer receives the signal from the parent and quickly shuts down. The consumer might have been expecting a message from the producer and be blocked in a state where it cannot receive the shutdown signal. It could get stuck waiting for the producer and never finish, preventing the parent from finishing as well.

Situations like this are not at all uncommon. So if you must write concurrent code that involves shutting down gracefully, expect to spend much of your time getting the shutdown to happen correctly.

Recommendation: *Think about shut-down early and get it working early. It's going to take longer than you expect. Review existing algorithms because this is probably harder than you think.*

Testing Threaded Code

Proving that code is correct is impractical. Testing does not guarantee correctness. However, good testing can minimize risk. This is all true in a single-threaded solution. As soon as there are two or more threads using the same code and working with shared data, things get substantially more complex.

Recommendation: *Write tests that have the potential to expose problems and then run them frequently, with different programatic configurations and system configurations and load. If tests ever fail, track down the failure. Don't ignore a failure just because the tests pass on a subsequent run.*

That is a whole lot to take into consideration. Here are a few more fine-grained recommendations:

- Treat spurious failures as candidate threading issues.
- Get your nonthreaded code working first.
- Make your threaded code pluggable.
- Make your threaded code tunable.
- Run with more threads than processors.
- Run on different platforms.
- Instrument your code to try and force failures.

Treat Spurious Failures as Candidate Threading Issues

Threaded code causes things to fail that "simply cannot fail." Most developers do not have an intuitive feel for how threading interacts with other code (authors included). Bugs in threaded code might exhibit their symptoms once in a thousand, or a million, executions. Attempts to repeat the systems can be frustratingly. This often leads developers to write off the failure as a cosmic ray, a hardware glitch, or some other kind of "one-off." It is best to assume that one-offs do not exist. The longer these "one-offs" are ignored, the more code is built on top of a potentially faulty approach.

Recommendation: *Do not ignore system failures as one-offs.*

Get Your Nonthreaded Code Working First

This may seem obvious, but it doesn't hurt to reinforce it. Make sure code works outside of its use in threads. Generally, this means creating POJOs that are called by your threads. The POJOs are not thread aware, and can therefore be tested outside of the threaded environment. The more of your system you can place in such POJOs, the better.

Recommendation: *Do not try to chase down nonthreading bugs and threading bugs at the same time. Make sure your code works outside of threads*.

Make Your Threaded Code Pluggable

Write the concurrency-supporting code such that it can be run in several configurations:

- One thread, several threads, varied as it executes
- Threaded code interacts with something that can be both real or a test double.
- Execute with test doubles that run quickly, slowly, variable.
- Configure tests so they can run for a number of iterations.

Recommendation: *Make your thread-based code especially pluggable so that you can run it in various configurations.*

Make Your Threaded Code Tunable

Getting the right balance of threads typically requires trial an error. Early on, find ways to time the performance of your system under different configurations. Allow the number of threads to be easily tuned. Consider allowing it to change while the system is running. Consider allowing self-tuning based on throughput and system utilization.

Run with More Threads Than Processors

Things happen when the system switches between tasks. To encourage task swapping, run with more threads than processors or cores. The more frequently your tasks swap, the more likely you'll encounter code that is missing a critical section or causes deadlock.

Run on Different Platforms

In the middle of 2007 we developed a course on concurrent programming. The course development ensued primarily under OS X. The class was presented using Windows XP running under a VM. Tests written to demonstrate failure conditions did not fail as frequently in an XP environment as they did running on OS X.

In all cases the code under test was known to be incorrect. This just reinforced the fact that different operating systems have different threading policies, each of which impacts the code's execution. Multithreaded code behaves differently in different environments.[16] You should run your tests in every potential deployment environment.

Recommendation: *Run your threaded code on all target platforms early and often.*

Instrument Your Code to Try and Force Failures

It is normal for flaws in concurrent code to hide. Simple tests often don't expose them. Indeed, they often hide during normal processing. They might show up once every few hours, or days, or weeks!

The reason that threading bugs can be infrequent, sporadic, and hard to repeat, is that only a very few pathways out of the many thousands of possible pathways through a vulnerable section actually fail. So the probability that a failing pathway is taken can be star-tlingly low. This makes detection and debugging very difficult.

How might you increase your chances of catching such rare occurrences? You can instrument your code and force it to run in different orderings by adding calls to methods like `Object.wait()`, `Object.sleep()`, `Object.yield()` and `Object.priority()`.

Each of these methods can affect the order of execution, thereby increasing the odds of detecting a flaw. It's better when broken code fails as early and as often as possible.

There are two options for code instrumentation:

- Hand-coded
- Automated

Hand-Coded

You can insert calls to `wait(), sleep(), yield()`, and `priority()` in your code by hand. It might be just the thing to do when you're testing a particularly thorny piece of code.

Here is an example of doing just that:

```
public synchronized String nextUrlOrNull() {
if(hasNext()) {
   String url = urlGenerator.next();
   Thread.yield(); // inserted for testing.
   updateHasNext();
   return url;
}
return null;
}
```

The inserted call to `yield()` will change the execution pathways taken by the code and possibly cause the code to fail where it did not fail before. If the code does break, it was not because you added a call to `yield()`.[17] Rather, your code was broken and this simply made the failure evident.

There are many problems with this approach:

- You have to manually find appropriate places to do this.
- How do you know where to put the call and what kind of call to use?
- Leaving such code in a production environment unnecessarily slows the code down.
- It's a shotgun approach. You may or may not find flaws. Indeed, the odds aren't with you.

What we need is a way to do this during testing but not in production. We also need to easily mix up configurations between different runs, which results in increased chances of finding errors in the aggregate.

Clearly, if we divide our system up into POJOs that know nothing of threading and classes that control the threading, it will be easier to find appropriate places to instrument the code. Moreover, we could create many different test jigs that invoke the POJOs under different regimes of calls to `sleep, yield`, and so on.

Automated

You could use tools like an Aspect-Oriented Framework, CGLIB, or ASM to programmatically instrument your code. For example, you could use a class with a single method:

```
public class ThreadJigglePoint {
public static void jiggle() {
}
}
```

You can add calls to this in various places within your code:

```
public synchronized String nextUrlOrNull() {
if(hasNext()) {
ThreadJiglePoint.jiggle();
String url = urlGenerator.next();
ThreadJiglePoint.jiggle();
updateHasNext();
ThreadJiglePoint.jiggle();
return url;
}
return null;
}
```

Now you use a simple aspect that randomly selects among doing nothing, sleeping, or yielding.

Or imagine that the `ThreadJigglePoint` class has two implementations. The first implements `jiggle` to do nothing and is used in production. The second generates a random number to choose between sleeping, yielding, or just falling through. If you run your tests a thousand times with random jiggling, you may root out some flaws. If the tests pass, at least you can say you've done due diligence. Though a bit simplistic, this could be a reasonable option in lieu of a more sophisticated tool.

There is a tool called ConTest,[18] developed by IBM that does something similar, but it does so with quite a bit more sophistication.

The point is to jiggle the code so that threads run in different orderings at different times. The combination of well-written tests and jiggling can dramatically increase the chance finding errors.

Recommendation: *Use jiggling strategies to ferret out errors.*

Conclusion

Concurrent code is difficult to get right. Code that is simple to follow can become nightmarish when multiple threads and shared data get into the mix. If you are faced with writing concurrent code, you need to write clean code with rigor or else face subtle and infrequent failures.

First and foremost, follow the Single Responsibility Principle. Break your system into POJOs that separate thread-aware code from thread-ignorant code. Make sure when you are testing your thread-aware code, you are only testing it and nothing else. This suggests that your thread-aware code should be small and focused.

Know the possible sources of concurrency issues: multiple threads operating on shared data, or using a common resource pool. Boundary cases, such as shutting down cleanly or finishing the iteration of a loop, can be especially thorny.

Learn your library and know the fundamental algorithms. Understand how some of the features offered by the library support solving problems similar to the fundamental algorithms.

Learn how to find regions of code that must be locked and lock them. Do not lock regions of code that do not need to be locked. Avoid calling one locked section from another. This requires a deep understanding of whether something is or is not shared. Keep the amount of shared objects and the scope of the sharing as narrow as possible. Change designs of the objects with shared data to accommodate clients rather than forcing clients to manage shared state.

Issues will crop up. The ones that do not crop up early are often written off as a onetime occurrence. These so-called one-offs typically only happen under load or at seemingly random times. Therefore, you need to be able to run your thread-related code in many configurations on many platforms repeatedly and continuously. Testability, which comes naturally from following the Three Laws of TDD, implies some level of plug-ability, which offers the support necessary to run code in a wider range of configurations.

You will greatly improve your chances of finding erroneous code if you take the time to instrument your code. You can either do so by hand or using some kind of automated technology. Invest in this early. You want to be running your thread-based code as long as possible before you put it into production.

If you take a clean approach, your chances of getting it right increase drastically.

Bibliography

[Lea99]: *Concurrent Programming in Java: Design Principles and Patterns*, 2d. ed., Doug Lea, Prentice Hall, 1999.

[PPP]: *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.

[PRAG]: *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.