Appendix A
Concurrency II

by Brett L. Schuchert

This appendix supports and amplifies the *Concurrency* chapter on page 177. It is written as a series of independent topics and you can generally read them in any order. There is some duplication between sections to allow for such reading.

Client/Server Example

Imagine a simple client/server application. A server sits and waits listening on a socket for a client to connect. A client connects and sends a request.

The Server

Here is a simplified version of a server application. Full source for this example is available starting on page 343, *Client/Server Nonthreaded*.

```
    ServerSocket serverSocket = new ServerSocket(8009);

while (keepProcessing) {
    try {
        Socket socket = serverSocket.accept();
        process(socket);
    } catch (Exception e) {
        handle(e);
    }
}
```

This simple application waits for a connection, processes an incoming message, and then again waits for the next client request to come in.

Here's client code that connects to this server:

```
    private void connectSendReceive(int i) {
    try {
       Socket socket = new Socket("localhost", PORT);
       MessageUtils.sendMessage(socket, Integer.toString(i));
       MessageUtils.getMessage(socket);
       socket.close();
    } catch (Exception e) {
       e.printStackTrace();
    }
  }
```

How well does this client/server pair perform? How can we formally describe that performance? Here's a test that asserts that the performance is "acceptable":

```
    @Test(timeout = 10000)
  public void shouldRunInUnder10Seconds() throws Exception {
     Thread[] threads = createThreads();
     startAllThreadsw(threads);
     waitForAllThreadsToFinish(threads);
  }
```

The setup is left out to keep the example simple (see "ClientTest.java" on page 344). This test asserts that it should complete within 10,000 milliseconds.

This is a classic example of validating the throughput of a system. This system should complete a series of client requests in ten seconds. So long as the server can process each individual client request in time, the test will pass.

What happens if the test fails? Short of developing some kind of event polling loop, there is not much to do within a single thread that will make this code any faster. Will using multiple threads solve the problem? It might, but we need to know where the time is being spent. There are two possibilities:

- I/O—using a socket, connecting to a database, waiting for virtual memory swapping, and so on.

- Processor—numerical calculations, regular expression processing, garbage collection, and so on.

Systems typically have some of each, but for a given operation one tends to dominate. If the code is processor bound, more processing hardware can improve throughput, making our test pass. But there are only so many CPU cycles available, so adding threads to a processor-bound problem will not make it go faster.

On the other hand, if the process is I/O bound, then concurrency can increase efficiency. When one part of the system is waiting for I/O, another part can use that wait time to process something else, making more effective use of the available CPU.

Adding Threading

Assume for the moment that the performance test fails. How can we improve the throughput so that the performance test passes? If the `process` method of the server is I/O bound, then here is one way to make the server use threads (just change the `processMessage`):

```
    void process(final Socket socket) {
    if (socket == null)
       return;

    Runnable clientHandler = new Runnable() {
       public void run() {
          try {
             String message = MessageUtils.getMessage(socket);
             MessageUtils.sendMessage(socket, "Processed: " + message);
             closeIgnoringException(socket);
          } catch (Exception e) {
             e.printStackTrace();
          }
       }
    };

    Thread clientConnection = new Thread(clientHandler);
    clientConnection.start();
  }
```

Assume that this change causes the test to pass;[1] the code is complete, correct?

Server Observations

The updated server completes the test successfully in just over one second. Unfortunately, this solution is a bit naive and introduces some new problems.

How many threads might our server create? The code sets no limit, so the we could feasibly hit the limit imposed by the Java Virtual Machine (JVM). For many simple systems this may suffice. But what if the system is meant to support many users on the public net? If too many users connect at the same time, the system might grind to a halt.

But set the behavioral problem aside for the moment. The solution shown has problems of cleanliness and structure. How many responsibilities does the server code have?

- Socket connection management
- Client processing
- Threading policy
- Server shutdown policy

Unfortunately, all these responsibilities live in the `process` function. In addition, the code crosses many different levels of abstraction. So, small as the process function is, it needs to be repartitioned.

The server has several reasons to change; therefore it violates the Single Responsibility Principle. To keep concurrent systems clean, thread management should be kept to a few, well-controlled places. What's more, any code that manages threads should do nothing other than thread management. Why? If for no other reason than that tracking down concurrency issues is hard enough without having to unwind other nonconcurrency issues at the same time.

If we create a separate class for each of the responsibilities listed above, including the thread management responsibility, then when we change the thread management strategy, the change will impact less overall code and will not pollute the other responsibilities. This also makes it much easier

to test all the other responsibilities without having to worry about threading. Here is an updated version that does just that:

```
    public void run() {
   while (keepProcessing) {
    try {
                                    ClientConnection clientConnection =
connectionManager.awaitClient();
     ClientRequestProcessor requestProcessor
       = new ClientRequestProcessor(clientConnection);
     clientScheduler.schedule(requestProcessor);
     } catch (Exception e) {
       e.printStackTrace();
     }
   }
   connectionManager.shutdown();
  }
```

This now focuses all things thread-related into one place, `clientScheduler`. If there are concurrency problems, there is just one place to look:

```
    public interface ClientScheduler {
    void schedule(ClientRequestProcessor requestProcessor);
  }
```

The current policy is easy to implement:

```
    public class ThreadPerRequestScheduler implements ClientScheduler
{
     public void schedule(final ClientRequestProcessor requestProcessor)
{
       Runnable runnable = new Runnable() {
          public void run() {
             requestProcessor.process();
          }
       };

       Thread thread = new Thread(runnable);
       thread.start();
```

```
    }
  }
```

Having isolated all the thread management into a single place, it is much easier to change the way we control threads. For example, moving to the Java 5 Executor framework involves writing a new class and plugging it in (Listing A-1).

Listing A-1 `ExecutorClientScheduler.java`

```
    import java.util.concurrent.Executor;
  import java.util.concurrent.Executors;

  public class ExecutorClientScheduler implements ClientScheduler {
    Executor executor;

    public ExecutorClientScheduler(int availableThreads) {
      executor = Executors.newFixedThreadPool(availableThreads);
    }

     public void schedule(final ClientRequestProcessor requestProcessor)
{
      Runnable runnable = new Runnable() {
        public void run() {
          requestProcessor.process();
        }
      };
      executor.execute(runnable);
    }
  }
```

Conclusion

Introducing concurrency in this particular example demonstrates a way to improve the throughput of a system and one way of validating that throughput through a testing framework. Focusing all concurrency code into a small number of classes is an example of applying the Single Responsibility Principle. In the case of concurrent programming, this becomes especially important because of its complexity.

Possible Paths of Execution

Review the method `incrementValue`, a one-line Java method with no looping or branching:

```
  public class IdGenerator {
 int lastIdUsed;

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

Ignore integer overflow and assume that only one thread has access to a single instance of `IdGenerator`. In this case there is a single path of execution and a single guaranteed result:

- The value returned is equal to the value of `lastIdUsed`, both of which are one greater than just before calling the method.

What happens if we use two threads and leave the method unchanged? What are the possible outcomes if each thread calls `incrementValue` once? How many possible paths of execution are there? First, the outcomes (assume `lastIdUsed` starts with a value of 93):

- Thread 1 gets the value of 94, thread 2 gets the value of 95, and `lastIdUsed` is now 95.
- Thread 1 gets the value of 95, thread 2 gets the value of 94, and `lastIdUsed` is now 95.
- Thread 1 gets the value of 94, thread 2 gets the value of 94, and `lastIdUsed` is now 94.

The final result, while surprising, is possible. To see how these different results are possible, we need to understand the number of possible paths of execution and how the Java Virtual Machine executes them.

Number of Paths

To calculate the number of possible execution paths, we'll start with the generated byte-code. The one line of java (`return ++lastIdUsed;`) becomes eight byte-code instructions. It is possible for the two threads to

interleave the execution of these eight instructions the way a card dealer interleaves cards as he shuffles a deck.[2] Even with only eight cards in each hand, there are a remarkable number of shuffled outcomes.

For this simple case of N instructions in a sequence, no looping or conditionals, and T threads, the total number of possible execution paths is equal to

$$\frac{(NT)!}{N!^T}$$

Calculating the Possible Orderings

This comes from an email from Uncle Bob to Brett:

With N steps and T threads there are $T^* N$ total steps. Prior to each step there is a context switch that chooses between the T threads. Each path can thus be represented as a string of digits denoting the context switches. Given steps A and B and threads 1 and 2, the six possible paths are 1122, 1212, 1221, 2112, 2121, and 2211. Or, in terms of steps it is A1B1A2B2, A1A2B1B2, A1A2B2B1, A2A1B1B2, A2A1B2B1, and A2B2A1B1. For three threads the sequence is 112233, 112323, 113223, 113232, 112233, 121233, 121323, 121332, 123132, 123123, ….

One characteristic of these strings is that there must always be N instances of each T. So the string 111111 is invalid because it has six instances of 1 and zero instances of 2 and 3.

So we want the permutations of N 1's, N 2's, … and N T's. This is really just the permutations of $N^* T$ things taken $N^* T$ at a time, which is $(N^* T)!$, but with all the duplicates removed. So the trick is to count the duplicates and subtract that from $(N^* T)!$.

Given two steps and two threads, how many duplicates are there? Each four-digit string has two 1s and two 2s. Each of those pairs could be swapped without changing the sense of the string. You could swap the 1s or the 2s both, or neither. So there are four isomorphs for each string, which means that there are three duplicates. So three out of four of the options are

duplicates; alternatively one of four of the permutations are NOT duplicates. 4! * .25 = 6. So this reasoning seems to work.

How many duplicates are there? In the case where $N = 2$ and $T = 2$, I could swap the 1s, the 2s, or both. In the case where $N = 2$ and $T = 3$, I could swap the 1s, the 2s, the 3s, 1s and 2s, 1s and 3s, or 2s and 3s. Swapping is just the permutations of N. Let's say there are P permutations of N. The number of different ways to arrange those permutations are $P^{**}T$.

So the number of possible isomorphs is $N!^{**}T$. And so the number of paths is $(T*N)!/(N!^{**}T)$. Again, in our $T = 2$, $N = 2$ case we get 6 (24/4).

For $N = 2$ and $T = 3$ we get 720/8 = 90.

For $N = 3$ and $T = 3$ we get 9!/6^3 = 1680.

For our simple case of one line of Java code, which equates to eight lines of byte-code and two threads, the total number of possible paths of execution is 12,870. If the type of `lastIdUsed` is a `long`, then every read/write becomes two operations instead of one, and the number of possible orderings becomes 2,704,156.

What happens if we make one change to this method?

```
  public synchronized void incrementValue() {
  ++lastIdUsed;
}
```

The number of possible execution pathways becomes two for two threads and N! in the general case.

Digging Deeper

What about the surprising result that two threads could both call the method once (before we added `synchronized`) and get the same numeric result? How is that possible? First things first.

What is an atomic operation? We can define an atomic operation as any operation that is uninterruptable. For example, in the following code, line 5, where 0 is assigned to `lastid`, is atomic because according to the Java Memory model, assignment to a 32-bit value is uninterruptable.

```
01: public class Example {
02:    int lastId;
03:
04:    public void resetId() {
05:       value = 0;
06:    }
07:
08:    public int getNextId() {
09:       ++value;
10:    }
11:}
```

What happens if we change type of `lastId` from `int` to `long`? Is line 5 still atomic? Not according to the JVM specification. It could be atomic on a particular processor, but according to the JVM specification, assignment to any 64-bit value requires two 32-bit assignments. This means that between the first 32-bit assignment and the second 32-bit assignment, some other thread could sneak in and change one of the values.

What about the pre-increment operator, ++, on line 9? The pre-increment operator can be interrupted, so it is not atomic. To understand, let's review the byte-code of both of these methods in detail.

Before we go any further, here are three definitions that will be important:

- Frame—Every method invocation requires a frame. The frame includes the return address, any parameters passed into the method and the local variables defined in the method. This is a standard technique used to define a call stack, which is used by modern languages to allow for basic function/method invocation and to allow for recursive invocation.
- Local variable—Any variables defined in the scope of the method. All nonstatic methods have at least one variable, **`this`**, which represents the current object, the object that received the most recent message (in the current thread), which caused the method invocation.
- Operand stack—Many of the instructions in the Java Virtual Machine take parameters. The operand stack is where those

parameters are put. The stack is a standard last-in, first-out (LIFO) data structure.

Here is the byte-code generated for `resetId()`:

Mnemonic	Description	Operand Stack After
`ALOAD 0`	Load the 0th variable onto the operand stack. What is the 0th variable? It is **`this`**., the current object. When the method was called, the receiver of the message, an instance of `Example`, was pushed into the local variable array of the frame created for method invocation. This is always the first variable put in every instance method.	`this`

Mnemonic	Description	Operand Stack After
`ICONST_0`	Put the constant value 0 onto the operand stack.	`this, 0`
`PUTFIELD lastId`	Store the top value on the stack (which is 0) into the field value of the object referred to by the object reference one away from the top of the stack, **`this`**.	<empty>

These three instructions are guaranteed to be atomic because, although the thread executing them could be interrupted after any one of them, the information for the PUTFIELD instruction (the constant value 0 on the top of the stack and the reference to `this` one below the top, along with the field value) cannot be touched by another thread. So when the assignment occurs, we are guaranteed that the value 0 will be stored in the field value. The operation is atomic. The operands all deal with information local to the method, so there is no interference between multiple threads.

So if these three instructions are executed by ten threads, there are 4.38679733629e+24 possible orderings. However, there is only one possible outcome, so the different orderings are irrelevant. It just so happens that the same outcome is guaranteed for longs in this case as well. Why? All ten threads are assigning a constant value. Even if they interleave with each other, the end result is the same.

With the `++` operation in the `getNextId` method, there are going to be problems. Assume that `lastId` holds 42 at the beginning of this method. Here is the byte-code for this new method:

Mnemonic	Description	Operand Stack After
`ALOAD 0`	Load `this` onto the operand stack	`this`
`DUP`	Copy the top of the stack. We now have two copies of `this` on the operand stack.	`this, this`
`GETFIELD lastId`	Retrieve the value of the field `lastId` from the object pointed to on the top of the stack (`this`) and store that value back on to the stack.	`this`, 42
`ICONST_1`	Push the integer constant 1 on the stack.	`this`, 42, 1
`IADD`	Integer add the top two values on the operand stack and store the result back on to the operand stack.	`this`, 43
`DUP_X1`	Duplicate the value 43 and put it before `this`.	43, `this`, 43
`PUTFIELD value`	Store the top value on the operand stack, 43, into the field value of the current object, represented by the next-to-top value on the operand stack, `this`.	43
`IRETURN`	return the top (and only) value on the stack.	<empty>

Imagine the case where the first thread completes the first three instructions, up to and including GETFIELD, and then it is interrupted. A second thread takes over and performs the entire method, incrementing `lastId` by one; it gets 43 back. Then the first thread picks up where it left off; 42 is still on the operand stack because that was the value of `lastId` when it executed GETFIELD. It adds one to get 43 again and stores the result. The value 43 is returned to the first thread as well. The result is that one of the increments is lost because the first thread stepped on the second thread after the second thread interrupted the first thread.

Making the `getNexId()` method synchronized fixes this problem.

Conclusion

An intimate understanding of byte-code is not necessary to understand how threads can step on each other. If you can understand this one example, it should demonstrate the possibility of multiple threads stepping on each other, which is enough knowledge.

That being said, what this trivial example demonstrates is a need to understand the memory model enough to know what is and is not safe. It is a common misconception that the ++ (pre- or post-increment) operator is atomic, and it clearly is not. This means you need to know:

- Where there are shared objects/values
- The code that can cause concurrent read/update issues
- How to guard such concurrent issues from happening

Knowing Your Library

Executor Framework

As demonstrated in the `ExecutorClientScheduler.java` on page 321, the `Executor` framework introduced in Java 5 allows for sophisticated execution using thread pools. This is a class in the `java.util.concurrent` package.

If you are creating threads and are not using a thread pool or *are* using a hand-written one, you should consider using the `Executor`. It will make your code cleaner, easier to follow, and smaller.

The `Executor` framework will pool threads, resize automatically, and recreate threads if necessary. It also supports *futures,* a common concurrent programming construct. The `Executor` framework works with classes that implement `Runnable` and also works with classes that implement the `Callable` interface. A `Callable` looks like a `Runnable`, but it can return a result, which is a common need in multithreaded solutions.

A *future* is handy when code needs to execute multiple, independent operations and wait for both to finish:

```
public String processRequest(String message) throws Exception {
Callable<String> makeExternalCall = new Callable<String>() {

  public String call() throws Exception {
    String result = “”;
    // make external request
    return result;
  }
```

```
    };

    Future<String> result = executorService.submit(makeExternalCall);
    String partialResult = doSomeLocalProcessing();
    return result.get() + partialResult;
  }
```

In this example, the method starts executing the `makeExternalCall` object. The method continues other processing. The final line calls `result.get()`, which blocks until the future completes.

Nonblocking Solutions

The Java 5 VM takes advantage of modern processor design, which supports reliable, nonblocking updates. Consider, for example, a class that uses synchronization (and therefore blocking) to provide a thread-safe update of a value:

```
    public class ObjectWithValue {
    private int value;
    public void synchronized incrementValue() { ++value; }
    public int getValue() { return value; }
  }
```

Java 5 has a series of new classes for situations like this: `AtomicBoolean`, `AtomicInteger`, and `AtomicReference` are three examples; there are several more. We can rewrite the above code to use a nonblocking approach as follows:

```
    public class ObjectWithValue {
    private AtomicInteger value = new AtomicInteger(0);

    public void incrementValue() {
    value.incrementAndGet();
    }
    public int getValue() {
       return value.get();
    }
  }
```

Even though this uses an object instead of a primitive and sends messages like `incrementAndGet()` instead of ++, the performance of this class will nearly always beat the previous version. In some cases it will only be slightly faster, but the cases where it will be slower are virtually nonexistent.

How is this possible? Modern processors have an operation typically called *Compare and Swap (CAS)*. This operation is analogous to optimistic locking in databases, whereas the synchronized version is analogous to pessimistic locking.

The `synchronized` keyword always acquires a lock, even when a second thread is not trying to update the same value. Even though the performance of intrinsic locks has improved from version to version, they are still costly.

The nonblocking version starts with the assumption that multiple threads generally do not modify the same value often enough that a problem will arise. Instead, it efficiently detects whether such a situation has occurred and retries until the update happens successfully. This detection is almost always less costly than acquiring a lock, even in moderate to high contention situations.

How does the Virtual Machine accomplish this? The CAS operation is atomic. Logically, the CAS operation looks something like the following:

```
    int variableBeingSet;

  void simulateNonBlockingSet(int newValue) {
    int currentValue;
    do {
      currentValue = variableBeingSet
    } while(currentValue != compareAndSwap(currentValue, newValue));
  }

  int synchronized compareAndSwap(int currentValue, int newValue) {
    if(variableBeingSet == currentValue) {
      variableBeingSet = newValue;
      return currentValue;
    }
```

```
    return variableBeingSet;
}
```

When a method attempts to update a shared variable, the CAS operation verifies that the variable getting set still has the last known value. If so, then the variable is changed. If not, then the variable is not set because another thread managed to get in the way. The method making the attempt (using the CAS operation) sees that the change was not made and retries.

Nonthread-Safe Classes

There are some classes that are inherently not thread safe. Here are a few examples:

- `SimpleDateFormat`
- Database Connections
- Containers in `java.util`
- Servlets

Note that some collection classes have individual methods that are thread-safe. However, any operation that involves calling more than one method is not. For example, if you do not want to replace something in a `HashTable` because it is already there, you might write the following code:

```
    if(!hashTable.containsKey(someKey)) {
    hashTable.put(someKey, new SomeValue());
}
```

Each individual method is thread-safe. However, another thread might add a value in between the `containsKey` and `put` calls. There are several options to fix this problem.

- Lock the `HashTable` first, and make sure all other users of the `HashTable` do the same—client-based locking:

```
    synchronized(map) {
  if(!map.conainsKey(key))
    map.put(key, value);
}
```

- Wrap the `HashTable` in its own object and use a different API—server-based locking using an ADAPTER:

```
public class WrappedHashtable<K, V> {
private Map<K, V> map = new Hashtable<K, V>();

public synchronized void putIfAbsent(K key, V value) {
   if (map.containsKey(key))
      map.put(key, value);
}
}
```

• Use the thread-safe collections:

```
ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<Integer,
String>();
map.putIfAbsent(key, value);
```

The collections in `java.util.concurrent` have operations like `putIfAbsent()` to accommodate such operations.

Dependencies Between Methods Can Break Concurrent Code

Here is a trivial example of a way to introduce dependencies between methods:

```
public class IntegerIterator implements Iterator<Integer>
private Integer nextValue = 0;

public synchronized boolean hasNext() {
   return nextValue < 100000;
}
public synchronized Integer next() {
   if (nextValue == 100000)
      throw new IteratorPastEndException();
   return nextValue++;
}
public synchronized Integer getNextValue() {
   return nextValue;
```

```
    }
}
```

Here is some code to use this `IntegerIterator`:

```
   IntegerIterator iterator = new IntegerIterator();
while(iterator.hasNext()) {
  int nextValue = iterator.next();
  // do something with nextValue
}
```

If one thread executes this code, there will be no problem. But what happens if two threads attempt to share a single instance of `IngeterIterator` with the intent that each thread will process the values it gets, but that each element of the list is processed only once? Most of the time, nothing bad happens; the threads happily share the list, processing the elements they are given by the iterator and stopping when the iterator is complete. However, there is a small chance that, at the end of the iteration, the two threads will interfere with each other and cause one thread to go beyond the end of the iterator and throw an exception.

Here's the problem: Thread 1 asks the question `hasNext()`, which returns `true`. Thread 1 gets preempted and then Thread 2 asks the same question, which is still `true`. Thread 2 then calls `next()`, which returns a value as expected but has a side effect of making `hasNext()` return `false`. Thread 1 starts up again, thinking `hasNext()` is still `true`, and then calls `next()`. Even though the individual methods are synchronized, the client uses ***two*** methods.

This is a real problem and an example of the kinds of problems that crop up in concurrent code. In this particular situation this problem is especially subtle because the only time where this causes a fault is when it happens during the final iteration of the iterator. If the threads happen to break just right, then one of the threads could go beyond the end of the iterator. This is the kind of bug that happens long after a system has been in production, and it is hard to track down.

You have three options:

- Tolerate the failure.
- Solve the problem by changing the client: client-based locking

• Solve the problem by changing the server, which additionally changes the client: server-based locking

Tolerate the Failure

Sometimes you can set things up such that the failure causes no harm. For example, the above client could catch the exception and clean up. Frankly, this is a bit sloppy. It's rather like cleaning up memory leaks by rebooting at midnight.

Client-Based Locking

To make `IntegerIterator` work correctly with multiple threads, change this client (and every other client) as follows:

```
IntegerIterator iterator = new IntegerIterator();

while (true) {
  int nextValue;
 synchronized (iterator) {
  if (!iterator.hasNext())
    break;
  nextValue = iterator.next();
 }
 doSometingWith(nextValue);
}
```

Each client introduces a lock via the `synchronized` keyword. This duplication violates the DRY principle, but it might be necessary if the code uses non-thread-safe third-party tools.

This strategy is risky because all programmers who use the server must remember to lock it before using it and unlock it when done. Many (many!) years ago I worked on a system that employed client-based locking on a shared resource. The resource was used in hundreds of different places throughout the code. One poor programmer forgot to lock the resource in one of those places.

The system was a multi-terminal time-sharing system running accounting software for Local 705 of the trucker's union. The computer was in a raised-floor, environment-controlled room 50 miles north of the

Local 705 headquarters. At the headquarters they had dozens of data entry clerks typing union dues postings into the terminals. The terminals were connected to the computer using dedicated phone lines and 600bps half-duplex modems. (This was a very, *very* long time ago.)

About once per day, one of the terminals would "lock up." There was no rhyme or reason to it. The lock up showed no preference for particular terminals or particular times. It was as though there were someone rolling dice choosing the time and terminal to lock up. Sometimes more than one terminal would lock up. Sometimes days would go by without any lock-ups.

At first the only solution was a reboot. But reboots were tough to coordinate. We had to call the headquarters and get everyone to finish what they were doing on all the terminals. Then we could shut down and restart. If someone was doing something important that took an hour or two, the locked up terminal simply had to stay locked up.

After a few weeks of debugging we found that the cause was a ring-buffer counter that had gotten out of sync with its pointer. This buffer controlled output to the terminal. The pointer value indicated that the buffer was empty, but the counter said it was full. Because it was empty, there was nothing to display; but because it was also full, nothing could be added to the buffer to be displayed on the screen.

So we knew why the terminals were locking, but we didn't know why the ring buffer was getting out of sync. So we added a hack to work around the problem. It was possible to read the front panel switches on the computer. (This was a very, very, *very* long time ago.) We wrote a little trap function that detected when one of these switches was thrown and then looked for a ring buffer that was both empty and full. If one was found, it reset that buffer to empty. *Voila!* The locked-up terminal(s) started displaying again.

So now we didn't have to reboot the system when a terminal locked up. The Local would simply call us and tell us we had a lock-up, and then we just walked into the computer room and flicked a switch.

Of course sometimes they worked on the weekends, and we didn't. So we added a function to the scheduler that checked all the ring buffers once

per minute and reset any that were both empty and full. This caused the displays to unclog before the Local could even get on the phone.

It was several more weeks of poring over page after page of monolithic assembly language code before we found the culprit. We had done the math and calculated that the frequency of the lock-ups was consistent with a single unprotected use of the ring buffer. So all we had to do was find that one faulty usage. Unfortunately, this was so very long ago that we didn't have search tools or cross references or any other kind of automated help. We simply had to pore over listings.

I learned an important lesson that cold Chicago winter of 1971. Client-based locking really blows.

Server-Based Locking

The duplication can be removed by making the following changes to `IntegerIterator`:

```
public class IntegerIteratorServerLocked {
private Integer nextValue = 0;
public synchronized Integer getNextOrNull() {
   if (nextValue < 100000)
      return nextValue++;
   else
     return null;
}
}
```

And the client code changes as well:

```
while (true) {
Integer nextValue = iterator.getNextOrNull();
if (next == null)
   break;
// do something with nextValue
}
```

In this case we actually change the API of our class to be multithread aware.[3] The client needs to perform a `null` check instead of checking `hasNext()`.

In general you should prefer server-based locking for these reasons:

• It reduces repeated code—Client-based locking forces each client to lock the server properly. By putting the locking code into the server, clients are free to use the object and not worry about writing additional locking code.

• It allows for better performance—You can swap out a thread-safe server for a non-thread safe one in the case of single-threaded deployment, thereby avoiding all overhead.

• It reduces the possibility of error—All it takes is for one programmer to forget to lock properly.

• It enforces a single policy—The policy is in one place, the server, rather than many places, each client.

• It reduces the scope of the shared variables—The client is not aware of them or how they are locked. All of that is hidden in the server. When things break, the number of places to look is smaller.

What if you do not own the server code?

• Use an ADAPTER to change the API and add locking

```
public class ThreadSafeIntegerIterator {
private IntegerIterator iterator = new IntegerIterator();

public synchronized Integer getNextOrNull() {
   if(iterator.hasNext())
      return iterator.next();
   return null;
}
}
```

• OR better yet, use the thread-safe collections with extended interfaces

Increasing Throughput

Let's assume that we want to go out on the net and read the contents of a set of pages from a list of URLs. As each page is read, we will parse it to

accumulate some statistics. Once all the pages are read, we will print a summary report.

The following class returns the contents of one page, given a URL.

```
  public class PageReader {
 //…
 public String getPageFor(String url) {
  HttpMethod method = new GetMethod(url);

  try {
   httpClient.executeMethod(method);
   String response = method.getResponseBodyAsString();
   return response;
  } catch (Exception e) {
   handle(e);
  } finally {
   method.releaseConnection();
  }
 }
}
```

The next class is the iterator that provides the contents of the pages based on an iterator of URLs:

```
  public class PageIterator {
 private PageReader reader;
 private URLIterator urls;

 public PageIterator(PageReader reader, URLIterator urls) {
  this.urls = urls;
  this.reader = reader;
 }

 public synchronized String getNextPageOrNull() {
  if (urls.hasNext())
   getPageFor(urls.next());
  else
   return null;
 }
```

```
    public String getPageFor(String url) {
      return reader.getPageFor(url);
    }
  }
```

An instance of the `PageIterator` can be shared between many different threads, each one using it's own instance of the `PageReader` to read and parse the pages it gets from the iterator.

Notice that we've kept the `synchronized` block very small. It contains just the critical section deep inside the `PageIterator`. It is always better to synchronize as little as possible as opposed to synchronizing as much as possible.

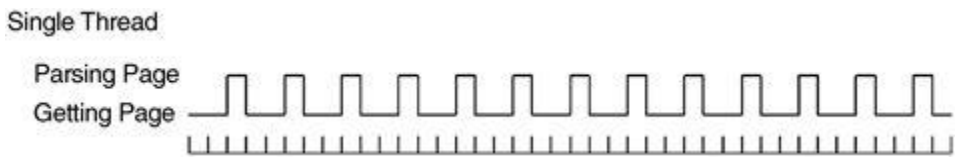
Single-Thread Calculation of Throughput

Now lets do some simple calculations. For the purpose of argument, assume the following:

- I/O time to retrieve a page (average): 1 second
- Processing time to parse page (average): .5 seconds
- I/O requires 0 percent of the CPU while processing requires 100 percent.

For *N* pages being processed by a single thread, the total execution time is 1.5 seconds * *N*. Figure A-1 shows a snapshot of 13 pages or about 19.5 seconds.

Figure A-1 Single thread



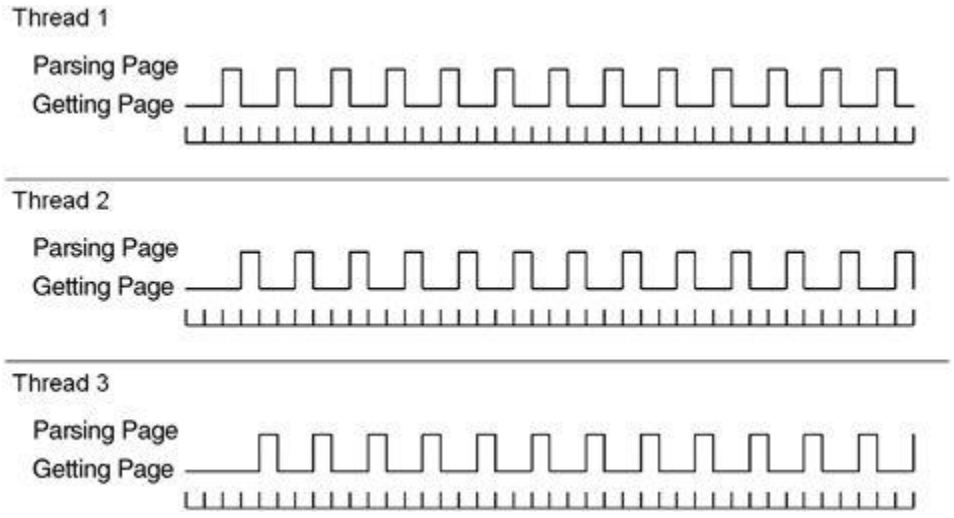
Multithread Calculation of Throughput

If it is possible to retrieve pages in any order and process the pages independently, then it is possible to use multiple threads to increase

throughput. What happens if we use three threads? How many pages can we acquire in the same time?

As you can see in Figure A-2, the multithreaded solution allows the process-bound parsing of the pages to overlap with the I/O-bound reading of the pages. In an idealized world this means that the processor is fully utilized. Each one-second page read is overlapped with two parses. Thus, we can process two pages per second, which is three times the throughput of the single-threaded solution.

Figure A-2 Three concurrent threads



Deadlock

Imagine a Web application with two shared resource pools of some finite size:

- A pool of database connections for local work in process storage
- A pool of MQ connections to a master repository

Assume there are two operations in this application, create and update:

- Create—Acquire connection to master repository and database. Talk to service master repository and then store work in local work in process database.

• Update—Acquire connection to database and then master repository. Read from work in process database and then send to the master repository

What happens when there are more users than the pool sizes? Consider each pool has a size of ten.

• Ten users attempt to use create, so all ten database connections are acquired, and each thread is interrupted after acquiring a database connection but before acquiring a connection to the master repository.

• Ten users attempt to use update, so all ten master repository connections are acquired, and each thread is interrupted after acquiring the master repository but before acquiring a database connection.

• Now the ten "create" threads must wait to acquire a master repository connection, but the ten "update" threads must wait to acquire a database connection.

• Deadlock. The system never recovers.

This might sound like an unlikely situation, but who wants a system that freezes solid every other week? Who wants to debug a system with symptoms that are so difficult to reproduce? This is the kind of problem that happens in the field, then takes weeks to solve.

A typical "solution" is to introduce debugging statements to find out what is happening. Of course, the debug statements change the code enough so that the deadlock happens in a different situation and takes months to again occur.[4]

To really solve the problem of deadlock, we need to understand what causes it. There are four conditions required for deadlock to occur:

• Mutual exclusion

• Lock & wait

• No preemption

• Circular wait

Mutual Exclusion

Mutual exclusion occurs when multiple threads need to use the same resources and those resources

- Cannot be used by multiple threads at the same time.
- Are limited in number.

A common example of such a resource is a database connection, a file open for write, a record lock, or a semaphore.

Lock & Wait

Once a thread acquires a resource, it will not release the resource until it has acquired all of the other resources it requires and has completed its work.

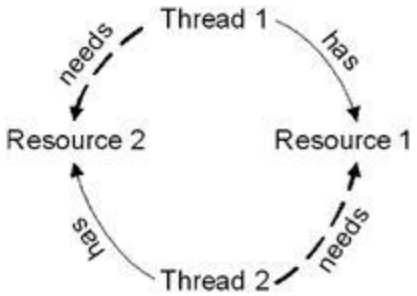
No Preemption

One thread cannot take resources away from another thread. Once a thread holds a resource, the only way for another thread to get it is for the holding thread to release it.

Circular Wait

This is also referred to as the deadly embrace. Imagine two threads, T1 and T2, and two resources, R1 and R2. T1 has R1, T2 has R2. T1 also requires R2, and T2 also requires R1. This gives something like Figure A-3:

Figure A-3



All four of these conditions must hold for deadlock to be possible. Break any one of these conditions and deadlock is not possible.

Breaking Mutual Exclusion

One strategy for avoiding deadlock is to sidestep the mutual exclusion condition. You might be able to do this by

- Using resources that allow simultaneous use, for example, `AtomicInteger`.
- Increasing the number of resources such that it equals or exceeds the number of competing threads.
- Checking that all your resources are free before seizing any.

Unfortunately, most resources are limited in number and don't allow simultaneous use. And it's not uncommon for the identity of the second resource to be predicated on the results of operating on the first. But don't be discouraged; there are three conditions left.

Breaking Lock & Wait

You can also eliminate deadlock if you refuse to wait. Check each resource before you seize it, and release all resources and start over if you run into one that's busy.

This approach introduces several potential problems:

- Starvation—One thread keeps being unable to acquire the resources it needs (maybe it has a unique combination of resources that seldom all become available).
- Livelock—Several threads might get into lockstep and all acquire one resource and then release one resource, over and over again. This is especially likely with simplistic CPU scheduling algorithms (think embedded devices or simplistic hand-written thread balancing algorithms).

Both of these can cause poor throughput. The first results in low CPU utilization, whereas the second results in high and useless CPU utilization.

As inefficient as this strategy sounds, it's better than nothing. It has the benefit that it can almost always be implemented if all else fails.

Breaking Preemption

Another strategy for avoiding deadlock is to allow threads to take resources away from other threads. This is usually done through a simple request mechanism. When a thread discovers that a resource is busy, it asks the owner to release it. If the owner is also waiting for some other resource, it releases them all and starts over.

This is similar to the previous approach but has the benefit that a thread is allowed to wait for a resource. This decreases the number of startovers. Be warned, however, that managing all those requests can be tricky.

Breaking Circular Wait

This is the most common approach to preventing deadlock. For most systems it requires no more than a simple convention agreed to by all parties.

In the example above with Thread 1 wanting both Resource 1 and Resource 2 and Thread 2 wanting both Resource 2 and then Resource 1, simply forcing both Thread 1 and Thread 2 to allocate resources in the same order makes circular wait impossible.

More generally, if all threads can agree on a global ordering of resources and if they all allocate resources in that order, then deadlock is impossible. Like all the other strategies, this can cause problems:

- The order of acquisition might not correspond to the order of use; thus a resource acquired at the start might not be used until the end. This can cause resources to be locked longer than strictly necessary.
- Sometimes you cannot impose an order on the acquisition of resources. If the ID of the second resource comes from an operation performed on the first, then ordering is not feasible.

So there are many ways to avoid deadlock. Some lead to starvation, whereas others make heavy use of the CPU and reduce responsiveness. TANSTAAFL![5]

Isolating the thread-related part of your solution to allow for tuning and experimentation is a powerful way to gain the insights needed to determine the best strategies.

Testing Multithreaded Code

How can we write a test to demonstrate the following code is broken?

```
01: public class ClassWithThreadingProblem {
02:    int nextId;
03:
04:    public int takeNextId() {
05:       return nextId++;
06:    }
07:}
```

Here's a description of a test that will prove the code is broken:

- Remember the current value of `nextId`.
- Create two threads, both of which call `takeNextId()` once.
- Verify that `nextId` is two more than what we started with.
- Run this until we demonstrate that `nextId` was only incremented by one instead of two.

Listing A-2 shows such a test:

Listing A-2 `ClassWithThreadingProblemTest.java`

```
01: package example;
02:
03: import static org.junit.Assert.fail;
04:
05: import org.junit.Test;
06:
07: public class ClassWithThreadingProblemTest {
08:    @Test
09:    public void twoThreadsShouldFailEventually() throws Exception
{
10:       final ClassWithThreadingProblem classWithThreadingProblem
           = new ClassWithThreadingProblem();
11:
12:       Runnable runnable = new Runnable() {
13:          public void run() {
14:             classWithThreadingProblem.takeNextId();
15:          }
```

```
16:         };
17:
18:         for (int i = 0; i < 50000; ++i) {
19:             int startingId = classWithThreadingProblem.lastId;
20:             int expectedResult = 2 + startingId;
21:
22:             Thread t1 = new Thread(runnable);
23:             Thread t2 = new Thread(runnable);
24:             t1.start();
25:             t2.start();
26:             t1.join();
27:             t2.join();
28:
29:             int endingId = classWithThreadingProblem.lastId;
30:
31:             if (endingId != expectedResult)
32:                 return;
33:         }
34:
35:         fail("Should have exposed a threading issue but it did not.");
36:     }
37: }
```

Line	Description
10	Create a single instance of `ClassWithThreadingProblem`. Note, we must use the final keyword because we use it below in an anonymous inner class.
12-16	Create an anonymous inner class that uses the single instance of `ClassWithThreadingProblem`.
18	Run this code "enough" times to demonstrate that the code failed, but not so much that the test "takes too long." This is a balancing act; we don't want to wait too long to demonstrate failure. Picking this number is hard—although later we'll see that we can greatly reduce this number.
19	Remember the starting value. This test is trying to prove that the code in `ClassWithThreadingProblem` is broken. If this test passes, it proved that the code was broken. If this test fails, the test was unable to prove that the code is broken.
20	We expect the final value to be two more than the current value.
22-23	Create two threads, both of which use the object we created in lines 12–16. This gives us the potential of two threads trying to use our single instance of `ClassWithThreadingProblem` and interfering with each other.

Line	Description
24-25	Make our two threads eligible to run.
26-27	Wait for both threads to finish before we check the results.
29	Record the actual final value.
31-32	Did our `endingId` differ from what we expected? If so, return end the test—we've proven that the code is broken. If not, try again.
35	If we got to here, our test was unable to prove the production code was broken in a "reasonable" amount of time; our code has failed. Either the code is not broken or we didn't run enough iterations to get the failure condition to occur.

This test certainly sets up the conditions for a concurrent update problem. However, the problem occurs so infrequently that the vast majority of times this test won't detect it.

Indeed, to truly detect the problem we need to set the number of iterations to over one million. Even then, in ten executions with a loop count of 1,000,000, the problem occurred only once. That means we probably ought to set the iteration count to well over one hundred million to get reliable failures. How long are we prepared to wait?

Even if we tuned the test to get reliable failures on one machine, we'll probably have to retune the test with different values to demonstrate the

failure on another machine, operating system, or version of the JVM.

And this is a *simple* problem. If we cannot demonstrate broken code easily with this problem, how will we ever detect truly complex problems?

So what approaches can we take to demonstrate this simple failure? And, more importantly, how can we write tests that will demonstrate failures in more complex code? How will we be able to discover if our code has failures when we do not know where to look?

Here are a few ideas:

- **Monte Carlo Testing.** Make tests flexible, so they can be tuned. Then run the test over and over—say on a test server—randomly changing the tuning values. If the tests ever fail, the code is broken. Make sure to start writing those tests early so a continuous integration server starts running them soon. By the way, make sure you carefully log the conditions under which the test failed.
- Run the test on every one of the target deployment platforms. Repeatedly. Continuously. The longer the tests run without failure, the more likely that
  - The production code is correct or
  - The tests aren't adequate to expose problems.
- Run the tests on a machine with varying loads. If you can simulate loads close to a production environment, do so.

Yet, even if you do all of these things, you still don't stand a very good chance of finding threading problems with your code. The most insidious problems are the ones that have such a small cross section that they only occur once in a billion opportunities. Such problems are the terror of complex systems.

Tool Support for Testing Thread-Based Code

IBM has created a tool called ConTest.[6] It instruments classes to make it more likely that non-thread-safe code fails.

We do not have any direct relationship with IBM or the team that developed ConTest. A colleague of ours pointed us to it. We noticed vast

improvement in our ability to find threading issues after a few minutes of using it.

Here's an outline of how to use ConTest:

- Write tests and production code, making sure there are tests specifically designed to simulate multiple users under varying loads, as mentioned above.
- Instrument test and production code with ConTest.
- Run the tests.

When we instrumented code with ConTest, our success rate went from roughly one failure in ten million iterations to roughly one failure in *thirty* iterations. Here are the loop values for several runs of the test after instrumentation: 13, 23, 0, 54, 16, 14, 6, 69, 107, 49, 2. So clearly the instrumented classes failed much earlier and with much greater reliability.

Conclusion

This chapter has been a very brief sojourn through the large and treacherous territory of concurrent programming. We barely scratched the surface. Our emphasis here was on disciplines to help keep concurrent code clean, but there is much more you should learn if you are going to be writing concurrent systems. We recommend you start with Doug Lea's wonderful book *Concurrent Programming in Java: Design Principles and Patterns*.[7]

In this chapter we talked about concurrent update, and the disciplines of clean synchronization and locking that can prevent it. We talked about how threads can enhance the throughput of an I/O-bound system and showed the clean techniques for achieving such improvements. We talked about deadlock and the disciplines for preventing it in a clean way. Finally, we talked about strategies for exposing concurrent problems by instrumenting your code.

Tutorial: Full Code Examples

Client/Server Nonthreaded

Listing A-3 `Server.java`

```
package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
   ServerSocket serverSocket;
   volatile boolean keepProcessing = true;

    public Server(int port, int millisecondsTimeout) throws IOException
{
      serverSocket = new ServerSocket(port);
      serverSocket.setSoTimeout(millisecondsTimeout);
   }

   public void run() {
      System.out.printf("Server Starting\n");

      while (keepProcessing) {
         try {
            System.out.printf("accepting client\n");
            Socket socket = serverSocket.accept();
            System.out.printf("got client\n");
            process(socket);
         } catch (Exception e) {
            handle(e);
         }
      }
   }

   private void handle(Exception e) {
      if (!(e instanceof SocketException)) {
```

```
            e.printStackTrace();
        }
    }

    public void stopProcessing() {
        keepProcessing = false;
        closeIgnoringException(serverSocket);
    }
    void process(Socket socket) {
        if (socket == null)
            return;

        try {
            System.out.printf("Server: getting message\n");
            String message = MessageUtils.getMessage(socket);
            System.out.printf("Server: got message: %s\n", message);
            Thread.sleep(1000);
            System.out.printf("Server: sending reply: %s\n", message);
            MessageUtils.sendMessage(socket, "Processed: " + message);
            System.out.printf("Server: sent\n");
            closeIgnoringException(socket);
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

    private void closeIgnoringException(Socket socket) {
        if (socket != null)
            try {
                socket.close();
            } catch (IOException ignore) {
            }
    }

    private void closeIgnoringException(ServerSocket serverSocket) {
        if (serverSocket != null)
```

```
            try {
               serverSocket.close();
            } catch (IOException ignore) {
            }
      }
   }
```

Listing A-4 `ClientTest.java`

```
   package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
   ServerSocket serverSocket;
   volatile boolean keepProcessing = true;
    public Server(int port, int millisecondsTimeout) throws IOException
{
      serverSocket = new ServerSocket(port);
      serverSocket.setSoTimeout(millisecondsTimeout);
   }

   public void run() {
      System.out.printf("Server Starting\n");

      while (keepProcessing) {
         try {
            System.out.printf("accepting client\n");
            Socket socket = serverSocket.accept();
            System.out.printf("got client\n");
            process(socket);
         } catch (Exception e) {
            handle(e);
```

```
        }
    }
}

private void handle(Exception e) {
    if (!(e instanceof SocketException)) {
        e.printStackTrace();
    }
}

public void stopProcessing() {
    keepProcessing = false;
    closeIgnoringException(serverSocket);
}

void process(Socket socket) {
    if (socket == null)
        return;

    try {
        System.out.printf("Server: getting message\n");
        String message = MessageUtils.getMessage(socket);
        System.out.printf("Server: got message: %s\n", message);
        Thread.sleep(1000);
        System.out.printf("Server: sending reply: %s\n", message);
        MessageUtils.sendMessage(socket, "Processed: " + message);
        System.out.printf("Server: sent\n");
        closeIgnoringException(socket);
    } catch (Exception e) {
        e.printStackTrace();
    }

}

private void closeIgnoringException(Socket socket) {
    if (socket != null)
        try {
```

```
            socket.close();
         } catch (IOException ignore) {
         }
   }

   private void closeIgnoringException(ServerSocket serverSocket) {
      if (serverSocket != null)
         try {
            serverSocket.close();
         } catch (IOException ignore) {
         }
   }
}
```

Listing A-5 `MessageUtils.java`

```
   package common;

import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.net.Socket;

public class MessageUtils {
   public static void sendMessage(Socket socket, String message)
         throws IOException {
      OutputStream stream = socket.getOutputStream();
      ObjectOutputStream oos = new ObjectOutputStream(stream);
      oos.writeUTF(message);
      oos.flush();
   }

   public static String getMessage(Socket socket) throws IOException {
      InputStream stream = socket.getInputStream();
      ObjectInputStream ois = new ObjectInputStream(stream);
      return ois.readUTF();
```

```
    }
  }
```

Client/Server Using Threads

Changing the server to use threads simply requires a change to the process message (new lines are emphasized to stand out):

```
    void process(final Socket socket) {
    if (socket == null)
       return;

    Runnable clientHandler = new Runnable() {
       public void run() {

          try {
             System.out.printf("Server: getting message\n");
             String message = MessageUtils.getMessage(socket);
             System.out.printf("Server: got message: %s\n", message);
              Thread.sleep(1000);
              System.out.printf("Server: sending reply: %s\n", message);
              MessageUtils.sendMessage(socket, "Processed: " + message);
             System.out.printf("Server: sent\n");
             closeIgnoringException(socket);
           } catch (Exception e) {
             e.printStackTrace();
          }
       }
    };

    Thread clientConnection = new Thread(clientHandler);
    clientConnection.start();
  }
```