Writing multithreaded Java applications

Learn to avoid problems common in concurrent programming

Alex Roetter (aroetter@CS.Stanford.edu)
Software engineer
Teton Data Systems

01 February 2001

The Java Thread API allows programmers to write applications that can take advantage of multiple processors and perform background tasks while still retaining the interactive feel that users require. Alex Roetter introduces the Java Thread API, outlines issues involved in multithreading, and offers solutions to common problems.

When you are writing graphical programs that use AWT or Swing, multithreading is a necessity for all but the most trivial programs. Thread programming poses many difficulties, and many developers often find themselves falling prey to problems such as incorrect application behavior and deadlocked applications.

In this article, we'll explore the problems associated with multithreading and discuss solutions to the most common pitfalls.

What are threads?

A program or process can contain multiple threads that execute instructions according to program code. Like multiple processes that can run on one computer, multiple threads appear to be doing their work in parallel. Implemented on a multi-processor machine, they actually *can* work in parallel. Unlike processes, threads share the same address space; that is, they can read and write the same variables and data structures.

When you're writing multithreaded programs, you must take great care that no one thread disturbs the work of any other thread. You can liken this approach to an office where the workers function independently and in parallel except when they need to use shared office resources or communicate with one another. One worker can speak to another worker only if the other worker is "listening" and they both speak the same language. Additionally, a worker can't use a copy machine until it is free and in a useable state (no half-completed copy jobs, paper jams, and so on). As we work through this article, you'll see how you can get threads to coordinate and cooperate in a Java program much like workers in a well-behaved organization.

In a multithreaded program, threads are obtained from the pool of available ready-to-run threads and run on the available system CPUs. The OS can move threads from the processor to either a ready or blocking queue, in which case the thread is said to have "yielded" the processor. Alternatively, the Java virtual machine (JVM) can manage thread movement -- under either a cooperative or preemptive model -- from a ready queue onto the processor, where the thread can begin executing its program code.

Cooperative threading allows the threads to decide when they should give up the processor to other waiting threads. The application developer determines exactly when threads will yield to other threads, allowing them to work very efficiently with one another. A disadvantage is that a malicious or poorly written thread can starve other threads while it consumes all available CPU time.

Under the *preemptive threading* model, the OS interrupts threads at any time, usually after allowing them to run for a period of time (known as a time-slice). As a result, no thread can ever unfairly hog the processor. However, interrupting threads at any time poses problems for the program developer. Using our office example, consider what would happen if a worker preempts another worker making copies halfway through her copy job: the new worker would start his copy job on a machine that already has originals on the glass or copies in the output tray. The preemptive threading model requires that threads use shared resources appropriately, while the cooperative model requires threads to share execution time. Because the JVM specification does not mandate a particular threading model, Java developers must write programs for both models. We'll see how to design programs for either model after looking a bit at threads and communication among threads.

Threads and the Java language

To create a thread using the Java language, you instantiate an object of type `Thread` (or a subclass) and send it the `start()` message. (A program can send the `start()` message to any object that implements the `Runnable` interface.) The definition of each thread's behavior is contained in its `run()` method. A run method is equivalent to `main()` in a traditional program: a thread will continue running until `run()` returns, at which point the thread dies.

Locks

Most applications require threads to communicate and synchronize their behavior to one another. The simplest way to accomplish this task in a Java program is with locks. To prevent multiple accesses, threads can acquire and release a lock before using resources. Imagine a lock on the copy machine for which only one worker can possess a key at a time. Without the key, use of the machine is impossible. Locks around shared variables allow Java threads to quickly and easily communicate and synchronize. A thread that holds a lock on an object knows that no other thread will access that object. Even if the thread with the lock is preempted, another thread cannot acquire the lock until the original thread wakes up, finishes its work, and releases the lock. Threads that attempt to acquire a lock in use go to sleep until the thread holding the lock releases it. After the lock is freed, the sleeping thread moves to the ready-to-run queue.

In Java programming, each object has a lock; a thread can acquire the lock for an object by using the `synchronized` keyword. Methods, or synchronized blocks of code, can only be executed by one thread at a time for a given instantiation of a class, because that code requires obtaining the object's lock before execution. Continuing with our copier analogy, to avoid clashing copiers, we can simply synchronize access to the copier resource, allowing only one worker access at a time, as shown in the following code sample. We achieve this by having methods (in the `Copier` object) that modify the copier state be declared as synchronized methods. Workers that need to use a `Copier` object have to wait in line because only one thread per Copier object can be executing synchronized code.

```
class CopyMachine {

   public synchronized void makeCopies(Document d, int nCopies) {
      //only one thread executes this at a time
   }

   public void loadPaper() {
      //multiple threads could access this at once!

      synchronized(this) {
         //only one thread accesses this at a time
         //feel free to use shared resources, overwrite members, etc.
      }
   }
}
```

Fine-grain locks

Often, using a lock at the object level is too coarse. Why lock up an entire object, disallowing access to any other synchronized method for only brief access to shared resources? If an object has multiple resources, it's unnecessary to lock all threads out of the whole object in order to have one thread use only a subset of the thread's resources. Because every object has a lock, we can use dummy objects as simple locks, as shown here:

```
class FineGrainLock {

   MyMemberClass x, y;
   Object xlock = new Object(), ylock = new Object();

   public void foo() {
      synchronized(xlock) {
         //access x here
      }

      //do something here - but don't use shared resources

      synchronized(ylock) {
         //access y here
      }
   }

   public void bar() {
      synchronized(xlock) {
         synchronized(ylock) {
            //access both x and y here
         }
      }
      //do something here - but don't use shared resources
   }
```

```
}
```

These methods need not be synchronized at the method level by declaring the whole method with the `synchronized` keyword; they are using the member locks, not the object-wide lock that a synchronized method acquires.

Semaphores

Frequently, several threads will need to access a smaller number of resources. For example, imagine a number of threads running in a Web server answering client requests. These threads need to connect to a database, but only a fixed number of available database connections are available. How can you assign a number of database connections to a larger number of threads efficiently? One way to control access to a pool of resources (rather than just with a simple one-thread lock) is to use what is known as a counting semaphore. A *counting semaphore* encapsulates managing the pool of available resources. Implemented on top of simple locks, a semaphore is a thread-safe counter initialized to the number of resources available for use. For example, we would initialize a semaphore to the number of database connections available. As each thread acquires the semaphore, the number of available connections is decremented by one. Upon consumption of the resource, the semaphore is released, incrementing the counter. Threads that attempt to acquire a semaphore when all the resources managed by the semaphore are in use simply block until a resource is free.

A common use of semaphores is in solving the "consumer-producer problem." This problem occurs when one thread is completing work that another thread will use. The consuming thread can only obtain more data after the producing thread finishes generating it. To use a semaphore in this manner, you create a semaphore with the initial value of zero and have the consuming thread block on the semaphore. For each unit of work completed, the producing thread signals (releases) the semaphore. Each time a consumer consumes a unit of data and needs another, it attempts to acquire the semaphore again, resulting in the value of the semaphore always being the number of units of completed work ready for consumption. This approach is more efficient than having a consuming thread wake up, check for completed work, and sleep if nothing is available.

Though semaphores are not directly supported in the Java language, they are easily implemented on top of object locks. A simple implementation follows:

```
class Semaphore {
   private int count;
   public Semaphore(int n) {
      this.count = n;
   }

   public synchronized void acquire() {
      while(count == 0) {
         try {
            wait();
         } catch (InterruptedException e) {
            //keep trying
         }
      }
      count--;
   }
```

```
   public synchronized void release() {
      count++;
      notify(); //alert a thread that's blocking on this semaphore
   }
}
```

Common locking problems

Unfortunately, the use of locks brings with it many problems. Let's take a look at some common problems and their solutions:

Deadlocking

Deadlocking is a classic multithreading problem in which all work is incomplete because different threads are waiting for locks that will never be released. Imagine two threads, which represent two hungry people who must share one fork and knife and take turns eating. They each need to acquire two locks: one for the shared fork resource and one for the shared knife resource. Imagine if thread "A" acquires the knife and thread "B" acquires the fork. Thread A will now block waiting for the fork, while thread B blocks waiting for the knife, which thread A has. Though a contrived example, this sort of situation occurs often, albeit in scenarios much harder to detect. Although difficult to detect and hash out in every case, by following these few rules, a system's design can be free of deadlocking scenarios:

- Have multiple threads acquire a group of locks in the same order. This approach eliminates problems where the owner of X is waiting for the owner of Y, who is waiting for X.
- Group multiple locks together under one lock. In our case, create a silverware lock that must be acquired before either the fork or knife is obtained.
- Label resources with variables that are readable without blocking. After the silverware lock is acquired, a thread could examine variables to see if a complete set of silverware is available. If so, it could obtain the relevant locks; if not, it could release the master silverware lock and try again later.
- Most importantly, design the entire system thoroughly before writing code. Multithreading is difficult, and a thorough design before you start to code will help avoid difficult-to-detect locking problems.

Volatile Variables

The `volatile` keyword was introduced to the language as a way around optimizing compilers. Take the following code for example:

```
class VolatileTest {

   boolean flag;
   public void foo() {
      flag = false;
      if(flag) {
         //this could happen
      }
   }
}
```

An optimizing compiler might decide that the body of the `if` statement would never execute, and not even compile the code. If this class were accessed by multiple threads, `flag` could be set by

another thread after it has been set in the previous code, but before it is tested in the `if` statement. Declaring variables with the `volatile` keyword tells the compiler not to optimize out sections of code by predicting the value of the variable at compile time.

Inaccessible threads
Occasionally threads have to block on conditions other than object locks. IO is the best example of this problem in Java programming. When threads are blocked on an IO call inside an object, that object must still be accessible to other threads. That object is often responsible for canceling the blocking IO operation. Threads that make blocking calls in a synchronized method often make such tasks impossible. If the other methods of the object are also synchronized, that object is essentially frozen while the thread is blocked. Other threads will be unable to message the object (for example, to cancel the IO operation) because they cannot acquire the object lock. Be sure to not synchronize code that makes blocking calls, or make sure that a non-synchronized method exists on an object with synchronized blocking code. Although this technique requires some care to ensure that the resulting code is still thread safe, it allows objects to be responsive to other threads when a thread holding its locks is blocked.

Designing for different threading models

The determination of whether the threading model is preemptive or cooperative is up to the implementers of the virtual machine and can vary across different implementations. As a result, Java developers must write programs that work under both models.

Under the preemptive model, as you learned earlier, threads can be interrupted in the middle of any section of code, except for an atomic block of code. Atomic sections are code segments that once started will be finished by the current thread before it is swapped out. In Java programming, assignment to variables smaller than 32 bits is an atomic operation, which excludes variables of types `double` and `long` (both are 64 bits). As a result, atomic operations do not need synchronization. The use of locks to properly synchronize access to shared resources is sufficient to ensure that a multithreaded program works properly with a preemptive virtual machine.

For cooperative threads, it is up to the programmer to ensure that threads give up the processor routinely so that they do not deprive other threads of execution time. One way to do this is to call `yield()`, which moves the current thread off the processor and onto the ready queue. A second approach is to call `sleep()`, which makes the thread give up the processor and does not allow it to run until the amount of time specified in the argument to sleep has passed.

As you might expect, simply placing these calls at arbitrary points in code doesn't always work. If a thread is holding a lock (because it's in a synchronized method or block of code), it does not release the lock when it calls `yield()`. This means that other threads waiting for the same lock will not get to run, even though the running thread has yielded to them. To alleviate this problem, call `yield()` when not in a synchronized method. Surround the code to be synchronized in a synchronized block within a non-synchronized method and call `yield()` outside of that block.

Another solution is to call `wait()`, which makes the processor give up the lock belonging to the object it is currently in. This approach works fine if the object is synchronized at the method level,

because it is only using that one lock. If it is using a fine-grained lock, `wait()` will not give up those locks. In addition, a thread that is blocked on a call to `wait()` will not awaken until another thread calls `notify()`, which moves the waiting thread to the ready queue. To wake up all threads that are blocking on a `wait()` call, a thread calls `notifyAll()`.

Threads and AWT/Swing

In Java programs with GUIs that use Swing and/or the AWT, the AWT event handler runs in its own thread. Developers must be careful to not tie up this GUI thread performing time-consuming work, because it is responsible for handling user events and redrawing the GUI. In other words, the program will appear frozen whenever the GUI thread is busy. Swing callbacks, such as Mouse Listeners and Action Listeners are all notified (by having appropriate methods called) by the Swing thread. This approach means that any substantial amount of work the listeners are to do must be performed by having the listener callback method spawn another thread to do the work. The goal is to get the listener callback to return quickly, allowing the Swing thread to respond to other events.

If a Swing thread is running asynchronously, responding to events and repainting the display, how can other threads modify Swing state safely? As I just mentioned, Swing callbacks run in the Swing thread; therefore, they can safely modify Swing data and draw to the screen.

But what about other changes that don't happen as a result of a Swing callback? Having a non-Swing thread modify Swing data is not thread safe. Swing provides two methods to solve this problem: `invokeLater()` and `invokeAndWait()`. To modify the Swing state, simply call either of these methods, passing a `Runnable` object that does the proper work. Because `Runnable` objects are usually their own threads, you might think that this object is spawned off as a thread to execute. In fact, this is not the case, as that too would not be thread safe. Instead, Swing puts this object in a queue and executes its run method at an arbitrary point in the future. This makes the changes to Swing state thread safe.

Wrapup

The design of the Java language makes multithreading essential for all but the simplest applets. In particular, IO and GUI programming both require multithreading to provide a seamless experience for the user. By following the simple rules outlined in this article, as well as thoroughly designing a system -- including its access to shared resources -- before you've begun programming, you can avoid many common and difficult-to-detect threading pitfalls.

Resources

- For an updated and more detailed look at threading, read Brian Goetz's excellent tutorial Introduction to Java threads.
- Brian Goetz is the the master when it comes to concurrency, and his regular series on developerWorks, *Java theory and practice,* frequently covers this important topic.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.
- Browse the technology bookstore for books on these and other technical topics.

About the author

Alex Roetter

Alex Roetter has several years of experience writing multithreaded applications in Java and other languages. He holds a B.S. in Computer Science from Stanford University. You can contact Alex at aroetter@CS.Stanford.edu.