6.1 Synchronization Terms

Programmers with a background in a particular threading system generally tend to use terms specific to that system to refer to some of the concepts we discuss in this chapter, and programmers without a background in certain threading systems may not necessarily understand the terms we use. So here's a comparison of particular terms you may be familiar with and how they relate to the terms in this chapter:

Barrier

A barrier is a rendezvous point for multiple threads: all threads must arrive at the barrier before any of them are permitted to proceed past the barrier. J2SE 5.0 supplies a barrier class, and a barrier class for previous versions of Java can be found in the Appendix A.

Condition variable

A condition variable is not actually a lock; it is a variable associated with a lock. Condition variables are often used in the context of data synchronization. Condition variables generally have an API that achieves the same functionality as Java's wait-and-notify mechanism; in that mechanism, the condition variable is actually the object lock it is protecting. J2SE 5.0 also supplies explicit condition variables, and a condition variable implementation for previous versions of Java can be found in the Appendix A. Both kinds of condition variables are discussed in Chapter 4.

Critical section

A critical section is a synchronized method or block. Critical sections do not nest like synchronized methods or blocks.

Event variable

Event variable is another term for a condition variable.

Lock

This term refers to the access granted to a particular thread that has entered a synchronized method or block. We say that a thread that has entered such a method or block has acquired the lock. As we discussed in Chapter 3, a lock is associated with either a particular instance of an object or a particular class.

Monitor

A generic synchronization term used inconsistently between threading systems. In some systems, a monitor is simply a lock; in others, a monitor is similar to the wait-and-notify mechanism.

Mutex

Another term for a lock. Mutexes do not nest like synchronization methods or blocks and generally can be used across processes at the operating system level.

Reader/writer locks

A lock that can be acquired by multiple threads simultaneously as long as the threads agree to only read from the shared data or that can be acquired by a single thread that wants to write to the shared data. J2SE 5.0 supplies a reader-writer lock class, and a similar class for previous versions of Java can be found in the Appendix A.

Semaphores

Semaphores are used inconsistently in computer systems. Many developers use semaphores to lock objects in the same way Java locks are used; this usage makes them equivalent to mutexes. A more sophisticated use of semaphores is to take advantage of a counter associated with them to nest acquisitions to the critical sections of code; Java locks are exactly equivalent to semaphores in this usage. Semaphores are also used to gain access to resources other than code. Semaphore classes that implement most of these features are available in J2SE 5.0.

6.2 Synchronization Classes Added in J2SE 5.0

You probably noticed a strong pattern while reading this list of terms: beginning with J2SE 5.0, almost all these things are included in the core Java libraries. We'll take a brief look into these J2SE 5.0 classes.

6.2.1 Semaphore

In Java, a semaphore is basically a lock with an attached counter. It is similar to the Lock interface as it can also be used to prevent access if the lock is granted; the difference is the counter. In those terms, a semaphore with a counter of one is the same thing as a lock (except that the semaphore would not nest, whereas the lock—depending on its implementation—might).

The Semaphore class keeps tracks of the number of permits it can issue. It allows multiple threads to grab one or more permits; the actual usage of the permits is up to the developer. Therefore, a semaphore can be used to represent the number of locks that can be granted. It could also be used to throttle the number of threads working in parallel, due to resource limitations such as network connections or disk space.

Let's take a look at the Semaphore interface:

```
public class Semaphore {

    public Semaphore(long permits);

    public Semaphore(long permits, boolean fair);

    public void acquire( ) throws InterruptedException;

    public void acquireUninterruptibly( );

    public void acquire(long permits) throws InterruptedException;

    public void acquireUninterruptibly(long permits);

    public boolean tryAcquire( );

    public boolean tryAcquire(long timeout, TimeUnit unit);

    public boolean tryAcquire(long permits);

    public boolean tryAcquire(long permits,

                        long timeout, TimeUnit unit);

    public void release(long permits);

    public void release( );

    public long availablePermits( );

}
```

The Semaphore interface is very similar to the Lock interface. The acquire() and release() methods are similar to the lock() and unlock()

methods of the Lock interface—they are used to grab and release permits, respectively. The tryAcquire( ) methods are similar to the tryLock() methods in that they allow the developer to try to grab the lock or permits. These methods also allow the developer to specify the time to wait if the permits are not immediately available and the number of permits to acquire or release (the default number of permits is one).

Semaphores have a few differences from locks. First, the constructor requires the specification of the number of permits to be granted. There are also methods that return the number of total and free permits. This class implements only a grant and release algorithm; unlike the Lock interface, no attached condition variables are available with semaphores. There is no concept of nesting; multiple acquisitions by the same thread acquire multiple permits from the semaphore.

If a semaphore is constructed with its fair flag set to true, the semaphore tries to allocate the permits in the order that the requests are made—as close to first-come-first-serve as possible. The downside to this option is speed: it takes more time for the virtual machine to order the acquisition of the permits than to allow an arbitrary thread to acquire a permit.

6.2.2 Barrier

Of all the different types of thread synchronization tools, the barrier is probably the easiest to understand and the least used. When we think of synchronization, our first thought is of a group of threads executing part of an overall task followed by a point at which they must synchronize their results. The barrier is simply a waiting point where all the threads can sync up either to merge results or to safely move on to the next part of the task. This is generally used when an application operates in phases. For example, many compilers make multiple passes between loading the source and generating the executable, with many interim files. A barrier, when used in this regard, can make sure that all of the threads are in the same phase.

Given its simplicity, why is the barrier not more commonly used? The functionality is simple enough that it can be accomplished with the low-level tools provided by Java. We can solve the coordination problem in two ways, without using a barrier. First, we can simply have the threads wait on a condition variable. The last thread releases the barrier by notifying all of the other threads. A second option is to simply await termination of the threads by using the join() method. Once all threads have been joined, we can start new threads for the next phase of the program.

However, in some cases it is preferable to use barriers. When using the join() method, threads are exiting and we're starting new ones. Therefore, the threads lose any state that they have stored in their previous thread object; they need to store that state prior to terminating. Furthermore, if we must always create new threads, logical operations cannot be placed together; since new threads have to be created for each subtask, the code for each subtask must be placed in separate run() methods. It may be easier to code all of the logic as one method, particularly if the subtasks are very small.

Let's examine the interface to the barrier class:

```
public class CyclicBarrier {

    public CyclicBarrier(int parties);

    public CyclicBarrier(int parties, Runnable barrierAction);

    public int await( ) throws InterruptedException, BrokenBarrierException;

    public int await(long timeout, TimeUnit unit) throws InterruptedException,

                BrokenBarrierException, TimeoutException;

    public void reset( );

    public boolean isBroken( );

    public int getParties( );

    public int getNumberWaiting( );
```

```
}
```

The core of the barrier is the await() method. This method basically behaves like the conditional variable's await() method. There is an option to either wait until the barrier releases the thread or for a timeout condition. There is no need to have a signal() method because notification is accomplished by the barrier when the correct number of parties are waiting.

When the barrier is constructed, the developer must specify the number of parties (threads) using the barrier. This number is used to trigger the barrier: the threads are all released when the number of threads waiting on the barrier is equal to the number of parties specified. There is also an option to specify an action—an object that implements the run() method. When the trigger occurs, the run() method on the barrierAction object is called prior to releasing the threads. This allows code that is not threadsafe to execute; generally, it calls the cleanup code for the previous phase and/or setup code for the next phase. The last thread that reaches the barrier—the triggering thread—is the thread that executes the action.

Each thread that calls the await() method gets back a unique return value. This value is related to the arrival order of the thread at the barrier. This value is needed for cases when the individual threads need to negotiate how to divide up work during the next phase of the process. The first thread to arrive is one less than the number of parties; the last thread to arrive will have a value of zero.

In normal usage, the barrier is very simple. All the threads wait until the number of required parties arrive. Upon arrival of the last thread, the action is executed, the threads are released, and the barrier can be reused. However, exception conditions can occur and cause the barrier to fail. When the barrier fails, the CyclicBarrier class breaks the barrier and releases all of the threads waiting on the await( ) method with a BrokenBarrierException. The barrier can be broken for a number of reasons. The waiting threads can be interrupted, a thread may break through the barrier due to a timeout condition, or an exception could be thrown by the barrier action.

In every exception condition, the barrier simply breaks, thus requiring that the individual threads resolve the matter. Furthermore, the barrier can no longer be reused until it is reinitialized. That is, part of the complex (and application-specific) algorithm to resolve the situation includes the need to reinitialize the barrier. To reinitialize the barrier, you use the reset() method. However, if there are threads already waiting on the barrier, the barrier will not initialize; in fact, it will break. Reinitialization of the barrier is complex enough that it may be safer to create a new barrier.

Finally, the CyclicBarrier class provides a few operational support methods. These methods provide informational data on the number of threads already waiting on the barrier, or whether the barrier is already broken.

6.2.3 Countdown Latch

The countdown latch implements a synchronization tool that is very similar to a barrier. In fact, it can be used instead of a barrier. It also can be used to implement a functionality that some threading systems (but not Java) support with semaphores. Like the barrier class, methods are provided that allow threads to wait for a condition. The difference is that the release condition is not the number of threads that are waiting. Instead, the threads are released when the specified count reaches zero.

The CountDownLatch class provides a method to decrement the count. It can be called many times by the same thread. It can also be called by a thread that is not waiting. When the count reaches zero, all waiting threads are released. It may be that no threads are waiting. It may be that more threads than the specified count are waiting. And any thread that attempts to wait after the latch has triggered is immediately released. The latch does not reset. Furthermore, later attempts to lower the count will not work.

Here's the interface of the countdown latch:

```
public class CountDownLatch {

    public CountDownLatch(int count);

    public void await( ) throws InterruptedException;

    public boolean await(long timeout, TimeUnit unit)

                throws InterruptedException;
```

```
    public void countDown( );

    public long getCount( );

}
```

This interface is pretty simple. The initial count is specified in the constructor. A couple of overloaded methods are provided for threads to wait for the count to reach zero. And a couple of methods are provided to control the count—one to decrement and one to retrieve the count. The boolean return value for the timeout variant of the await() method indicates whether the latch was triggered—it returns true if it is returning because the latch was released.

6.2.4 Exchanger

The exchanger implements a synchronization tool that does not really have equivalents in any other threading system. The easiest description of this tool is that it is a combination of a barrier with data passing. It is a barrier in that it allows pairs of threads to rendezvous with each other; upon meeting in pairs, it then allow the pairs to exchange one set of data with each other before separating.

This class is closer to a collection class than a synchronization tool—it is mainly used to pass data between threads. It is also very specific in that threads have to be paired up, and a specific data type must be exchanged. But this class does have its advantages. Here is its interface:

```
public class Exchanger<V> {

    public Exchanger( );

    public V exchange(V x) throws InterruptedException;

    public V exchange(V x, long timeout, TimeUnit unit)

            throws InterruptedException, TimeoutException;

}
```

The exchange() method is called with the data object to be exchanged with another thread. If another thread is already waiting, the exchange() method returns with the other thread's data. If no other thread is waiting, theexchange() method waits for one. A timeout option can control how long the calling thread waits.

Unlike the barrier class, this class is very safe to use: it will not break. It does not matter how many parties are using this class; they are paired up as the threads come in. Timeouts and interrupts also do not break the exchanger as they do in the barrier class; they simply generate an exception condition. The exchanger continues to pair threads around the exception condition.

6.2.5 Reader/Writer Locks

Sometimes you need to read information from an object in an operation that may take a fairly long time. You need to lock the object so that the information you read is consistent, but you don't necessarily need to prevent another thread from also reading data from the object at the same time. As long as all the threads are only reading the data, there's no reason why they shouldn't read the data in parallel since this doesn't affect the data each thread is reading.

In fact, the only time we need data locking is when data is being changed, that is, when it is being written. Changing the data introduces the possibility that a thread reading the data sees the data in an inconsistent state. Until now, we've been content to have a lock that allows only a single thread to access the data whether the thread is reading or writing, based on the theory that the lock is held for a short time.

If the lock needs to be held for a long time, it makes sense to consider allowing multiple threads to read the data simultaneously so that these threads don't need to compete against each other to acquire the lock. Of course, we must still allow only a single thread to write the data, and we must make sure that none of the threads that were reading the data are still active while our single writer thread is changing the internal state of the data.

Here are the classes and interfaces in J2SE 5.0 that implement this type of locking:

```
public interface ReadWriteLock {

    Lock readLock( );

    Lock writeLock( );

}



public class ReentrantReadWriteLock implements ReadWriteLock {

    public ReentrantReadWriteLock( );

    public ReentrantReadWriteLock(boolean fair);

    public Lock writeLock( );

    public Lock readLock( );

}
```

You create a reader-writer lock by instantiating an object using the ReentrantReadWriteLock class. Like the ReentrantLock class, an option allows the locks to be distributed in a fair fashion. By "fair," this class means that the lock is granted on very close to a first-come-first-serve basis. When the lock is released, the next set of readers/writer is granted the lock based on arrival time.

Usage of the lock is predictable. Readers should obtain the read lock while writers should obtain the write lock. Both of these locks are objects of the Lock class—their interface is discussed in Chapter 3. There is one major difference, however: reader-writer locks have different support for condition variables. You can obtain a condition variable related to the write lock by calling the newCondition() method; calling that method on a read lock generates an UnsupportedOperationException .

These locks also nest, which means that owners of the lock can repeatedly acquire the locks as necessary. This allows for callbacks or other complex algorithms to execute safely. Furthermore, threads that own the write lock can also acquire the read lock. The reverse is not true. Threads that own the read lock cannot acquire the write lock; upgrading the lock is not allowed. However, downgrading the lock is allowed. This is accomplished by acquiring the read lock before releasing the write lock.

Later in this chapter, we examine the topic of lock starvation in depth. Reader-writer locks have special issues in this regard.

In this section, we've examined higher-level synchronization tools provided by J2SE 5.0. These tools all provide functionality that in the past could have been implemented by the base tools provided by Java—either through an implementation by the developer or by the use of third-party libraries. These classes don't provide new functionality that couldn't be accomplished in the past; these tools are written totally in Java. In a sense, they can be considered convenience classes; that is, they are designed to make development easier and to allow application development at a higher level.

There is also a lot of overlap between these classes. A Semaphore can be used to partially simulate aLock simply by declaring a semaphore with one permit. The write lock of a reader-writer lock is practically the same as a mutually exclusive lock. A semaphore can be used to simulate a reader-writer lock, with a limited set of readers, simply by having the reader thread acquire one permit while the writer thread acquires all the permits. A countdown latch can be used as a barrier simply by having each thread decrement the count

prior to waiting.

The major advantage in using these classes is that they offload threading and data synchronization issues. Developers should design their programs at as high a level as possible and not have to worry about low-level threading issues. The possibility of deadlock, lock and CPU starvation, and other very complex issues is mitigated somewhat. Using these libraries, however, does not remove the responsibility for these problems from the developer.