From: Operating System Concepts in Java, 8th edition, 2010. by Silberschatz, Galvin, Gagne


```
        dp.takeForks(i);
        eat();
        dp.returnForks(i);
```

It is easy to show that this solution ensures that no two neighboring philosophers are eating simultaneously and that no deadlocks will occur. However, it is possible for a philosopher to starve to death. We do not present a solution to that problem but rather ask you in the chapter-ending exercises to develop one.

6.8 Java Synchronization

Now that we have provided a grounding in synchronization theory, we can describe how Java synchronizes the activity of threads, allowing the programmer to develop generalized solutions to enforce mutual exclusion between threads. When an application ensures that data remain consistent even when accessed concurrently by multiple threads, the application is said to be **thread-safe**.

6.8.1 Bounded Buffer

In Chapter 3, we described a shared-memory solution to the bounded-buffer problem. This solution suffers from two disadvantages. First, both the producer and the consumer use busy-waiting loops if the buffer is either full or empty. Second, the variable count, which is shared by the producer and the consumer, may develop a race condition, as described in Section 6.1. This section addresses these and other problems while developing a solution using Java synchronization mechanisms.

6.8.1.1 Busy Waiting and Livelock

Busy waiting was introduced in Section 6.5.2, where we examined an implementation of the acquire() and release() semaphore operations. In that section, we described how a process could block itself as an alternative to busy waiting. One way to accomplish such blocking in Java is to have a thread call the Thread.yield() method. Recall from Section 5.7 that, when a thread invokes the yield() method, the thread stays in the runnable state but allows the JVM to select another runnable thread to run. The yield() method makes more effective use of the CPU than busy waiting does.

In this instance, however, using *either* busy waiting or yielding may lead to another problem, known as **livelock**. Livelock is similar to deadlock; both prevent two or more threads from proceeding, but the threads are unable to proceed for different reasons. Deadlock occurs when every thread in a set is blocked waiting for an event that can be caused only by another blocked thread in the set. Livelock occurs when a thread continuously attempts an action that fails.

Here is one scenario that could cause livelock. Recall that the JVM schedules threads using a priority-based algorithm, favoring high-priority threads over threads with lower priority. If the producer has a priority higher than that of the consumer and the buffer is full, the producer will enter the while loop


```
// Producers call this method
public synchronized void insert(E item) {
   while (count == BUFFER_SIZE)
      Thread.yield();

   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
   ++count;
}


// Consumers call this method
public synchronized E remove() {
   E item;

   while (count == 0)
      Thread.yield();

   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   --count;

   return item;
}
```

Figure 6.27 Synchronized insert() and remove() methods.

and either busy-wait or yield() to another runnable thread while waiting for count to be decremented to less than BUFFER_SIZE. As long as the consumer has a priority lower than that of the producer, it may never be scheduled by the JVM to run and therefore may never be able to consume an item and free up buffer space for the producer. In this situation, the producer is livelocked waiting for the consumer to free buffer space. We will see shortly that there is a better alternative than busy waiting or yielding while waiting for a desired event to occur.

6.8.1.2 Race Condition

In Section 6.1, we saw an example of the consequences of a race condition on the shared variable count. Figure 6.27 illustrates how Java's handling of concurrent access to shared data prevents race conditions.

In describing this situation, we introduce a new keyword: synchronized. Every object in Java has associated with it a single lock. An object's lock may be owned by a single thread. Ordinarily, when an object is being referenced (that is, when its methods are being invoked), the lock is ignored. When a method is declared to be synchronized, however, calling the method requires owning the lock for the object. If the lock is already owned by another thread, the thread calling the synchronized method blocks and is placed in the **entry set** for the object's lock. The entry set represents the set of threads waiting for the lock to become available. If the lock is available when a synchronized

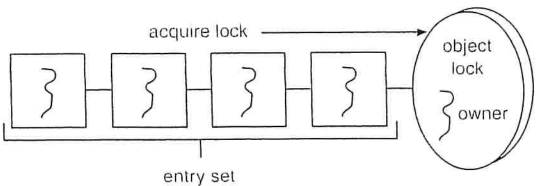

Figure 6.28 Entry set.

method is called, the calling thread becomes the owner of the object's lock and can enter the method. The lock is released when the thread exits the method. If the entry set for the lock is not empty when the lock is released, the JVM arbitrarily selects a thread from this set to be the owner of the lock. (When we say "arbitrarily," we mean that the specification does not require that threads in this set be organized in any particular order. However, in practice, most virtual machines order threads in the wait set according to a FIFO policy.) Figure 6.28 illustrates how the entry set operates.

If the producer calls the `insert()` method, as shown in Figure 6.27, and the lock for the object is available, the producer becomes the owner of the lock; it can then enter the method, where it can alter the value of `count` and other shared data. If the consumer attempts to call the `synchronized remove()` method while the producer owns the lock, the consumer will block because the lock is unavailable. When the producer exits the `insert()` method, it releases the lock. The consumer can now acquire the lock and enter the `remove()` method.

6.8.1.3 Deadlock

At first glance, this approach appears at least to solve the problem of having a race condition on the variable `count`. Because both the `insert()` method and the `remove()` method are declared `synchronized`, we have ensured that only one thread can be active in either of these methods at a time. However, lock ownership has led to another problem.

Assume that the buffer is full and the consumer is sleeping. If the producer calls the `insert()` method, it will be allowed to continue, because the lock is available. When the producer invokes the `insert()` method, it sees that the buffer is full and performs the `yield()` method. All the while, the producer still owns the lock for the object. When the consumer awakens and tries to call the `remove()` method (which would ultimately free up buffer space for the producer), it will block because it does not own the lock for the object. Thus, both the producer and the consumer are unable to proceed because (1) the producer is blocked waiting for the consumer to free space in the buffer and (2) the consumer is blocked waiting for the producer to release the lock.

By declaring each method as `synchronized`, we have prevented the race condition on the shared variables. However, the presence of the `yield()` loop has led to a possible deadlock.


```
// Producers call this method
public synchronized void insert(E item) {
   while (count == BUFFER_SIZE) {
      try {
         wait();
      }
      catch (InterruptedException e) { }
   }

   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
   ++count;

   notify();
}

// Consumers call this method
public synchronized E remove() {
   E item;

   while (count == 0) {
      try {
         wait();
      }
      catch (InterruptedException e) { }
   }

   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   --count;

   notify();

   return item;
}
```

Figure 6.29 `insert()` and `remove()` methods using `wait()` and `notify()`.

6.8.1.4 Wait and Notify

Figure 6.29 addresses the `yield()` loop by introducing two new Java methods: `wait()` and `notify()`. In addition to having a lock, every object also has associated with it a **wait set** consisting of a set of threads. This wait set is initially empty. When a thread enters a `synchronized` method, it owns the lock for the object. However, this thread may determine that it is unable to continue because a certain condition has not been met. That will happen, for example, if the producer calls the `insert()` method and the buffer is full. The thread then will release the lock and wait until the condition that will allow it to continue is met, thus avoiding the previous deadlock situation.

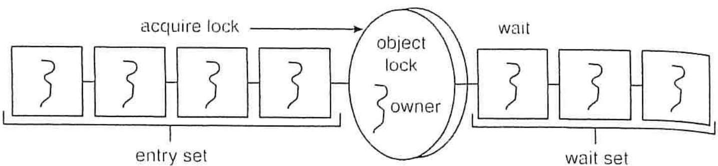

Figure 6.30 Entry and wait sets.

When a thread calls the `wait()` method, the following happens:

1. The thread releases the lock for the object.
2. The state of the thread is set to blocked.
3. The thread is placed in the wait set for the object.

Consider the example in Figure 6.29. If the producer calls the `insert()` method and sees that the buffer is full, it calls the `wait()` method. This call releases the lock, blocks the producer, and puts the producer in the wait set for the object. Because the producer has released the lock, the consumer ultimately enters the `remove()` method, where it frees space in the buffer for the producer. Figure 6.30 illustrates the entry and wait sets for a lock. (Note that `wait()` can result in an `InterruptedException` being thrown. We will cover this in Section 6.8.6.)

How does the consumer thread signal that the producer may now proceed? Ordinarily, when a thread exits a `synchronized` method, the departing thread releases only the lock associated with the object, possibly removing a thread from the entry set and giving it ownership of the lock. However, at the end of the `synchronized insert()` and `remove()` methods, we have a call to the method `notify()`. The call to `notify()`:

1. Picks an arbitrary thread T from the list of threads in the wait set
2. Moves T from the wait set to the entry set
3. Sets the state of T from blocked to runnable

T is now eligible to compete for the lock with the other threads. Once T has regained control of the lock, it returns from calling `wait()`, where it may check the value of `count` again.

Next, we describe the `wait()` and `notify()` methods in terms of the program shown in Figure 6.29. We assume that the buffer is full and the lock for the object is available.

- The producer calls the `insert()` method, sees that the lock is available, and enters the method. Once in the method, the producer determines that the buffer is full and calls `wait()`. The call to `wait()` releases the lock for the object, sets the state of the producer to blocked, and puts the producer in the wait set for the object.

- The consumer ultimately calls and enters the `remove()` method, as the lock for the object is now available. The consumer removes an item from the buffer and calls `notify()`. Note that the consumer still owns the lock for the object.
- The call to `notify()` removes the producer from the wait set for the object, moves the producer to the entry set, and sets the producer's state to runnable.
- The consumer exits the `remove()` method. Exiting this method releases the lock for the object.
- The producer tries to reacquire the lock and is successful. It resumes execution from the call to `wait()`. The producer tests the `while` loop, determines that room is available in the buffer, and proceeds with the remainder of the `insert()` method. If no thread is in the wait set for the object, the call to `notify()` is ignored. When the producer exits the method, it releases the lock for the object.

The `BoundedBuffer` class shown in Figure 6.31 represents the complete solution to the bounded-buffer problem using Java synchronization. This class may be substituted for the `BoundedBuffer` class used in the semaphore-based solution to this problem in Section 6.6.1.

```
public class BoundedBuffer<E> implements Buffer<E>
{
   private static final int BUFFER_SIZE = 5;

   private int count, in, out;
   private E[] buffer;

   public BoundedBuffer() {
     // buffer is initially empty
     count = 0;
     in = 0;
     out = 0;
     buffer = (E[]) new Object[BUFFER_SIZE];
   }

   public synchronized void insert(E item) {
     // Figure 6.29
   }

   public synchronized E remove() {
     // Figure 6.29
   }
}
```

Figure 6.31 Bounded buffer.

```
/**
 * myNumber is the number of the thread
 * that wishes to do some work
 */
public synchronized void doWork(int myNumber) {
   while (turn != myNumber) {
      try {
         wait();
      }
      catch (InterruptedException e) { }
   }

   // Do some work for awhile . . .

   /**
    * Finished working. Now indicate to the
    * next waiting thread that it is their
    * turn to do some work.
    */
   turn = (turn + 1) % 5;

   notify();
}
```

Figure 6.32 doWork() method.

6.8.2 Multiple Notifications

As described in Section 6.8.1.4, the call to `notify()` arbitrarily selects a thread from the list of threads in the wait set for an object. This approach works fine when only one thread is in the wait set, but consider what can happen when there are multiple threads in the wait set and more than one condition for which to wait. It is possible that a thread whose condition has not yet been met will be the thread that receives the notification.

Suppose, for example, that there are five threads {$T0$, $T1$, $T2$, $T3$, $T4$} and a shared variable `turn` indicating which thread's turn it is. When a thread wishes to do work, it calls the `doWork()` method in Figure 6.32. Only the thread whose number matches the value of `turn` can proceed; all other threads must wait their turn.

Assume the following:

- `turn = 3`.
- $T1$, $T2$, and $T4$ are in the wait set for the object.
- $T3$ is currently in the `doWork()` method.

Example of notifyAll

When thread $T3$ is done, it sets `turn` to 4 (indicating that it is $T4$'s turn) and calls `notify()`. The call to `notify()` arbitrarily picks a thread from the wait set. If $T2$ receives the notification, it resumes execution from the call to `wait()` and tests the condition in the `while` loop. $T2$ sees that this is not its

turn, so it calls `wait()` again. Ultimately, $T3$ and $T0$ will call `doWork()` and will also invoke the `wait()` method, since it is the turn for neither $T3$ nor $T0$. Now, all five threads are blocked in the wait set for the object. Thus, we have another deadlock to handle.

Because the call to `notify()` arbitrarily picks a single thread from the wait set, the developer has no control over which thread is chosen. Fortunately, Java provides a mechanism that allows all threads in the wait set to be notified. The `notifyAll()` method is similar to `notify()`, except that *every* waiting thread is removed from the wait set and placed in the entry set. If the call to `notify()` in `doWork()` is replaced with a call to `notifyAll()`, when $T3$ finishes and sets `turn` to 4, it calls `notifyAll()`. This call has the effect of removing $T1$, $T2$, and $T4$ from the wait set. The three threads then compete for the object's lock once again. Ultimately, $T1$ and $T2$ call `wait()`, and only $T4$ proceeds with the `doWork()` method.

In sum, the `notifyAll()` method is a mechanism that wakes up all waiting threads and lets the threads decide among themselves which of them should run next. In general, `notifyAll()` is a more expensive operation than `notify()` because it wakes up all threads, but it is regarded as a more conservative strategy appropriate for situations in which multiple threads may be in the wait set for an object.

```
public class Database implements ReadWriteLock
{
   private int readerCount;
   private boolean dbWriting;

   public Database() {
      readerCount = 0;
      dbWriting = false;
   }

   public synchronized void acquireReadLock() {
      // Figure 6.34
   }

   public synchronized void releaseReadLock() {
      // Figure 6.34
   }

   public synchronized void acquireWriteLock() {
      // Figure 6.35
   }

   public synchronized void releaseWriteLock() {
      // Figure 6.35
   }
}
```

Figure 6.33 Solution to the readers–writers problem using Java synchronization.

```
public synchronized void acquireReadLock() {
    while (dbWriting == true) {
        try {
            wait();
        }
        catch(InterruptedException e) { }
    }

    ++readerCount;
}

public synchronized void releaseReadLock() {
    --readerCount;

    /**
     * The last reader indicates that
     * the database is no longer being read.
     */
    if (readerCount == 0)
        notify();
}
```

Figure 6.34 Methods called by readers.

In the following section, we look at a Java-based solution to the readers-writers problem that requires the use of both `notify()` and `notifyAll()`.

6.8.3 A Java-Based Solution to the Readers-Writers Problem

We can now provide a solution to the first readers–writers problem by using Java synchronization. The methods called by each reader and writer thread are defined in the `Database` class in Figure 6.33, which implements the `ReadWriteLock` interface shown in Figure 6.17. The `readerCount` keeps track of the number of readers; a value > 0 indicates that the database is currently being read. `dbWriting` is a boolean variable indicating whether the database is currently being accessed by a writer. `acquireReadLock()`, `releaseReadLock()`, `acquireWriteLock()`, and `releaseWriteLock()` are all declared as `synchronized` to ensure mutual exclusion to the shared variables.

When a writer wishes to begin writing, it first checks whether the database is currently being either read or written. If the database is being read or written, the writer enters the wait set for the object. Otherwise, it sets `dbWriting` to `true`. When a writer is finished, it sets `dbWriting` to `false`. When a reader invokes `acquireReadLock()`, it first checks whether the database is currently being written. If the database is unavailable, the reader enters the wait set for the object; otherwise, it increments `readerCount`. The final reader calling `releaseReadLock()` invokes `notify()`, thereby notifying a waiting writer. When a writer invokes `releaseWriteLock()`, however, it calls the `notifyAll()` method rather than `notify()`. Consider the effect on readers. If several readers wish to read the database while it is being written, and the


```
public synchronized void acquireWriteLock() {
    while (readerCount > 0 || dbWriting == true) {
        try {
            wait();
        }
        catch(InterruptedException e) { }
    }

    /**
     * Once there are no readers or a writer,
     * indicate that the database is being written.
     */
    dbWriting = true;
}

public synchronized void releaseWriteLock() {
    dbWriting = false;

    notifyAll();
}
```

Figure 6.35 Methods called by writers.

writer invokes `notify()` once it has finished writing, only one reader will receive the notification. Other readers will remain in the wait set even though the database is available for reading. By invoking `notifyAll()`, a departing writer is ensured of notifying all waiting readers.

6.8.4 Block Synchronization

The amount of time between when a lock is acquired and when it is released is defined as the **scope** of the lock. Java also allows blocks of code to be declared as `synchronized`, because a `synchronized` method that has only a small percentage of its code manipulating shared data may yield a scope that is too large. In such an instance, it may be better to synchronize only the block of code that manipulates shared data than to synchronize the entire method. Such a design results in a smaller lock scope. Thus, in addition to declaring `synchronized` methods, Java also allows block synchronization, as illustrated in Figure 6.36. Access to the `criticalSection()` method in Figure 6.36 requires ownership of the lock for the `mutexLock` object.

We can also use the `wait()` and `notify()` methods in a synchronized block. The only difference is that they must be invoked with the same object that is being used for synchronization. This approach is shown in Figure 6.37.

6.8.5 Synchronization Rules

The `synchronized` keyword is a straightforward construct, but it is important to know a few rules about its behavior.

```
Object mutexLock = new Object();
. . .
public void someMethod() {
   nonCriticalSection();

   synchronized(mutexLock) {
      criticalSection();
   }

   remainderSection();
}
```

Figure 6.36 Block synchronization.

Java's rules on "synchronized"

1. A thread that owns the lock for an object can enter another `synchronized` method (or block) for the same object. This is known as a **recursive** or **reentrant** lock.
2. A thread can nest `synchronized` method invocations for different objects. Thus, a thread can simultaneously own the lock for several different objects.
3. If a method is not declared `synchronized`, then it can be invoked regardless of lock ownership, even while another `synchronized` method for the same object is executing.
4. If the wait set for an object is empty, then a call to `notify()` or `notifyAll()` has no effect.
5. `wait()`, `notify()`, and `notifyAll()` may only be invoked from `synchronized` methods or blocks; otherwise, an `IllegalMonitorStateException` is thrown.

It is also possible to declare `static` methods as `synchronized`. This is because, along with the locks that are associated with object instances, there is a single **class lock** associated with each class. Thus, for a given class, there

```
Object mutexLock = new Object();
. . .
synchronized(mutexLock) {
   try {
      mutexLock.wait();
   }
   catch (InterruptedException ie) { }
}

synchronized(mutexLock) {
   mutexLock.notify();
}
```

Figure 6.37 Block synchronization using `wait()` and `notify()`.