Chapter 4 Monitors

Monitors, at their most general, are classes that control threads' access to resources. Java synchronization is based on the variety of monitors that were developed in the early 1970s by Per Brinch-Hansen and C. A. R. Hoare, but Java synchronization is not as well developed as Brinch-Hansen and Hoare monitors are.

In this chapter, we will compare Java's synchronization to monitors. We will present our Monitor class. This class will allow you to use full-monitor facilities in Java, although it will not allow the compiler to check your program for correctness, as Brinch-Hansen's and Hoare's do. The implementation of the `Monitor` class will itself be instructive on how to use `Semaphores`.

Real Monitors and Java Monitors

When Brinch-Hansen and Hoare did their work, they were faced with the following problems: Concurrent programs were unreliable and hard to write. Semaphores were already understood; Dijkstra had developed them half a decade earlier, but semaphores were often used incorrectly, and a compiler could provide no help on using them.

What Brinch-Hansen was after was a facility for synchronizing concurrent programs that would be easy to use and could also be checked by a compiler.

The first requirement for this facility is that while they are concurrently executing, threads (processes) must be accessing different sets of variables. When they have to access the same variables, they must be granted mutually exclusive access. Moreover, one thread must be able to wait for a condition that another thread can cause.

Brinch-Hansen and Hoare called their solution *monitors*. They gave this definition of a monitor:

A monitor is essentially a shared class with explicit queues.

Thus, it has two components: *a shared class* and *explicit queues*. We will consider them one at a time.

Shared Classes. The effect of a shared class can occur in Java by creating a class in which all fields are private and all methods are synchronized.

In a monitor, the only variables that can be accessed by more than one thread are the fields of a monitor. Since the fields are private, they cannot be accessed outside the monitor. Since the methods of the monitor are synchronized, the accesses inside the monitor are mutually exclusive. This meets the requirement that threads have mutually exclusive access to shared variables.

Explicit Queues. Threads must be able to wait for other threads to cause conditions. In a monitor, the conditions are represented by *condition variables.* These variables contain queues of threads waiting for the conditions to hold.

In the bounded-buffer example, there are two conditions that a thread could be waiting for. If a thread wishes to read a value from the buffer, the buffer must first contain a value. If a thread wishes to place a value in the buffer, the buffer must not be full.

Java has only one queue for threads waiting at an object, so in our bounded-buffer code, we must notify *all* waiting threads when we put a value into the buffer (making it nonempty) or remove a value (making it nonfull). We must wake up all of the threads in order to see if the buffer permits their execution. This is inefficient.

Using true monitors, we would use two condition variables. When we put a value in the queue, we would only wake up one thread waiting for the queue to be nonempty. When we remove a value, we would only wake up one thread waiting for the queue to be nonfull. The code to do this in a dialect of Pascal might be

```
shared class BoundedBuffer (size: integer)
var
        notEmpty, notFull: Condition;
        hd, tl: integer;
        buffer: array 0 .. size-1 of Object;
procedure put(v: Object)
        begin
                if tl - hd >= buffer.length then
                     await notFull;
                buffer[tl mod buffer.length] := v;
                tl:=tl+1;
                signal notEmpty;
        end
function get(): Object
        begin
                var v: Object;
                if tl=hd then await notEmpty;
                v := buffer[hd mod buffer.length];
                hd:=hd+1;
                signal notFull;
                return v;
        end
begin
                hd:=0;
                tl:=0;
end.
```

Notice that not only is there more than one queue that a thread can wait on, but also that the threads do not wait in loops. In Java, a thread waits in a `while` loop testing the condition, for example,

```
    while (!condition) wait();
```

In Java, the thread awakened by `notify()` or `notifyAll()` does not get immediate control of the lock. By the time the thread runs, the condition may no longer be true, so it must check the condition again. With a monitor, however, the shared object is immediately handed off to the thread that has been awakened, so that when the thread starts running, the condition is guaranteed to still be true.

Overall. Monitors provide greater security from system breakdown than Java does. Compilers for languages that support monitors can assure that all fields that can be accessed by more than one thread are protected by monitors. They can also assure mutual exclusion to those fields. They can check for the possibility of deadlock due to cycles in the graph of monitor calls. The implementation of these compilers can guarantee that a thread waiting for a condition will gain access to the monitor immediately, so the condition will still be true.

Despite Java's goal of reliability and catching as many errors at compile time as possible, it provides nowhere near the same level of support for writing reliable concurrent programs that monitors provide.

This led Brinch-Hansen to express a great disappointment with Java's synchronization.

He writes:

> *Java's most serious mistake was the decision to use the sequential part of the language to implement the run-time support for the parallel features.*
>
> *In 1975, Concurrent Pascal demonstrated that platform-independent parallel programs (even small operating systems) can be written as a secure programming language with monitors. It is astounding to me that Java's insecure parallelism is taken seriously by the programming language community a quarter of a century after the invention of monitors and Concurrent Pascal. It has no merit.*[1]

Class `Monitor` in the `Thread` Package

The thread package provides the `Monitor` class with condition variables similar to those defined by Hoare and Brinch-Hansen. The condition variables are provided by the inner class `Condition`. Of course, the syntax is not the same, and the compile-time checking provided by monitors is not possible.

To use a monitor to protect your class, you can have your class extend `Monitor`, as in the following bounded-buffer example:

```
class BoundedBuffer3 extends Monitor{
 Condition notEmpty=new Condition();
 Condition notFull=new Condition();
 volatile int hd=0,tl=0;
 Object[] buffer;
 public BoundedBuffer3(int size) {
    buffer=new Object[size];
 }
 public void put(Object v)
        throws InterruptedException {
    enter();
    if(tl - hd >= buffer.length) notFull.await();
    buffer[tl++ % buffer.length] = v;
    notEmpty.signal();
    leave();
 }
 public Object get()
        throws InterruptedException {
    enter();
    Object v;
    if (tl==hd) notEmpty.await();
    v = buffer[hd++ % buffer.length];
    notFull.signal();
    leave();
    return v;
 }
}
```

[1] Brinch-Hansen, Per, "Java's Insecure Parallelism," *ACM SIGPLAN Notices*, 34(4) April 1999.

Or, you can use a separate `Monitor` object:

```
class BoundedBuffer4 {
 Monitor mon = new Monitor();
 Monitor.Condition notEmpty = mon.new Condition();
 Monitor.Condition notFull = mon.new Condition();
 volatile int hd=0,tl=0;
 Object[] buffer;
 public BoundedBuffer4(int size) {
    buffer=new Object[size];
 }
 public void put(Object v)
        throws InterruptedException {
    mon.enter();
    if(tl - hd >= buffer.length) notFull.await();
    buffer[tl++ % buffer.length] = v;
    notEmpty.signal();
    mon.leave();
 }
 public Object get()
        throws InterruptedException {
    mon.enter();
    Object v;
    if (tl==hd) notEmpty.await();
    v = buffer[hd++ % buffer.length];
    notFull.leaveWithSignal();
    return v;
 }
}
```

A Monitor must be entered with a call to the method `enter()` and exited with a call to the method `leave()`. You may enter the same monitor more than once, for example, calling one monitor-protected method from another. Of course, you must eventually leave as many times as you enter.

`Condition` is an inner class. It must be created within a monitor object. It represents a condition that a thread may wish to wait on.

To wait for condition `C` to hold, a thread calls

```
C.await();
```

To signal that the condition holds, a thread calls

```
C.signal();
```

If one or more threads are awaiting the condition at the time of a signal, one of the threads will be given immediate control of the monitor. The thread executing the `signal()` will wait to reacquire the monitor.

To simultaneously signal a condition and leave the monitor, you can call `leaveWithSignal()`. It is more efficient than a `signal()` followed by a `leave()`, since the executing thread does not have to wait to reacquire the monitor before executing the `leave()`. The method is called with

```
C.leaveWithSignal();
```

Be sure that you declare the fields of the monitor-protected class to be `volatile`. A Java compiler is permitted to cache fields in registers. This causes problems for concurrency, since one thread could change its cached copy of a shared field without storing it into memory. Then another thread would not see the change in its own cached copy or in memory. A thread, however, *is* forced to synchronize shared fields with their copies in memory when entering or leaving synchronized sections of code or when executing `wait`, `notify`, or `notifyAll`. This permits concurrent threads to communicate. Unfortunately, when trying to use other facilities to provide synchronization, such as the `Monitor` class, the compiler doesn't know to synchronize its cache with memory, so two threads trying to communicate through a monitor-protected class may not be able to see each other's changes. But if the fields are declared `volatile`, the compiler will store into memory on assignment and fetch from memory on access. This may slow down execution, but it will permit threads to communicate.

`Monitor`*'s Methods*

A thread enters a monitor by calling the `enter()` method. It must call this before

```
public void enter()
```

accessing the fields of the protected object. The fields must be declared `volatile` for reasons already given.

A thread leaves the monitor by calling the method `leave()`. A thread must not access

```
public void leave()
           throws MonitorException
```

fields of the protected object after calling `leave()`. Leave throws `MonitorException` if the thread executing this does not have the monitor locked.

To leave the monitor temporarily, a thread may call `release()`. Release returns a `MonitorLock` object that can be used to reenter later. Note that if the thread has entered several monitors, it can release and reacquire any or all of them, no matter where it is in the program.

```
public MonitorLock release()
     throws MonitorException
```

`MonitorLock` contains the entry count, the number of times the monitor has been entered, but not yet left, and the identification of the monitor and the thread that had it locked. `MonitorLock`'s `reacquire()` method is called reentered the monitor later.

`Release` throws `MonitorException` if the thread executing this does not own the monitor.

Interface `MonitorCondition`*'s Methods*

Interface `MonitorCondition` is implemented by class `Monitor.Condition` (i.e., the local class `Condition` within class `Monitor`). To create a `MonitorCondition` in monitor `M`, call `M.new Condition()`. The purpose of `MonitorCondition` is to allow threads to wait for conditions to hold and to signal that the conditions hold.

To wait for the condition to hold, a thread calls the method `await()`. This method throws

```
public void await()
           throws java.lang.InterruptedException,
                 MonitorException
```

`InterruptedException` if interrupted while waiting, and `MonitorException` if the executing thread does not own this `Monitor`.

A thread calls the method `signal()` to signal that the condition has occurred. If there

```
public void signal()
            throws MonitorException
```

are any waiting threads, it will wake up one of them to resume execution,
hand over the monitor to it, and itself wait to reenter the monitor. `Signal` throws `MonitorException` if the executing thread does not own the `Monitor`.

The method `leaveWithSignal()` simultaneously signals a condition and leaves the monitor. It is equivalent to

```
          cond.signal(); mon.leave();
```

```
public void leaveWithSignal()
            throws MonitorException
```

If there are any waiting threads, it signals one of them to resume execution and hands over the monitor to it.

If the current thread has entered the monitor only once, `leaveWithSignal()` is faster than a call to `signal()` followed by a call to `leave()`. The executing thread does not have to enter the monitor again just to leave it. However, if this thread has entered the monitor more than once, `leaveWithSignal()` behaves like `signal()` followed by `leave()`. After the signaled thread has run, the signaling thread will reenter the monitor to complete its execution.

The method `leaveWithSignal()` throws `InterruptedException` if the thread is interrupted while trying to reenter the monitor. It throws `MonitorException` if the thread executing this is not inside the `Monitor`.

***Interface* `MonitorLock`**

The `Lock` class within `Monitor` (`Monitor.Lock`) has only one method: `reacquire().` It is used to enter the monitor again after an earlier `release()`. It

```
public void reacquire()
     throws MonitorException
```

throws `InterruptedException` if the thread is interrupted while waiting to enter. It throws `MonitorException` if the thread that called `reacquire()` wasn't the thread that released the monitor.

A `Monitor.Lock` object contains a reference to the thread that executed `release()`, the monitor it released, and the count of the number of locks it held on the monitor. The `reacquire()` method reestablishes all the locks the thread had on the monitor.

`MonitorLocks` and the methods `release()` and `reacquire()` are not part of the original specification of monitors. Almost certainly, the designers of the monitors concept would not approve of them. They were added as a possibility for avoiding deadlock. They allow a thread to release several monitors if it must wait for a condition.

Examples using `Monitor` Objects

You've already seen two implementations of a bounded buffer implemented using monitors. We'll show `Monitor` objects being used for implementations of `SimpleFuture` (see Example 3-18) and a subset of `SharedTableOfQueues` (whose full implementation is described in the chapter entitled "Shared Tables of Queues").

SimpleFuture

The code for our implementation of `SimpleFuture` using a `Monitor` object is shown in Example 4–1. Its fields are

- `value`: This is the value stored in the future. To indicate that there is currently no value present, the `value` field points to the future object itself. (The theory is that one value that the user will never wish to put in the future is the future itself.)
- `is_set`: This is a condition variable. Threads that try to get the value before it is set wait on this condition.

Recall that a future is a variable that can be assigned only once. When threads attempt to fetch a value from the future, they are delayed until the value has been assigned. In the Java implementation, we could release all threads with a `notifyAll()` method call. With monitors, we cannot awaken all at the same time.

Let's examine the implementation of methods `isSet`, `getValue`, and `setValue`.

`isSet().` The way we determine whether a value has been set is by examining the `value` field. If it references this `SimpleFuture` itself, then there is no value present.

The other conspicuous trick in `isSet()` is using `try...return...finally...leave` to leave the monitor after returning the value. This may not be as obvious as saving the value to be returned in a variable, leaving the monitor, and then executing a return statement.

`setValue().` The `if` statement at the top of the code tests to see if the future has already been assigned a value and, if so, to refuse to reassign it. If the value hasn't been set yet, `setValue()` does the assignment and hands off the monitor to one of the waiting threads (if there are any). It does *not* loop and wake them all up. Each thread that is awakened hands off the monitor to the next waiting thread.

`getValue().` Method `getValue()` is forced to wait if the value hasn't been assigned yet, but can return immediately if the value has been. It will throw an `InterruptedException` if it is interrupted while waiting. The `try`...`catch` handles leaving the monitor while interrupted.

As explained for `setValue()`, the call to `leaveWithSignal()` hands off the monitor to another waiting thread. Each of the waiting threads hands off the monitor to the next

Example 4–1 Class `SimpleFuture`

```
public class SimpleFuture extends Monitor {
 private volatile Object value;
 private Condition is_set=new Condition();
 public SimpleFuture() {value=this;}
 public SimpleFuture(Object val) {value=val;}
 public Object getValue()
    throws InterruptedException {
        enter();
      try {
        if (value==this) is_set.await();
        is_set.leaveWithSignal();
        return value;
      } catch (InterruptedException ie) {
        leave();
        throw ie;
      }
 }
 public boolean isSet() {
        enter();
        try {
            return (value!=this);
        } finally {leave();}
 }
 public void setValue(Object val) {
        enter();
        if (value!=this) {
                leave();
                return;
        }
        value=val;
        is_set.leaveWithSignal();
 }
}
```

as it exits. Since `leaveWithSignal()` is equivalent to `leave()`when no threads are waiting, it is okay to call it even if there are no threads waiting.

`SharedTableOfQueues`

This implementation of a subset of `SharedTableOfQueues` is a subclass of `Monitor`. Its overall structure is shown in Example 4–2. It uses a hash table to look up the queues. The queues themselves are represented by the local class `Folder`. (The name was chosen for historical reasons: It was used in the `Memo` system, which provides a distributed memory communication system based on a shared table of queues.)

Example 4–2 **Class `SharedTableOfQueues`**

```
class SharedTableOfQueues extends Monitor {
  Hashtable tbl=new Hashtable();
   private class Folder {
        volatile QueueComponent q=new QueueComponent();
        volatile Condition notEmpty=new Condition();
        volatile int numWaiting=0;
        }
 public void put(Object key, Object value) { ... }
 public Object get(Object key)
      throws InterruptedException {... }
 public Object getSkip(Object key) {... }
}
```

The class `Folder` is used strictly as a data structure. It has no methods. Its fields are

- `q`: This is the actual queue. It is an object of class `QueueComponent` which is used in other classes in the thread package as well as well.
- `notEmpty`: This is the `Condition` that the queue, `q`, is not empty. Threads calling `get()`wait for this condition. This condition is bound to the surrounding `SharedTableOfQueue`'s Monitor.
- `numWaiting`: This field counts the number of waiting threads. It is used to decide when to remove folders. If there are no values in the queue and no threads waiting, the folder may be removed; it will be created in exactly the same state if it is referenced again.

The methods are implemented as follows:

`put()`. The `put(key, value)` method (see Example 4–3), puts a `value` into the queue associated with a `key`. It uses a hash table to look up the queue, represented by a `Folder` object. The queue has to be created if one doesn't already exist. The `put()` method leaves the monitor signaling the folder's `notEmpty` condition.

`get()`. The `get(key)` method has to accomplish several things:

- If the `key` does not already have a folder, one is created.
- Field `numWaiting` is incremented before this thread waits for the queue to have something in it and is decremented afterwards.

Example 4–3 The `SharedTableOfQueues` method `put()`

```
public void put(Object key, Object value) {
       enter();
       Folder f = (Folder)tbl.get(key);
       if (f==null) tbl.put(key,f=new Folder());
       f.q.put(value);
       f.notEmpty.leaveWithSignal();
}
```

Example 4–4 The `SharedTableOfQueues` method `get()`

```
public Object get(Object key)
           throws InterruptedException {
  Folder f=null;
  enter();
  try {
       f=(Folder)tbl.get(key);
       if (f==null) tbl.put(key,f=new Folder());
       f.numWaiting++;
       if (f.q.isEmpty()) f.notEmpty.await();
       f.numWaiting--;
       return f.q.get();
  } finally {
       if (    f!=null &&
               f.q.isEmpty() &&
               f.numWaiting==0  )
                       tbl.remove(key);
       leave();
  }
}
```

- After getting a value from the queue, if the queue is empty and there are no threads waiting to remove something from it, `get()` removes the queue.

`getSkip()`. Method `getSkip(key)`returns immediately, whether or not the queue contained a value. If there is no folder associated with the key or if the queue is empty, it simply returns null. Its code is shown in Example 4–5.

Example 4–5 `SharedTableOfQueues` method `getSkip()`

```
public Object getSkip(Object key) {
  Folder f=null;
  enter();
  try {
     f=(Folder)tbl.get(key);
     if (f==null || f.q.isEmpty()) {
       return null;
     }
     return f.q.get();
  } finally {
       if (    f!=null &&
               f.q.isEmpty() &&
               f.numWaiting==0  )
                       tbl.remove(key);
       leave();
  }
}
```

As with `get()`, `getSkip()` removes the queue if it is empty and no other threads are waiting to get a value from it.

Implementation of `Monitor`, `Condition`, and `MonitorLock`

The overall structure of class `Monitor` is shown in Example 4–6.

The `Monitor` class has three fields. The Semaphore, `monitorEntry`, gives exclusive access to the monitor:

- `monitorEntry.down()`: locks the monitor on entry,
- `monitorEntry.up()`: unlocks the monitor on exit.

The `current` field references the thread that currently possesses the monitor. It is null if the monitor isn't currently owned by any thread. The `monitorEntryCount` field

Example 4–6 Structure of class `Monitor`.

```
public class Monitor{
 Semaphore monitorEntry=new Semaphore(1);
 volatile Thread current=null;
 volatile int monitorEntryCount=0;
 public class Condition implements MonitorCondition {
    volatile int waiting=0;
    Semaphore waitCond=new Semaphore(0);
    public void await()
      throws InterruptedException, MonitorException {...}
    public void signal()
       throws MonitorException {...}
    public void leaveWithSignal()
      throws MonitorException {...}
 }
 public void enter() {...}
 public void leave() throws MonitorException {...}
 private class Lock implements MonitorLock{
 int n = monitorEntryCount;
 Thread owner = current;
 public void reacquire() throws
       MonitorException {...}
}
 public MonitorLock release() throws MonitorException {...}
}
```

`Monitor` fields

`java.lang.Thread current;` The current thread.

`Semaphore monitorEntry;` Semaphore to lock the `Monitor`

`int monitorEntryCount;` The number of times the monitor's current owner has entered it minus the number of times it has exited it.

counts the number of times the thread that currently owns the monitor has entered it minus the number of times it has left. Upon leaving the monitor, if `monitorEntryCount` becomes zero, the monitor is unlocked.

Monitor Entry and Exit

Method `enter()` checks to see if the current thread already owns the monitor. If so, it merely increments the `monitorEntryCount`. If not, it must seize the monitor by downing the `monitorEntry` semaphore. Once it owns the monitor, it assigns itself (a reference to its `Thread` object) to the `current` field and places an initial count of unity in the `monitorEntryCount` field.

Example 4–7 The `Monitor` method `entry()`

```
public void enter() {
       if (current == Thread.currentThread())
               monitorEntryCount++;
       else {
               boolean interrupted=Thread.interrupted();
               for(;;) try {
                       monitorEntry.down(); break;
               } catch (InterruptedException ie) {
                       interrupted=true;
               }
               current = Thread.currentThread();
               monitorEntryCount=1;
               if (interrupted) current.interrupt();
       }
}
```

However, a question concerning many of these methods is, “How should they work with `InterruptedException`?” Our underlying implementation waits to acquire the monitor by downing semaphores. The `down()` method can throw an `InterruptedException`. If we just pass that through, we could throw `InterruptedException`s on `enter()`, `await()`, `signal()`, `leaveWithSignal()`, and `reacquire()`. Having to put in `try–catch` statements around all these calls would make `Monitors` inconvenient to use.

We follow the lead of Java’s monitors. `InterruptedException` will not be thrown upon entering or leaving a monitor, but can be thrown upon waiting for a condition to be signalled. Thus, `enter()`, `signal()`, `leaveWithSignal()`, and `reacquire()` should not be able to throw `InterruptedException`. They should, however, preserve the state of the thread. If the thread was interrupted when it called the method, it should still be interrupted when it returns.

This explains the boolean variable `interrupted`. This variable is set if the thread enters the method in an interrupted state, or if it is interrupted while waiting on a semaphore within the method. At the end of the method, the thread is again interrupted if the `interrupted` flag is set.

Thus, we have the following code:

```
for(;;) try {
        monitorEntry.down(); break;
} catch (InterruptedException ie) {
        interrupted=true;
}
```

The infinite loop will keep catching interrupts while the thread is trying to enter the monitor. When there is an interrupt, the `interrupted` flag is set to allow the thread to be interrupted again when it leaves the method. If the method successfully downs the `monitorEntry` semaphore, it breaks out of the loop.

Method `leave()` first checks that the thread that called it actually owns the monitor. If the thread doesn't own the monitor, it throws an exception. If it does own the monitor, it decrements `monitorEntryCount`. If the count is still greater than zero, this thread still owns the monitor. If the count goes to zero, the monitor is free, so `leave()` sets the current reference to null and ups the `monitorEntry` semaphore to allow some other thread to seize the monitor.

Example 4–8 The `Monitor` method `leave()`

```
public void leave() throws MonitorException{
       if (current!=Thread.currentThread())
           throw new MonitorException("leave()");
       monitorEntryCount--;
       if(monitorEntryCount==0) {
               current = null;
               monitorEntry.up();
       }
}
```

Method `release()` allows a thread to temporarily give up all locks the thread has on a monitor. First, it checks that the thread executing it owns the monitor. If the thread does not own the monitor, `release()` throws an exception. If it does own the monitor, `release()` creates a `Monitor.Lock` object to return. The initialization for the `Lock` object grabs copies of the `Monitor` objects fields `current` and `monitorEntryCount`. Then, `release()` releases the monitor.

Example 4–9 The `Monitor` method `release()`

```
public MonitorLock release() throws MonitorException {
       if (current!=Thread.currentThread())
           throw new MonitorException("release()");
       Lock L=new Lock();
       current = null;
       monitorEntryCount=0;
       monitorEntry.up();
       return L;
}
```

`Monitor.Condition`

The local class `Condition` within `Monitor` implements interface `MonitorCondition`. It is used to represent conditions (also known as *events*) in the Brinch-Hansen and Hoare monitors.

`Monitor.Condition` fields

`Semaphore waitCond;` The semaphore upon which the waiting threads wait.

`int waiting;` The number of threads waiting on this condition.

`Condition` has two fields. The `waitCond` field is a semaphore used to delay threads that are waiting on the condition. The `waiting` field is a count of the number of threads waiting on the semaphore. This count is important, as semaphores maintain the count of the number of up operations performed, whereas conditon variables are memoryless. As we will see in `signal()`, a signal operation can only be translated into upping the semaphore if there are threads waiting.

In the code for the methods in `Condition`, fields `current`, `monitorEntry`, and `monitorEntryCount` are fields of the surrounding `Monitor` object.

Method `await()` first checks that the executing thread owns the surrounding monitor. If the thread does not own the monitor, it throws an exception. If it does own the monitor, it releases the monitor and waits as follows:

- It saves the `monitorEntryCount` in the local variable `count` so it can restore it later.
- It marks the monitor as free by setting `current` to `null` and `monitorEntryCount` to zero.

- It increments the `waiting` field of the condition.
- It ups the `monitorEntry` semaphore to allow another thread to enter the monitor.
- It waits on the `Condition`'s `waitCond` semaphore.
- Once it's been awakened, it restores its own reference to `current` and the proper count to `monitorEntryCount`.

Example 4–10 The `Monitor.Condition` method `await()`

```
public void await()
  throws InterruptedException, MonitorException {
    if (current!=Thread.currentThread())
        throw new MonitorException("await()");
    int count = monitorEntryCount;
    monitorEntryCount = 0;
    current = null;
    waiting++;
    monitorEntry.up();
    waitCond.down();
    current=Thread.currentThread();
    monitorEntryCount=count;
}
```

Notice that `await()` increments the `waiting` field before waiting on the `waitCond` semaphore, but does not decrement it after being awakened. Methods `signal()` and `leaveWithSignal()` handle that. They decrement the count, because, as soon as they up the semaphore, they awaken the thread, even if it hasn't started running yet.

Decrementing the waiting count in `signal()` would be especially important if we were implementing Javalike monitors, where the currently executing thread continues running inside the monitor or another thread can seize the monitor before the awakened thread gets to run. Here's what could happen: Thread `X` is the only thread awaiting a condition. The condition count is one. Thread `Y` signals the condition, so `X` is made runnable. But if `Y` doesn't decrement the count, the count will stay one. Now, thread `Y` leaves the monitor, and thread `Z` seizes it. `Z` signals the same condition, and since the count is one, it ups the semaphore again. Since there is no waiting thead, the semaphore remembers the up in its internal counter. `Z` leaves, and `X` runs. But the next thread, `W`, that tries to await the condition will not wait at all, because the semaphore's counter was greater than zero.

Decrementing `waiting` in `signal()` is not as important here, because the monitor is handed off from one thread to another, and thus the monitor remains locked, the awakened thread begins running in it, and the thread that called `signal()` queues up on the `monitorEntry` semaphore to reenter the monitor.

Example 4–11 The `Monitor.Condition` method `signal()`

```
public void signal()
   throws MonitorException {
    if (current!=Thread.currentThread())
        throw new MonitorException("signal()");
    if (waiting>0) {
            waiting--;
            int count = monitorEntryCount;
            monitorEntryCount = 0;
            current = null;
            waitCond.up();
            boolean interrupted=Thread.interrupted();
            for(;;) try {
                    monitorEntry.down(); break;
            } catch (InterruptedException ie) {
                    interrupted=true;
            }
            current=Thread.currentThread();
            monitorEntryCount=count;
            if (interrupted) current.interrupt();
    }
}
```

Method `signal()` begins with the usual check that the thread executing it actually owns the monitor, throwing an exception if it doesn't.

If the `waiting` count is zero, signal performs no operation. Otherwise, it must hand off the monitor to one of the threads waiting for the condition. It does this as follows:

- It decrements the condition's `waiting` count, for the reasons just explained.
- It saves its `monitorEntryCount` to restore later.
- It clears fields `current` and `monitorEntryCount` to denote that it has no ownership of the Monitor. This is actually unnecessary.
- It wakes up one waiting thread by upping `waitCond`.
- It waits to enter the monitor again by downing `monitorEntry`. Here, in a way described by `enter()`, it preserves but ignores the fact of the thread being interrupted.

- Once it gets past `monitorEntry`, it reestablishes fields `current` and `monitorEntryCount` to denote that it owns the monitor again.

Example 4–12 The `Monitor.Condition` method `leaveWithSignal()`

```
      public void leaveWithSignal()
        throws MonitorException {
          if (current!=Thread.currentThread())
              throw new MonitorException("leaveWithSignal()");
          monitorEntryCount--;
          if (waiting>0) {
                  waiting--;
                  if (monitorEntryCount>0) {
                          int count = monitorEntryCount;
                          monitorEntryCount = 0;
                          current = null;
                          waitCond.up();
                        boolean interrupted=
                                 Thread.interrupted();
                          for(;;) try {
                                  monitorEntry.down(); break;
                          } catch (InterruptedException ie) {
                                  interrupted=true;
                          }
                          monitorEntryCount = count;
                          current=Thread.currentThread();
                          if (interrupted) current.interrupt();
                  } else {
                          current = null;
                          waitCond.up();
                  }
          } else {
                  if (monitorEntryCount==0) {
                          current = null;
                          monitorEntry.up();
                  }
          }
      }
   }
```

Method `leaveWithSignal()` begins simply enough with checking that the `current` thread owns the monitor and then decrementing its `monitorEntryCount`. Then, alas, there are four cases to consider:

1. There is a thread waiting at the condition, and `monitorEntryCount` is not zero. The `leaveWithSignal()` must behave like a `signal()`, since the current thread must continue executing in the monitor after the awakened thread has had its chance. This code was explained for method `signal()`.
2. There is a thread waiting at the condition, and `monitorEntryCount` *is* zero. The current thread can hand off the monitor and continue executing. This only requires three things:
   - **a.** decrementing the `Condition`'s `waiting` count,
   - **b.** setting `current` to `null` so the current thread can't try to enter the monitor again and think it already owns it, and
   - **c.** upping the `Condition`'s `waitCond` semaphore to wake up one waiting thread.
3. There are no threads waiting at the condition, and `monitorEntryCount` is zero. The current thread must leave the monitor. This code was explained in the method `leave()`from class `Monitor`.
4. There are no threads waiting at the condition and `monitorEntryCount` is not zero. The current thread goes on executing in the monitor. This doesn't require any code.

`Monitor.Lock`

`Lock` is a class defined in `Monitor`. You cannot create an instance of it directly, but only by calling `Monitor`'s `release()` method.

`Monitor.Lock` fields

```
int n = monitorEntryCount;
Thread owner = current;
```

Method `release()` will give up your lock on the monitor, but will save enough information in the `Monitor.Lock` object it retuns to allow you to reacquire the monitor. It must remember the following:

1. Who you are, so somebody else cannot reacquire the monitor pretending to be you, and
2. How many times you have entered the monitor so that it can reestablish the proper number of locks.

There is only one method for a `MonitorLock`. Calling `reacquire()` will relock the monitor for you. If a different thread tries to reacquire the monitor, `reacquire()` will

Example 4–13 The `Monitor.Lock` method `reacquire()`

```
public void reacquire() throws
               MonitorException {
       if (owner != Thread.currentThread())
           throw new MonitorException(
               "attempt to reacquire Monitor"+
                       " by a different thread");
       boolean interrupted=Thread.interrupted();
       for(;;) try {
               monitorEntry.down(); break;
       } catch (InterruptedException ie) {
               interrupted=true;
       }
       current = owner;
       monitorEntryCount=n;
       owner = null;
       if (interrupted) current.interrupt();
}
```

throw a `MonitorException`. Since `Monitor.Lock` class is declared within `Monitor`, `reacquire()` can access the fields of the surrounding `Monitor` object directly. Here's what it does:

- It checks that the monitor is being reacquired by the same thread that released it.
- It locks the monitor by downing the `monitorEntry` semaphore. Again, the `interrupted` flag is used to preserve the fact that the thread has been interrupted while ignoring it in this method.
- It records that it is the current owner of the monitor.
- It reestablishes the monitor entry count.
- It clears the owner of the `Monitor.Lock` object so that it cannot be used again to enter the monitor.

The Multiple Reader–Writer Monitors

Threads that share data structures can run into problems if one of the threads is writing into the structure while another thread is reading from it or if two threads try to write into it at the same time. This is because several values must typically be stored to move the data structure from one consistent state to another. If only some of the stores have been made when it is read, the reader will get confused. If more than one thread is writing into it, some of the fields may end up changed by one of the threads and other fields changed by the other. Thereafter, anything reading from it will be confused.

The easiest solution is to lock the data structure whenever accessing it. Any other thread trying to access the structure will have to wait until it is unlocked before being able to lock the structure itself.

The problem with locking the data structure is that although it is essential to prevent any thread from writing to the data structure while another thread is writing to or reading from it, there is no reason to prevent several threads from simultaneously reading. If many threads are permitted to read at the same time, we would expect better performance.

Policies

We present five policies for synchronizing threads trying to read from or write to a shared data structure:

1. The **single reader–writer monitor** locks the data structure so that only one thread can access the data structure at a time, whether the thread is a reader or a writer. It is equivalent to a semaphore.
2. The **writers-preferred monitor** gives the shared structure to a waiting writer (if there is one) and only if there is no writer waiting will it be given to readers. All waiting readers will be allowed to access the structure at the same time.
3. The **readers-preferred monitor** gives the shared structure to all waiting readers (if there are any) and gives it to a writer only if there is no reader waiting.
4. The **alternating readers–writers monitor** gives the shared structure to a waiting writer when readers have finished with it and to readers when a writer has finished with it.
5. The **take-a-number monitor** gives the shared resource to the threads in order of arrival. It requires a kind of busy waiting. The threads must check to see whether the number they've taken is now being served.

These algorithms are animated in: `http://www.toolsofcomputing.com`.

The **queued reader–writer monitor**, presented later in Chapter 8, “Shared Tables of Queues,” gives the shared resource to the threads in order of arrival. It uses a `FutureQueue` from the thread package to give the resource to the threads in a first-come, first-served order. Unlike the **take-a-number monitor**, which also gives the shared resource to the threads in order of arrival, the **queued reader–writer monitor** doesn't require busy waiting.

The writers-preferred, readers-preferred, and alternating reader–writer monitors all have complexities in design that are discussed in their individual sections.

Entering and Leaving the Critical Sections

Readers and writers must explicitly lock and unlock the shared resource. The locking is done via method calls to a monitor object that implements the desired policy, as in the following:

Reader
```
     monitor.startReading();
     try {
             ...read...
     } finally {monitor.stopReading();}
```

Writer
```
     monitor.startWriting();
     try {
             ...write...
     } finally {monitor.stopWriting();}
```
The reason for the `try...finally` statements is that the thread could throw an exception within the code to read or write. It is important to unlock shared resources upon abnormal exit.

The Single-Reader–Writer Monitor

This demonstration shows a simple lock on the data structure. Only a single reader or writer can access the shared resource at one time.

With a fair scheduling algorithm, each thread, in turn, should get a chance to run, cycling through the threads over and over. We've tried this out on a number of Java implementations. They are not fair. A small subset of the threads run while all the others wait.

The Single-Reader–Writer Monitor is equivalent to a lock or binary semaphore. Methods `startReading()` and `startWriting()` are identical, as are `stopReading()` and `stopWriting()`.

Example 4–14 **Class `SingleReaderWriter`**

```
class SingleReaderWriter extends
             MultipleReadersWritersMonitor{
int n=0; /* number readers reading and writers writing,
                        0 or 1*/
public void reset(){
    n=0;
}
public synchronized void startReading()
        throws InterruptedException{
    while (n!=0) wait();
    n=1;
}
public synchronized void stopReading(){
    n=0;
    notify();
}
public synchronized void startWriting()
        throws InterruptedException{
    while (n!=0) wait();
    n=1;
}
public synchronized void stopWriting(){
    n=0;
    notify();
}
public String getMonitorInfo(){
    return "Single Reader Writer Monitor";
}
}
```

The Readers-Preferred Monitor

The Readers-Preferred Monitor gives the resource to a reader (if there are any available). Only if there are no readers present will the resource be given to a writer.

If you run this animation with threads choosing randomly whether to read or write, you will notice that most threads end up waiting to write for a long time. If you run it with a fixed number of readers, you will observe the animation alternately giving the resource to a batch of readers and then to a writer while the readers are inactive.

State of the monitor. The state of the readers-preferred monitor is contained in four variables:

`nr`: The number of threads currently reading `nr` > = 0.

`nw`: The number of threads currently writing (zero or one).

`nrtotal`: The number of threads either reading or waiting to read `nrtotal` > = `nr`.

`nwtotal`: The number of threads either writing or waiting to write.

Example 4–15 The `ReadersPreferredMonitor` method `startReading()`

```
public synchronized void startReading()
throws InterruptedException{
       nrtotal++;
       while (nw!=0) wait();
       nr++;
}
```

`startReading()`. If a thread tries to start reading, it must wait until there are no threads currently writing; in other words, it must wait until `nw` is zero.

Example 4–16 The `ReadersPreferredMonitor` method `startWriting()`

```
public synchronized void startWriting()
throws InterruptedException{
       nwtotal++;
       while (nrtotal+nw != 0) wait();
       nw=1;
}
```

`startWriting()`. To start writing, a thread must wait until there are no other threads reading, waiting to read, or writing (i.e., `nrtotal` = `nw` = zero).

Example 4–17 The `ReadersPreferredMonitor` method `stopReading()`

```
public synchronized void stopReading(){
       nr--; nrtotal--;
       if (nrtotal==0) notify();
}
```

`stopReading()`. When the last present reader finishes reading, it wakes up a waiting writer (if there are any present), which will seize the monitor and start writing.

Example 4–18 The `ReadersPreferredMonitor` method `startWriting()`

```
        public synchronized void stopWriting(){
               nw=0; nwtotal--;
               notifyAll();
        }
```

`stopWriting()`. When a writer finishes writing, it wakes up all waiting readers and writers to let them compete for the monitor.

The Writers-Preferred Monitor

A Writers-Preferred Monitor gives the resource to a writer if any is waiting. Only if there is no writer available will it be given to readers.

If you run this animation with threads choosing randomly whether to read or write, you will observe that more and more threads accumulate waiting to read until there are no writers left. Then, all the waiting readers will run.

If you run it with a fixed set of readers, a batch of readers may run before the first writer enters the monitor, but thereafter, typically, all the writers will run to completion before any readers run again. (You can set the number of readers and rest time to give the readers a better chance.)

State of the monitor. The state of the monitor is contained in four variables:

`nr`: The number of threads currently reading.

`nw`: The number of threads currently writing (zero or one).

`nrtotal`: The number of threads either reading or waiting to read. (`nrtotal` ≥ `nr`).

`nwtotal`: The number of threads either writing or waiting to write.

`startReading()`. If a thread tries to start reading, it must wait until there are no threads either writing or waiting to write; in other words, it must wait until `nwtotal` is zero.

`startWriting()`. To start writing, a thread must wait until there are no other threads reading or writing, indicated by `nr` and `nw` both equaling zero.

Example 4–19 The `WritersPreferredMonitor` method `startReading`

```
public synchronized void startReading()
        throws InterruptedException{
    nrtotal++;
    while (nwtotal!=0) wait();
    nr++;
}
```

Example 4–20 The `WritersPreferredMonitor` method `startWriting`

```
public synchronized void startWriting()
        throws InterruptedException{
    nwtotal++;
    while (nr+nw != 0) wait();
    nw=1;
}
```

Example 4–21 The `WritersPreferredMonitor` method `stopReading()`

```
public synchronized void stopReading(){
    nr--; nrtotal--;
    if (nr==0) notifyAll();
}
```

`stopReading().` When the last reader of a group finishes reading, it wakes up all waiting readers and writers. If there are any waiting writers, no reader will be able to start reading, but will wait again. Eventually, a writer will seize the monitor and start writing.

Example 4–22 The `WritersPreferredMonitor` method `stopWriting()`

```
public synchronized void stopWriting(){
    nw=0; nwtotal--;
    notifyAll();
}
```

`stopWriting().` When a writer finishes writing, it wakes up all waiting readers and writers to let them compete for the monitor.

The Alternating Readers–Writers Monitor

The idea of the Alternating Readers–Writers Monitor is that readers and writers should take turns. Trying to specify this a bit more carefully, we come up with the following elements:

- General alternation. A batch of readers run, followed by a single writer, followed by another batch of readers, etc.
- Immediate access. If a reader arrives and there are no writers present, it is given access to the resource immediately. If a writer arrives and neither readers nor writers own the resource, it is allocated the resource immediately.

State of the monitor. The state of an Alternating Readers–Writers Monitor is contained in four variables:

`nr[2]`: The number of threads currently reading.

`thisBatch`: Index in `nr` of the batch of readers currently reading (zero or one).

`nextBatch`: Index in `nr` of the batch of readers waiting to read (always `1-thisBatch`).

`nw`: The number of threads currently writing (zero or one).

`nwtotal`: The number of threads either writing or waiting to write.

Example 4–23 The `AlternatingReadersWritersMonitor` method `startReading()`

```
public synchronized void startReading()
        throws InterruptedException{
    if (nwtotal==0) nr[thisBatch]++;
    else {
        nr[nextBatch]++;
        int myBatch=nextBatch;
        while (thisBatch!=myBatch) wait();
    }
}
```

`startReading().` When a thread tries to start reading, it checks first to see if there are any writers present. If there are none, it starts reading immediately, recording its presence in `nr[thisBatch]`.

If there are writers present, it must wait until one writer has run, so it adds itself to the number of readers in the next batch by incrementing `nr[nextBatch]`. It saves the value of `nextBatch` in the variable `myBatch` and waits until `thisBatch` equals `myBatch`, which will indicate that a writer has finished running.

Example 4–24 The `AlternatingReadersWritersMonitor` method `stopReading()`

```
public synchronized void stopReading(){
    nr[thisBatch]--;
    if (nr[thisBatch]==0) {
        notifyAll();
    }
}
```

`stopReading().` When a thread stops reading, it subtracts itself from `nr[thisBatch]`. It then notifies all waiting threads to try accessing the resources. Readers will wait again, since `thisBatch` has not been set to the value they are waiting for, but one of the writers will be able to start writing.

Example 4–25 The `AlternatingReadersWritersMonitor` method `startWriting()`

```
public synchronized void startWriting()
        throws InterruptedException{
    nwtotal++;
    while (nr[thisBatch]+nw != 0) wait();
    nw=1;
}
```

`startWriting().` When a writer attempts to start writing, it adds itself to the total number of writers present. Then, it waits until no threads are reading or writing. Once that condition is true, it sets `nw` to one to indicate that it is writing and begins writing.

Example 4–26 The `AlternatingReadersWritersMonitor` method `stopWriting()`

```
public synchronized void stopWriting(){
    nw=0; nwtotal--;
    int tmp=thisBatch;
    thisBatch=nextBatch;
    nextBatch=tmp;
    notifyAll();
}
```

`stopWriting().` When a thread stops writing, it sets `nw` back to zero to indicate that no thread is writing, and it subtracts itself from the total number of writers present. It then swaps the values of `thisBatch` and `nextBatch` to release the next batch of readers, and it notifies all waiting threads, which wakes up the readers.

The Take-a-Number Monitor

As each reader or writer arrives, it takes the next number, and threads are served in order by their number. However, all readers with consecutive numbers are served at the same time.

State of the monitor. The state of this monitor is contained in three variables:

`nr`: The number of threads currently reading.

`nextNumber`: The number to be taken by the next thread to arrive.

`nowServing`: The number of the thread to be served next.

Example 4–27 The `TakeANumberMonitor` method `startReading()`

```
 public synchronized void startReading()
     throws InterruptedException{
       int myNumber = nextNumber++;
       while (nowServing!=myNumber) wait();
       nr++;
       nowServing++;
       notifyAll();
}
```

`startReading()`. If a thread tries to start reading, it takes a number and waits until its number comes up. Then, it increments the number of readers and the number currently being served. By incrementing `nowServing`, a reader allows the reader with the next sequential number to start. If the thread with the next number is a writer, it will wait until the readers are done before starting.

Example 4–28 The `TakeANumberMonitor` method `startWriting()`

```
public synchronized void startWriting()
throws InterruptedException{
       int myNumber = nextNumber++;
       while (nowServing!=myNumber) wait();
       while(nr>0) wait();
}
```

`startWriting().` To start writing, a thread takes a number, waits until its number comes up, and then waits until any preceding readers are finished. It will not increment `nowServing` until it is finished writing, since no subsequent thread should run until this writer is done.

Example 4–29 The `TakeANumberMonitor` method `stopReading()`

```
        public synchronized void stopReading(){
               nr--;
               if (nr==0) notifyAll();
        }
```

`stopReading().` When the last present reader finishes reading, it wakes up a waiting writer (if there is one present). Unfortunately, it has to wake up all threads just for one of them to start writing.

Example 4–30 The `TakeANumberMonitor` method `stopWriting()`

```
        public synchronized void stopWriting(){
               nowServing++;
               notifyAll();
        }
```

`stopWriting().` When a writer finishes writing, it increments `nowServing` and wakes up all waiting readers and writers to let them see if they are next.

Chapter Wrap-up

We discussed what monitors are supposed to be and observed that Java's synchronized methods provide degenerate forms of monitors. We then studied the thread package's `Monitor` class, which provides more of the facilities of monitors, but without any of the syntactic support. The implementation of `Monitor` and related classes shows sophisticated uses of semaphores.

We then considered multiple reader–writer monitors, which provide shared access to resources for readers, but mutual exclusion for writers. The different implementations provide different scheduling policies. The readers-preferred monitor gives readers priority over writers. The writers-preferred monitors do exactly the opposite. The alternating readers–writers monitor alternates between giving access to a single writer and a batch of readers. The take-a-number monitor gives access to threads in first-come, first-served order while alternating between single writers and batches of readers. A tricky problem in implementing these monitors is properly allocating a monitor to a newly arriving thread when no thread is currently using it and no threads are queued for it when the new thread arrives.

Exercises

1. The language Python provides lock objects similar to binary semaphores (i.e., semaphores for which the count is either zero or one). Here are the facilities of a Java version of Python locks:

The constructor

```
Lock L=new Lock();
```

creates a `Lock` object, which is initially unlocked. The method

```
L.acquire();
```

which locks the `Lock` object (similarly to `down()` for a semaphore). It blocks the thread's execution until the `Lock` object is available. The call

```
boolean a=L.acquire(false);
```

will immediately return, saying whether it acquired the lock. It assigns `true` to `a` if the lock was acquired. If the lock wasn't acquired, it assigns `false`. The call `L.acquire(true)` is equivalent to `L.acquire()`. The method

```
L.release();
```

releases a lock, allowing another thread to acquire it. A blocked thread waiting for the lock is given the chance to acquire it.

If a thread tries to lock the same `Lock` object more than once, it is deadlocked with itself.

Implement `Lock` objects in Java that provide the facilities of Python locks.

2. Python provides reentrant locks similar to the monitors in Java that allow the same thread to synchronize more than once on the same object. For that matter, they are like the monitors described in this chapter, which allow a thread to enter the same monitor more than once before leaving it. The Java version is as follows:

The constructor call

```
RLock L=new RLock();
```

creates a reentrant lock. The operations on an `RLock` object are the same as on a `Lock` object, except that a thread will not block itself if it tries to acquire the same lock more than once before releasing it. The `RLock` object is not released for other threads until the thread that owns it has released it as many times as it has acquired it.

Implement `RLock` objects in Java that provide the facilities of Python `RLocks`.

3. The language Python provides `Condition` objects similar to the `Monitor.Condition` objects in the thread package. The following is a Java syntax for them:

The constructor call

```
Condition C=new Condition(L);
```

creates a `Condition` object that uses underlying `Lock` or `RLock` object `L`. If `L` is omitted, as in

```
Condition C=new Condition();
```

a new `RLock` object is created for the `Condition`. The call

```
C.acquire();
```

locks the underlying `Lock` object. It blocks until the lock object is available. Call

```
boolean a=C.acquire(false);
```

will return immediately, saying whether it acquired the lock or not. It assigns `true` to `a` if the lock was acquired. If the lock wasn't acquired, it assigns `false`. The call `C.acquire(true)` is equivalent to `C.acquire(),` and

```
C.release();
```

releases the underlying lock. The method

```
C.wait();
```

must only be called by the thread that has acquired the lock. It releases the lock and waits until some other thread calls `C.notify()` or `C.notifyAll()`. The method

```
C.wait(timeout);
```

will wait until awakened by `notify()` or `notifyAll()` or until `timeout` seconds have elapsed. The parameter `timeout` is a floating-point number, so fractions of a second may be specified. Calling

```
C.notify();
```

will awaken at most one thread waiting on a `Condition`. If no threads are waiting, `notify()` performs no operation. Method `notify()` may only be called by the thread that has acquired the condition's lock. As in Java, a thread being awakened must reacquire the lock before returning from `wait()`. The thread executing the `notify()` does not simultaneously release the lock, so the awakened thread will be delayed. Other threads may acquire the lock before the waiting thread does. The method

```
C.notifyAll();
```

will awaken all waiting threads. The same caveats apply to it as for `notify()`.

Implement a `Condition` class whose objects behave in the fashion just described.