4. Monitors

Problems with semaphores:

- shared variables and the semaphores that protect them are global variables
- Operations on shared variables and semaphores distributed throughout program
- difficult to determine how a semaphore is being used (mutual exclusion or condition synchronization) without examining all of the code.

The monitor concept was developed by Tony Hoare and Per Brinch Hansen in the early '70's to overcome these problems. (Same time period in which the concept of information hiding [Parnas 1972] and the class construct [Dahl et al. 1970] originated.)

Monitors support data encapsulation and information hiding and are easily adapted to an object-oriented environment.

4.1 Definition of Monitors

A monitor encapsulates shared data, all the operations on the data, and any synchronization required for accessing the data.

Object-oriented definition: a monitor is a synchronization object that is an instance of a special *monitor* class.

- A monitor class defines private variables and public and private access methods.
- The variables of a monitor represent shared data.
- Threads communicate by calling monitor methods that access shared variables.

4.1.1 Mutual Exclusion

At most one thread is allowed to execute inside a monitor at any time.

- Mutual exclusion is automatically provided by the monitor's implementation.
- If a thread calls a monitor method, but another thread is already executing inside the monitor, the calling thread must wait outside the monitor.
- A monitor has an entry queue to hold the calling threads that are waiting to enter the monitor.

4.1.2 Condition Variables and SC Signaling

Condition synchronization is achieved using condition variables and operations *wait()* and *signal()*.

A condition variable *cv* is declared as

```
conditionVariable cv;
```

- Operation *cv.wait()* is used to block a thread (analogous to a *P* operation).
- Operation *cv.signal()* unblocks a thread (analogous to a *V* operation).

A monitor has one entry queue plus one queue associated with each condition variable. For example, Listing 4.1 shows the structure of monitor class *boundedBuffer*. Class *boundedBuffer* inherits from class *monitor*. It has five data members, condition variables named *notFull* and *notEmpty*, and monitor methods *deposit()* and *withdraw()*.

Fig. 4.2 is a graphical view of class *boundedBuffer*, which shows its entry queue and the queues associated with condition variables *notFull* and *notEmpty*.

```
class boundedBuffer extends monitor {
   public void deposit(…) { … }
   public int withdraw (…) { … }
   public boundedBuffer( ) { … }

   private int fullSlots = 0;  // # of full slots in the buffer
   private int capacity = 0;  // capacity of the buffer
   private int [] buffer = null;  // circular buffer of ints
   // in is index for next deposit, out is index for next withdrawal
   private int in = 0, out = 0;
   // producer waits on notFull when the buffer is full
   private conditionVariable notFull;
   // consumer waits on notEmpty when the buffer is empty
   private conditionVariable notEmpty;
}
```

Listing 4.1 Monitor class *boundedBuffer.*

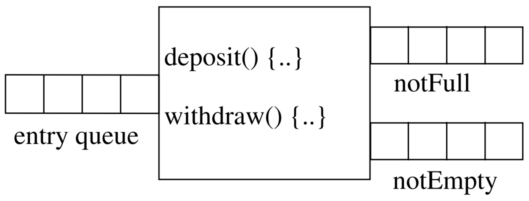


Figure 4.2 Graphical view of monitor class *boundedBuffer*.

A thread that is executing inside a monitor method blocks itself on condition variable *cv* by executing cv.wait():

- releases mutual exclusion (to allow another thread to enter the monitor)
- blocks the thread on the rear of the queue for *cv*.

A thread blocked on condition variable *cv* is awakened by cv.signal();

- If there are no threads blocked on *cv*, *signal()* has no effect; otherwise, *signal()* awakens the thread at the front of the queue for *cv*.
- For now, we will assume that the "signal-and-continue" (SC) discipline is used. After a thread executes an SC signal to awaken a waiting thread, the signaling thread continues executing in the monitor and the awakened thread is moved to the entry queue; *the awakened thread does not reenter the monitor immediately*.

A: denotes the set of threads that have been awakened by *signal()* operations and are waiting to reenter the monitor,

S: denotes the set of signaling threads,

C: denotes the set of threads that have called a monitor method but have not yet entered the monitor. (The threads in sets A and C wait in the entry queue.)

=> The relative priority associated with these three sets of threads is S > C = A.

cv.signalAll() wakes up all the threads that are blocked on condition variable *cv*.

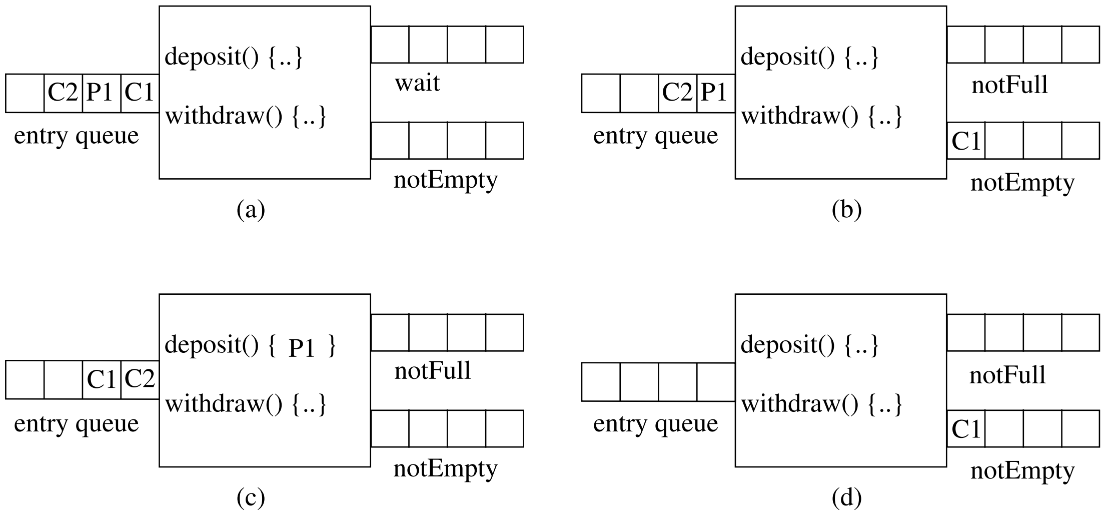
cv.empty() returns *true* if the queue for *cv* is empty, and *false* otherwise.

cv.length() returns the current length of the queue for *cv*.

Listing 4.3 shows a complete *boundedBuffer* monitor.

```
class boundedBuffer extends monitor {
   private  int fullSlots = 0; // number of full slots in the buffer
   private int capacity = 0;   // capacity of the buffer
   private int[] buffer = null;    // circular buffer of ints
   private int in = 0, out = 0;
   private  conditionVariable notFull = new conditionVariable();
   private  conditionVariable notEmpty = new conditionVariable();
   public boundedBuffer(int bufferCapacity ) {
      capacity = bufferCapacity;buffer = new int[bufferCapacity];}
      public void deposit(int value) {
      while (fullSlots == capacity)
         notFull.wait();
      buffer[in] = value; in = (in + 1) % capacity;     ++fullSlots;
      notEmpty.signal();     //alternatively:if (fullSlots == 1) notEmpty.signal();
   }
   public int withdraw() {
      int value;
      while (fullSlots == 0)
         notEmpty.wait();
      value = buffer[out]; out = (out + 1) % capacity; --fullSlots;
      notFull.signal();      //alternatively:if (fullSlots == capacity–1) notFull.signal();
      return value;
   }
}
```

Listing 4.3 Monitor class *boundedBuffer*.



Assume that the buffer is empty and that the thread at the front of the entry queue is Consumer$_1$ (C$_1$). The queues for condition variables *notFull* and *notEmpty* are also assumed to be empty (Fig. 4.4a).

When Consumer$_1$ enters method *withdraw()*, it executes the statement

```
while (fullSlots == 0)
    notEmpty.wait();
```

Since the buffer is empty, Consumer$_1$ blocks itself by executing a *wait()* operation on condition variable *notEmpty* (Fig. 4.4b)

Producer$_1$ (P$_1$) then enters the monitor. Since the buffer is not full, Producer$_1$ deposits an item and executes *notEmpty.signal()*.

This signal operation awakens Consumer$_1$ and moves Consumer$_1$ to the rear of the entry queue behind Consumer$_2$ (C$_2$) (Fig. 4.4c).

After its *signal()* operation, Producer$_1$ can continue executing in the monitor, but since there are no more statements to execute, Producer$_1$ exits the monitor.

Consumer_2 now barges ahead of Consumer_1 and consumes an item. Consumer_2 executes *notFull.signal()*, but there are no Producers waiting so the signal has no effect.

When Consumer_2 exits the monitor, Consumer_1 is allowed to reenter, but the loop condition (*fullSlots* == *0*) is true again:

```
while (fullSlots == 0)
    notEmpty.wait();
```

Thus, Consumer_1 is blocked once more on condition variable *notEmpty* (Fig4.4d). Even though Consumer_1 entered the monitor first, it is Consumer_2 that consumes the first item.

This example illustrates why the *wait()* operations in an SC monitor are usually found inside while-loops: A thread waiting on a condition variable cannot assume that the condition it is waiting for will be true when it reenters the monitor.

4.2 Monitor-Based Solutions to Concurrent Programming Problems

These solutions assume that condition variable queues are First-Come-First-Serve.

4.2.1 Simulating Counting Semaphores

4.2.1.1 Solution 1. Listing 4.5 shows an SC monitor with methods *P()* and *V()* that simulates a counting semaphore. In this implementation, a waiting thread may get stuck forever in the while-loop in method *P()*:

- assume that the value of *permits* is 0 when thread T1 calls *P()*. Since the loop condition (*permits == 0*) is true, T1 will block itself by executing a *wait* operation.
- assume some other thread executes *V()* and signals T1. Thread T1 will join the entry queue behind threads that have called *P*() and are waiting to enter for the first time.
- These other threads can enter the monitor and decrement *permits* before T1 has a chance to reenter the monitor and examine its loop condition. If the value of *permits* is 0 when T1 eventually evaluates its loop condition, T1 will block itself again by issuing another *wait* operation.

```
class countingSemaphore1 extends monitor {
   private int permits; // The value of permits is never negative.
   private conditionVariable permitAvailable = new conditionVariable();
   public countingSemaphore1(int initialPermits) { permits = initialPermits;}
   public void P()  {
      while (permits == 0)
         permitAvailable.wait();
      --permits;
   }
   public void V() {
      ++permits;
      permitAvailable.signal();
   }
}
```

Listing 4.5 Class *countingSemaphore1*.

4.2.1.2 Solution 2. The SC monitor in Listing 4.6 does not suffer from a starvation problem. Threads that call *P*() cannot barge ahead of signaled threads and “steal” their permits.

Consider the scenario that we described in Solution 1:

- If thread T1 calls *P()* when the value of *permits* is 0, T1 will decrement *permits* to –1 and block itself by executing a *wait* operation.
- When some other thread executes *V()*, it will increment *permits* to 0 and signal T1. Threads ahead of T1 in the entry queue can enter the monitor and decrement *permits* before T1 is allowed to reenter the monitor.
- However, these threads will block themselves on the *wait* operation in P(), since *permits* will have a negative value.
- Thread T1 will eventually be allowed to reenter the monitor. Since there are no statements after the *wait* operation, T1 will complete its *P()* operation.

⇒ In this solution, a waiting thread that is signaled is guaranteed to get a permit.

```
class countingSemaphore2 extends monitor {
   private int permits;  // The value of permits may be negative.
   private conditionVariable permitAvailable = new conditionVariable();
   public countingSemaphore2(int initialPermits) { permits = initialPermits;}
   public void P()  {
      --permits;
      if (permits < 0)
         permitAvailable.wait();
   }
   public void V() {
      ++permits;
      permitAvailable.signal();
   }
}
```

Listing 4.6 Class *countingSemaphore2*.

4.2.2 Simulating Binary Semaphores

In the SC monitor in Listing 4.7, Threads in *P()* wait on condition variable *allowP* while threads in *V()* wait on condition variable *allowV*. Waiting threads may get stuck forever in the while-loops in methods *P()* and *V()*.

```
class binarySemaphore extends monitor {
   private int permits;
   private conditionVariable allowP = new conditionVariable();
   private conditionVariable allowV = new conditionVariable();
   public binarySemaphore(int initialPermits) { permits = initialPermits;}
   public void P() {
      while (permits == 0)
         allowP.wait();
      permits = 0;
      allowV.signal();
   }
   public void V() {
      while (permits == 1)
         allowV.wait();
      permits = 1;
      allowP.signal();
   }
}
```

Listing 4.7 Class *binarySemaphore*.

4.2.3 Dining Philosophers

4.2.3.1 Solution 1. In the SC monitor in Listing 4.8, a philosopher picks up two chopsticks only if both of them are available. Each philosopher has three possible states: thinking, hungry and eating:

- A hungry philosopher can eat if her two neighbors are not eating.
- A philosopher blocks herself on a condition variable if she is hungry but unable to eat.
- After eating, a philosopher will unblock a hungry neighbor who is able to eat.

This solution is deadlock-free, but not starvation-free, since a philosopher can starve if one of its neighbors is always eating

However, the chance of a philosopher starving may be so highly unlikely that perhaps it can be safely ignored?

4.2.3.2 Solution 2. In Listing 4.9, each philosopher has an additional state called "starving":

- A hungry philosopher is not allowed to eat if she has a starving neighbor, even if both chopsticks are available.
- Two neighboring philosophers are not allowed to be starving at the same time.

$\Rightarrow$ a hungry philosopher enters the "starving" state if she cannot eat and her two neighbors are not starving.

This solution avoids starvation. If there are five philosophers, then no more than four philosophers can eat before a given hungry philosopher is allowed to eat. However, some philosophers may not be allowed to eat even when both chopsticks are available.

Compared to Solution 1, this solution limits the maximum time that a philosopher can be hungry, but it can also increase the average time that philosophers are hungry.

```
class diningPhilosopher1 extends monitor {
   final int n = …;       // number of philosophers
   final int thinking = 0; final int hungry = 1; final int eating = 2;
   int state[] = new int[n];   // state[i] indicates the state of philosopher i
   // philosopher i blocks herself on self[i] when she is hungry but unable to eat
   conditionVariable[] self  = new conditionVariable[n];
   diningPhilosopher1() {
   for (int i = 0; i < n; i++) state[i] = thinking;
   for (int j = 0; j < n; j++) self[j] = new conditionVariable( );
   }
   public void pickUp(int i) {
      state[i] = hungry;
      test(i);        // change state to eating if philosopher i is able to eat
      if (state[i] != eating)
      self[i].wait();
   }
   public void putDown(int i) {
      state[i] = thinking;
      test((i-1) % n);     // check the left neighbor
      test((i+1) % n);    // check the right neighbor
   }
   private void test(int k) {
   // if philosopher k is hungry and can eat, change her state and signal her queue.
      if (( state[k] == hungry) && (state[(k+n-1) %  n] != eating ) &&
            (state[(k+1) %  n] != eating )) {
         state[k] = eating;
         self[k].signal(); // no affect if philosopher k is not waiting on self[k]
      }
   }
}

Philosopher i executes:
while (true) {
   /* thinking  */
   dp1.pickUp(i);
   /* eating */
   dp1.putDown(i)
}
```

Listing 4.8 Class *diningPhilosopher1*.

```
class diningPhilosopher2 extends monitor {
   final int n = …;        // number of philosophers
   final int thinking = 0; final int hungry = 1;
   final int starving = 2; final int eating = 3;
   int state[] = new int[n];    // state[i] indicates the state of philosopher i
   // philosopher i blocks herself on self[i] when she is hungry, but unable to eat
   conditionVariable[] self  = new conditionVariable[n];
   diningPhilosopher2() {
      for (int i = 0; i < n; i++) state[i] = thinking;
      for (int j = 0; j < n; j++) self[j] = new conditionVariable( );
   }
   public void pickUp(int i) {
      state[i] = hungry;
      test(i);
      if (state[i] != eating)
          self[i].wait();
   }
   public void  putDown(int i) {
      state[i] = thinking;
      test((i-1) % n);
      test((i+1) % n);
   }
   private void test(int k) {
   // Determine whether the state of philosopher k should be changed to
   // eating or starving. A hungry philosopher is not allowed to eat if she has a
   // neighbor that's starving or eating.
      if (( state[k] == hungry || state[k] == starving ) &&
          (state[(k+n-1) % n] != eating  && state[(k+n-1) % n] != starving ) &&
          (state[(k+1) % n] != eating  && state[(k+1) % n] !=starving)) {
              state[k] = eating;
              self[k].signal();    // no effect if phil. k is not waiting on self[k],
      }                    // which is the case if test() was called from pickUp().
      // a hungry philosopher enters the "starving" state if she cannot eat and her
      // neighbors are not starving
      else if   ((state[k] == hungry) && (state[(k+n-1) % n] != starving ) &&
                (state[(k+1) % n] != starving )) {
          state[k] = starving;
      }
   }
}
```

Listing 4.9 Class *diningPhilosopher2*.

4.2.4 Readers and Writers

Listing 4.10 is an SC monitor implementation of strategy R>W.1, which allows concurrent reading and gives readers a higher priority than writers (see Section 3.5.4.) Reader and writer threads have the following form:

```
r_gt_w.1 rw;

Reader Threads:             Writer Threads:
   rw.startRead();             rw.startWrite();
   /* read shared data */      /* write to shared data */
   rw.endRead();               rw.endWrite();
```

Writers are forced to wait in method *startWrite()* if any writers are writing or any readers are reading or waiting.

In method *endWrite()*, all the waiting readers are signaled since readers have priority.

However, one or more writers may enter method *startWrite()* before the signaled readers reenter the monitor. Variable *signaledReaders* is used to prevent these barging writers from writing when the signaled readers are waiting in the entry queue and no more readers are waiting in *readerQ*.

Notice above that the shared data is read outside the monitor. This is necessary in order to allow concurrent reading.

```
class r_gt_w_1 extends monitor {
   int readerCount = 0;    // number of active readers
   boolean writing = false;  // true if a writer is writing
   conditionVariable readerQ = new conditionVariable();
   conditionVariable writerQ = new conditionVariable();
   int signaledReaders = 0;  // number of readers signaled in endWrite
   public void startRead() {
      if (writing) {            // readers must wait if a writer is writing
         readerQ.wait();
         --signaledReaders;// another signaled reader has started reading
      }
      ++readerCount;
   }
   public void endRead() {
      --readerCount;
      if (readerCount == 0 && signaledReaders==0)
         // signal writer if no more readers are reading and the signaledReaders
         //    have  read
         writerQ.signal();
   }
   public void startWrite() {
   // the writer waits if another writer is writing, or a reader is reading or waiting,
   // or the writer is barging
      while (readerCount > 0 || writing || !readerQ.empty() || signaledReaders>0)
          writerQ.wait();
      writing = true;
   }
   public void endWrite() {
      writing = false;
      if (!readerQ.empty()) { // priority is given to waiting readers
          signaledReaders = readerQ.length();
          readerQ.signalAll();
      }
      else writerQ.signal();
   }
}
```

Listing 4.10 Class *r_gt_w_1* allows concurrent reading and gives readers a higher priority than writers.

4.3 Monitors in Java

Java's wait, notify, and notifyAll operations combined with synchronized methods and user-defined classes enables the construction of objects that have some of the characteristics of monitors.

Adding synchronized to the methods of a Java class automatically provides mutual exclusion for threads accessing the data members of an instance of this class.

However, if some or all of the methods are inadvertently not synchronized, a data race may result. This enables the very types of bugs that monitors were designed to eliminate!

There are no explicit condition variables in Java. When a thread executes a wait operation, it can be viewed as waiting on a single, implicit condition variable associated with the object.

Operations wait, notify, and notifyAll use SC signaling:

- A thread must hold an object's lock before it can execute a wait, notify, or notifyAll operation. Thus, these operations must appear in a synchronized method or synchronized block (see below); otherwise, an *IllegalMonitorStateException* is thrown.

- Every Java object has a lock associated with it. Methods wait, notify, and notifyAll are inherited from class *Object*, the base class for all Java objects.

- When a thread executes wait, it releases the object's lock and waits in the "wait set" that is associated with the object:
  - A notify operation awakens a single waiting thread in the wait set.
  - A notifyAll operation awakens all the waiting threads.
  - Operations notify and notifyAll are not guaranteed to wake up the thread that has been waiting the longest.

- A waiting thread T may be removed from the wait set due to any one of the following actions: a notify or notifyAll operation; an interrupt action being performed on T; a timeout for a timed wait (e.g., wait(1000) allows T to stop waiting after one second); or a "spurious wakeup", which removes T without any explicit Java instructions to do so (Huh?!).

- A notified thread must reacquire the object's lock before it can begin executing in the method. Furthermore, notified threads that are trying to reacquire an object's lock compete with any threads that have called a method of the object and are trying to acquire the lock for the first time. The order in which these notified and calling threads obtain the lock is unpredictable.

- If a waiting thread T is interrupted at the same time that a notification occurs, then the result depends on which version of Java is being used:

In versions before J2SE 5.0 [JSR-133 2004], the notification may be "lost".

- Suppose that thread T and several other threads are in the wait set and thread T is notified and then interrupted before it reacquires the monitor lock. Then the wait operation that was executed by T throws *InterruptedException* and the notification gets lost, i.e., no waiting thread is allowed to proceed.
- Thus, it is recommended that the catch block for *InterruptedException* execute an additional notify or notifyAll to make up for any lost notifications [Hartley 1999]. Alternately, the programmer should use notifyAll instead of notify to wake up all waiting threads even when just one thread can logically proceed.

J2SE 5.0 removes the possibility of lost notifications. If the interrupt of T occurs before T is notified then T's interrupt status is set to false, wait throws *InterruptedException*, and some other waiting thread (if any exist at the time of the notification) receives the notification. If the notification occurs first, then T eventually returns normally from the wait with its interrupt status set to true.

(A thread can determine its interrupt status by invoking the static method *Thread.isInterrupted()*, and it can observe and clear its interrupt status by invoking the static method *Thread.interrupted()*.)

4.3.1 A Better *countingSemaphore*

Class *countingSemaphore* in Listing 4.11 has been revised to handle interrupts and spurious wakeups. It assumes that J2SE 5.0 interrupt semantics are used, which prevents notifications from being lost when a waiting thread is interrupted right before it is notified.

```
public final class countingSemaphore {
   private int permits = 0; int waitCount = 0; int notifyCount=0;
   public countingSemaphore(int initialPermits) {
      if (initialPermits>0) permits = initialPermits;
   synchronized public void P() throws InterruptedException {
     if (permits <= waitCount) {
        waitCount++;           // one more thread is waiting
        try {
           do { wait(); }      // spurious wakeups do not increment
           while (notifyCount == 0);//   notifyCount
        }
        finally { waitCount--; }   // one waiting thread notified or interrupted
        notifyCount--;    // one notification has been consumed
      }
     else {
        if (notifyCount > waitCount)   // if some notified threads were
           notifyCount--;      //      interrupted, adjust notifyCount
     }
     permits--;
   }
   synchronized public void V() {
     permits++;
     if (waitCount > notifyCount) { // if there are waiting threads yet to be
        notifyCount++;     //    notified, notify one thread
        notify();
     }
   }
}
```

Listing 4.11 Java Class *countingSemaphore* that handles interrupts and spurious wakeups.

Variable *notifyCount* counts notifications that are made in method V():

- The if-statement in method *V()* ensures that only as many notifications are done as there are waiting threads.
- A thread that awakens from a wait operation in *P()* executes wait again if *notifyCount* is zero since a spurious wakeup must have occurred (i.e., a notification must have been issued outside of *V()*.)

Assume that two threads are blocked on the wait operation in method *P()* and the value of *permits* is 0.

- Suppose that three *V()* operations are performed and all three *V()* operations are completed before either of the two notified threads can reacquire the lock.
- Then the value of *permits* is 3 and the value of *waitCount* is still 2.
- A thread that then calls *P()* and barges ahead of the notified threads is not required to wait and does not execute wait since the condition (*permits* <= *waitCount*) in the if-statement in method *P()* is false.

Assume that two threads are blocked on the wait operation in method *P()*

- Suppose two *V()* operations are executed: *notifyCount* and *permits* become 2.
- Suppose further that the two notified threads are interrupted so that *waitCount* becomes 0 (due to the interrupted threads decrementing *waitCount* in their finally blocks).
- If three threads now call method *P()*, two of the threads will be allowed to complete their *P()* operations, and before they complete *P()*, they will each decrement *notifyCount* since both will find that the condition *(notifyCount* > *waitCount*) is true.
- Now *permits*, *notifyCount* and *waitCount* are all 0.
- If another thread calls *P()* it will be blocked by the wait operation in *P()*, and if it is awakened by a spurious wakeup, it will execute wait again since *notifyCount* is zero.

Class *Semaphore* in J2SE 5.0 package *java.util.concurrent* provides methods *acquire()* and *release()* instead of *P()* and *V()*, respectively. The implementation *of acquire()* handles interrupts as follows.

- If a thread calling *acquire()* has its interrupted status set on entry to *acquire()*, or is interrupted while waiting for a permit, then InterruptedException is thrown and the calling thread's interrupted status is cleared.
- Any permits that were to be assigned to this thread are instead assigned to other threads trying to acquire permits.

Class *Semaphore* also provides method *acquireUninterruptibly()*:

- If a thread calling *acquireUninterruptibly()* is interrupted while waiting for a permit then it will continue to wait until it receives a permit.
- When the thread does return from this method its interrupt status will be set.

4.3.2 notify vs. notifyAll

A Java monitor only has a single (implicit) condition variable available to it so notify operations must be handled carefully.

Listing 4.12 shows Java class *binarySemaphore*.

- Threads that are blocked in *P*() or *V*() are waiting in the single queue associated with the implicit condition variable.
- Any execution of notifyAll awakens all the waiting threads, even though one whole group of threads, either those waiting in *P*() or those waiting in *V*(), cannot possibly continue.
- The first thread, if any, to find that its loop condition is false, exits the loop and completes its operation. The other threads may all end up blocking again.

If notify were used instead of notifyAll, the single thread that was awakened might be a member of the wrong group. If so, the notified thread would execute another wait operation and the notify operation would be lost, potentially causing deadlock.

Use notifyAll instead of notify unless the following requirements are met:

1. all waiting threads are waiting on conditions that are signaled by the same notifications. If one condition is signaled by a notification, then the other conditions are also signaled by this notification. Usually, when this requirement is met, all the waiting threads are waiting on the exact same condition.
2. each notification is intended to enable exactly one thread to continue. (In this case, it would be useless to wake up more than one thread.)

Example: In class *countingSemaphore* in Listing 3.15, all threads waiting in *P*() are waiting for the same condition ($permits \geq 0$), which is signaled by the notify operation in *V*(). Also, a notify operation enables one waiting thread to continue.

Even though both of these requirements might be satisfied in a given class, they may not be satisfied in subclasses of this class, so using notifyAll may be safer. (Use the `final` keyword to prevent subclassing.)

Monitor *boundedBuffer* in Listing 4.13 uses notifyAll operations since *Producers* and *Consumers* wait on the same implicit condition variable and they wait for different conditions that are signaled by different notifications.

```
final class boundedBuffer {
   private int fullSlots=0; private int capacity = 0;
   private int[] buffer = null; private int in = 0, out = 0;
   public boundedBuffer(int bufferCapacity) {
      capacity = bufferCapacity; buffer = new int[capacity];
   }
   public synchronized void deposit (int value) {
      while (fullSlots == capacity) // assume no interrupts are possible
         try { wait(); } catch (InterruptedException ex) {}
      buffer[in] = value; in = (in + 1) % capacity;
      if (fullSlots++ == 0) // note the use of post-increment.
         notifyAll();   // it is possible that Consumers are waiting for “not empty”
   }
   public synchronized int withdraw () {
      int value = 0;
      while (fullSlots == 0)   // assume no interrupts are possible
         try { wait(); } catch (InterruptedException ex) {}
      value = buffer[out];  out = (out + 1) % capacity;
      if (fullSlots-- == capacity)    // note the use of post-decrement.
         notifyAll(); // it is possible that Producers are waiting for “not full”
      return value;
   }
}
```

Listing 4.13 Java monitor *boundedBuffer*.

4.3.3 Simulating Multiple Condition Variables

It is possible to use simple Java objects to achieve an effect that is similar to the use of multiple condition variables.

Listing 4.14 shows a new version of class *binarySemaphore* that uses objects *allowP* and *allowV* in the same way that condition variables are used.

Notice that methods *P*() and *V*() are not synchronized since it is objects *allowP* and *allowV* that must be synchronized in order to perform wait and notify operations on them. (Adding synchronized to methods *P()* and *V()* would synchronize the *binarySemaphore2* object, not objects *allowP* and *allowV*, and would result in a deadlock.)

The use of a synchronized block:

```
synchronized (allowP) {
   /* block of code */
}
```

creates a block of code that is synchronized on object *allowP*.

- A thread must acquire *allowP's* lock before it can enter the block.
- The lock is released when the thread exits the block.

Note that a synchronized method:

```
public synchronized void F() {
   /* body of F */
}
```

is equivalent to a method whose body consists of a single synchronized block:

```
public void F() {
   synchronized(this) {
      /* body of F */
   }
}
```

```
public final class binarySemaphore2 {
   int vPermits = 0;   int pPermits = 0;
   Object allowP = null;    // queue of threads waiting in P()
   Object allowV = null;   // queue of threads waiting in V()
   public binarySemaphore2(int initialPermits) {
      if (initialPermits != 0 && initialPermits != 1) throw new
       IllegalArgumentException("initial binary semaphore value must be 0 or 1");
      pPermits = initialPermits;  // 1 or 0
      vPermits = 1 – pPermits;   // 0 or 1
      allowP = new Object(); allowV = new Object();
   }
   public void P() {
      synchronized (allowP) {
          --pPermits;
          if (pPermits < 0) // assume no interrupts are possible
             try { allowP.wait(); } catch (InterruptedException e) {}
      }
      synchronized (allowV) {
          ++vPermits;
          if (vPermits <=0)
             allowV.notify(); // signal thread waiting in V()
      }
   }
    public void V() {
       synchronized (allowV) {
          --vPermits;
          if (vPermits < 0) // assume no interrupts are possible
             try { allowV.wait(); } catch (InterruptedException e) {}
       }
       synchronized (allowP) {
           ++pPermits;
           if (pPermits <= 0)
             allowP.notify(); // signal thread waiting in P()
       }
   }
}
```

Listing 4.14 Java class *binarySemaphore2*.

4.4 Monitors in Pthreads

Pthreads does not provide a monitor construct, but it provides condition variables, which enables the construction of monitor-like objects.

4.4.1 Pthreads Condition Variables

The operations that can be performed on a Pthreads condition variable are *pthread_cond_wait()*, *pthread_cond_signal()*, and *pthread_cond_broadcast()*.

The *signal* and *broadcast* operations are similar to Java's notify and notifyAll operations, respectively.

Also like Java, *wait* operations are expected to be executed inside a critical section. Thus, a condition variable is associated with a *mutex* and this *mutex* is specified when a *wait* operation is performed.

- A condition variable is initialized by calling *pthread_cond_init*().

- When a thread waits on a condition variable it must have the associated *mutex* locked. This means that a *wait* operation on a condition variable must be preceded by a lock operation on the *mutex* associated with the condition variable.

- Each condition variable must be associated at any given time with only one mutex. On the other hand, a mutex may have any number of condition variables associated with it.

- A *wait* operation automatically unlocks the associated *mutex* if the calling thread is blocked and automatically tries to lock the associated *mutex* when the blocked thread is awakened by a *signal* or *broadcast* operation. The awakened thread competes with other threads for the *mutex*.

- A *signal* operation wakes up a single thread while a *broadcast* operation wakes up all waiting threads.

- A thread can *signal* or *broadcast* a condition variable without holding the lock for the associated *mutex*. (This is different from Java since a Java *notify* or *notifyAll* operation must be performed in a synchronized method or block.) A thread that executes a *signal* or *broadcast* continues to execute. If the thread holds the lock for the associated *mutex* when it performs a *signal* or *broadcast*, it should eventually release the lock.

Listing 4.15 shows a C++/Pthreads monitor class named *boundedBuffer* that solves the bounded buffer problem.

Producers wait on condition variable *notFull* and Consumers wait on condition variable *notEmpty*. The use of two explicit condition variables makes this Pthreads version similar to the solution in Listing 4.3.

Note that the *wait* operations appear in loops since Pthreads uses the SC signaling discipline.

Also, it is recommended to always use loops because of the possibility of "spurious wakeups", i.e., threads can be awakened without any *signal* or *broadcast* operations being performed.

```
#include <iostream>
#include <pthread.h>
#include "thread.h"
class boundedBuffer  {
private:
    int fullSlots;                  // # of full slots in the buffer
    int capacity; int* buffer; int in, out;
    pthread_cond_t notFull;  // Producers wait on notFull
    pthread_cond_t notEmpty;  // Consumers wait on notEmpty
    pthread_mutex_t mutex;  // exclusion for deposit() and withdraw()
public:
    boundedBuffer(int capacity_) : capacity(capacity_), fullSlots(0), in(0),
       out(0), buffer(new int[capacity_]) {
          pthread_cond_init(&notFull,NULL); pthread_cond_init(&notEmpty,NULL);
          pthread_mutex_init(&mutex,NULL);
    }
    ~boundedBuffer() {
       delete [] buffer;
       pthread_cond_destroy(&notFull); pthread_cond_destroy(&notEmpty);
       pthread_mutex_destroy(&mutex);
    }
    void deposit(int value) {
       pthread_mutex_lock(&mutex);
       while (fullSlots == capacity)
           pthread_cond_wait(&notFull,&mutex);
       buffer[in] = value;
       in = (in + 1) % capacity;   ++fullSlots;
       pthread_mutex_unlock(&mutex);
       pthread_cond_signal(&notEmpty);
    }
    int withdraw() {
       pthread_mutex_lock(&mutex);
       int value;
       while (fullSlots == 0)
           pthread_cond_wait(&notEmpty,&mutex);
       value = buffer[out];  out = (out + 1) % capacity;  --fullSlots;
       pthread_mutex_unlock(&mutex);
       pthread_cond_signal(&notFull);
       return value;
    }
};
```

```
class Producer : public Thread {
private:
    boundedBuffer& b;
    int num;
public:
    Producer (boundedBuffer* b_, int num_) : b(*b_),  num(num_) { }
    virtual void* run () {
       std::cout << "Producer Running" << std::endl;
       for (int i = 0; i < 3; i++) {
           b.deposit(i);
           std::cout << "Producer # " << num << " deposited " << i << std::endl;
       }
       return NULL;
    }
};

class Consumer : public Thread {
private:
    boundedBuffer& b;
    int num;
public:
    Consumer (boundedBuffer* b_, int num_) : b(*b_), num(num_) { }
    virtual void* run () {
       std::cout << "Consumer Running" << std::endl;
       int value = 0;
       for (int i = 0; i < 3; i++) {
           value = b.withdraw();
           std::cout << "Consumer # " << num << " withdrew " <<
             value << std::endl;
       }
       return NULL;
    }
};

int main ( ) {
    boundedBuffer* b1 = new boundedBuffer(3);
    Producer p1(b1, 1); Consumer c1(b1, 1);
    p1.start(); c1.start();
    p1.join(); c1.join();
    delete b1;
    return 0;
}
```
Listing 4.15 C++/Pthreads class *boundedBuffer*.

4.4.2 Condition Variables in J2SE 5.0

Package *java.util.concurrent.locks* in Java release J2SE 5.0, contains a lock class called *ReentrantLock* (see Section 3.6.4) and a condition variable class called *Condition*.

A *ReentrantLock* replaces the use of a synchronized method, and operations *await* and *signal* on a *Condition* replace the use of methods *wait* and *notify*.

A *Condition* object is bound to its associated *ReentrantLock* object. Method *newCondition()* is used to obtain a *Condition* object from a *ReentrantLock*:

```
ReentrantLock mutex = new ReentrantLock();
// notFull and notEmpty are bound to mutex
Condition notFull = mutex.newCondition();
Condition notEmpty = mutex.newCondition();
```

Conditions provide operations *await*, *signal*, and *signallAll*, including those with timeouts.

Listing 4.16 shows a Java version of class *boundedBuffer* using *Condition* objects.

- The try-finally clause ensures that *mutex* is unlocked no matter how the try block is executed.
- If an interrupt occurs before a *signal* then the *await* method must, after re-acquiring the lock, throw InterruptedException. (If the interrupted thread is signaled, then some other thread (if any exist at the time of the *signal*) receives the signal.)
- if the interrupted occurs after a *signal*, then the *await* method must return without throwing an exception, but with the current thread's interrupt status set.

```
import java.util.concurrent.locks;
final class boundedBuffer {
    private int fullSlots=0; private int capacity = 0; private int in = 0, out = 0;
    private int[] buffer = null;
    private ReentrantLock mutex;
    private Condition notFull;
    private Condition notEmpty;

    public boundedBuffer(int bufferCapacity) {
       capacity = bufferCapacity; buffer = new int[capacity];
       mutex = new ReentrantLock();
       // notFull and notEmpty are both attached to mutex
       notFull = mutex.newCondition(); notEmpty = mutex.newCondition();
    }
    public void deposit (int value) throws InterruptedException {
       mutex.lock();
       try {
           while (fullSlots == capacity)
               notFull.await();
           buffer[in] = value;
           in = (in + 1) % capacity;
           notEmpty.signal();
       } finally {mutex.unlock();}
    }
    public synchronized int withdraw () throws InterruptedException {
       mutex.lock();
       try {
           int value = 0;
           while (fullSlots == 0)
               notEmpty.await();
           value = buffer[out];
           out = (out + 1) % capacity;
           notFull.signal();
           return value;
       } finally {mutex.unlock();}
    }
}
```

Listing 4.16 Java class *boundedBuffer* using *Condition* objects.

4.5 Signaling Disciplines

4.5.1 Signal-and-Urgent-Wait (SU)

When a thread executes *cv.signal()*:

- if there are no threads waiting on condition variable *cv*, this operation has no effect.
- otherwise, the thread executing signal (which is called the "signaler" thread) awakens one thread waiting on *cv*, and blocks itself in a queue, called the reentry queue. The signaled thread reenters the monitor immediately.

When a thread executes *cv.wait()*:

- if the reentry queue is not empty, the thread awakens one signaler thread from the reentry queue and then blocks itself on the queue for *cv*.
- otherwise, the thread releases mutual exclusion (to allow a new thread to enter the monitor) and then blocks itself on the queue for *cv*.

When a thread completes and exits a monitor method:

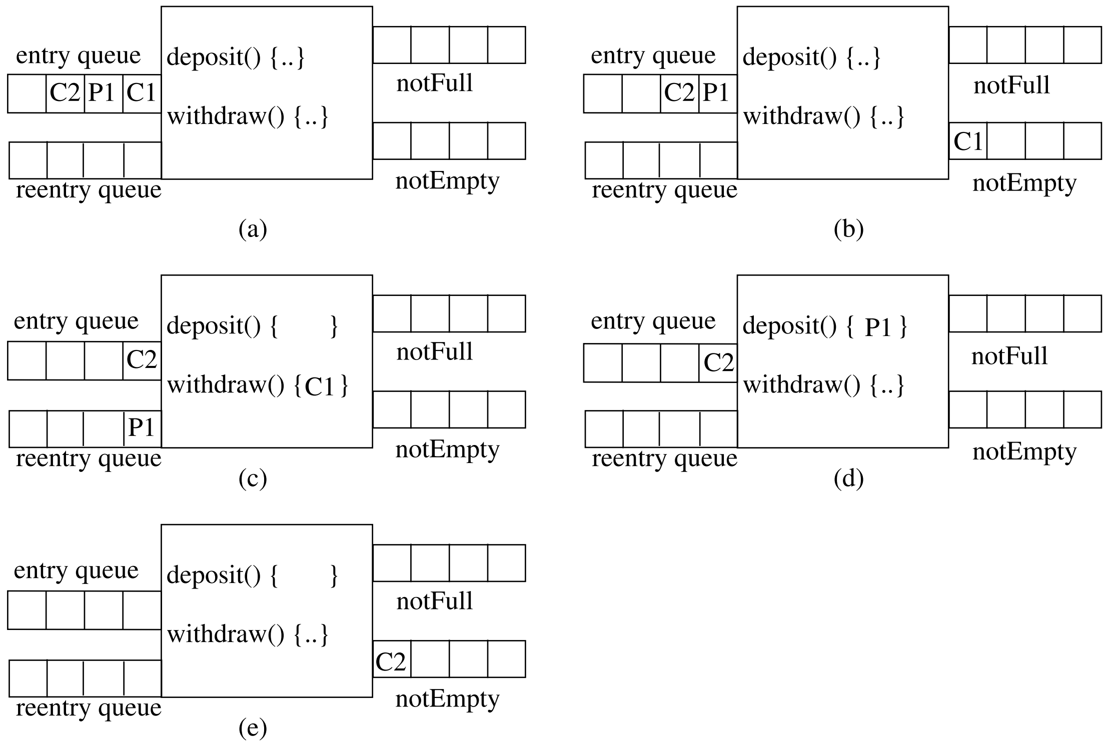
- if reentry queue is not empty, it awakens one signaler thread from the reentry queue.
- otherwise, it releases mutual exclusion to allow a new thread to enter the monitor.

In an SU monitor, the threads waiting to enter a monitor have three levels of priority (from highest to lowest):

- the awakened thread (A), which is the thread awakened by a signal operation
- signaler threads (S), which are the threads waiting in the reentry queue
- calling threads (C), which are the threads that have called a monitor method and are waiting in the entry queue.

The relative priority associated with the three sets of threads is A > S > C.

Considering again the *boundedBuffer* monitor in Listing 4.3. Assume that SU signals are used instead of SC signals. Assume we start with Fig. 4.17a.



When Consumer$_1$ enters method *withdraw()*, it executes the statement

```
while (fullSlots == 0)
    notEmpty.wait();
```

Since the buffer is empty, Consumer$_1$ is blocked by the *wait()* operation on condition variable *notEmpty* (Fig. 4.17b).

Producer$_1$ then enters the monitor:

- Since the buffer is not full, Producer$_1$ deposits its item, and executes *notEmpty.signal()*.
- This signal awakens Consumer$_1$ and moves Producer$_1$ to the Reentry queue (Fig. 4.17c).

Consumer$_1$ can now consume an item.

- When Consumer$_1$ executes *notFull.signal()*, there are no Producers waiting so none are signaled and Consumer$_1$ does not move to the Reentry queue.
- When Consumer$_1$ exits the monitor, Producer$_1$ is allowed to reenter the monitor since the Reentry queue has priority over the Entry queue (Fig. 4.17d).

Producer$_1$ has no more statements to execute so Producer$_1$ exits the monitor.

- Since the Reentry queue is empty, Consumer$_2$ is now allowed to enter the monitor.
- Consumer$_2$ finds that the buffer is empty and blocks itself on condition variable *notEmpty* (Fig. 4.17e).

Unlike the scenario that occurred when SC signals were used, Consumer$_2$ was not allowed to barge ahead of Consumer$_1$ and consume the first item.

Since signaled threads have priority over new threads, a thread waiting on a condition variable in an SU monitor can assume that the condition it is waiting for will be true when it reenters the monitor.

In the *boundedBuffer* monitor, we can replace the while-loops with if-statements and avoid the unnecessary reevaluation of the loop condition after a *wait()* operation returns.

As another example, Listing 4.18 shows an SU monitor implementation of strategy R>W.1. This implementation is simpler than the SC monitor in Section 4.2.4 since there is no threat of barging - when a writer signals waiting readers, these waiting readers are guaranteed to reenter the monitor before any new writers are allowed to enter *startWrite()*.

```
class r_gt_w_1SU extends monitorSU {
   int readerCount = 0;// number of active readers
   boolean writing = false;   // true if a writer is writing
   conditionVariable readerQ = new conditionVariable();
   conditionVariable writerQ = new conditionVariable();

   public void startRead() {
      if (writing)          // readers must wait if a writer is writing
         readerQ.wait();
      ++readerCount;
      readerQ.signal();  // continue cascaded wakeup of readers
   }
   public void endRead() {
      --readerCount;
      if (readerCount == 0)
         writerQ.signal();   // signal writer if there are no more readers reading
   }
   public void startWrite() {
   // writers wait if a writer is writing, or a reader is reading or waiting, or the
   // writer is barging
      while (readerCount > 0 || writing || !readerQ.empty())
          writerQ.wait();
      writing = true;
   }
   public void endWrite() {
      writing = false;
      if (!readerQ.empty()) { // priority is given to waiting readers
          readerQ.signal();  // start cascaded wakeup of readers
      }
      else
          writerQ.signal();
   }
}
```
Listing 4.18 Class *r_gt_w_1SU* allows concurrent reading and gives readers a higher priority than writers.

4.5.2 Signal-and-Exit (SE)

Signal-and-Exit (SE) is a special case of Signal-and-Urgent-Wait (SU).

When a thread executes an SE *signal* operation it does not enter the reentry queue; rather, it exits the monitor immediately.

- An SE *signal* statement is either the last statement of a method or it is followed immediately by a return statement.
- As with SU signals, the thread awakened by a signal operation is always the next thread to enter the monitor.

Since there are no signaling threads that want to remain in or reenter the monitor, the relative priority of the sets of awakened (A) and calling (C) threads is A > C.

When a *signal* statement appears in a monitor method, it is very often the last statement of the method, regardless of the type of *signal*. This was the case for all of the SC monitor examples in Section 4.2 and for the SU monitor in Listing 4.18.

Using SE semantics for these special SU *signal* operations avoids the extra cost of having a thread exit a monitor and join the reentry queue, and then later reenter the monitor only to immediately exit the monitor again.

The Java and C++ monitor classes shown later have a Signal-and-Exit operation that allows SE *signals* to be used at the end of SU monitor methods.

4.5.3 Urgent-Signal-and-Continue (USC)

- A thread that executes an USC *signal* operation continues to execute just as it would for an SC signal.
- A thread awakened by a *signal* operation has priority over threads waiting in the entry queue.

When *signal* operations appear only at the end of monitor methods, which is usually the case, this discipline is the same as the SE discipline, which is a special case of the SU discipline.

A thread waiting in the entry queue is allowed to enter a monitor only when no other threads are inside the monitor and there are no signaled threads waiting to reenter.

Thus, the relative priority associated with the three sets of threads is S > A > C.

Table 4.1 lists the signaling disciplines and shows the relative priorities associated with the three sets of threads. Another signaling discipline is described in Exercise 4.17.

Relative priority	Name
S > A = C	Signal-and-Continue [Lampson and Redell 1980]
A > S > C	Signal-and-Urgent-Wait [Hoare 1974]
A > C	Signal-and-Exit [Brinch Hansen 1975]
S > A > C	Urgent-Signal-and-Continue [Howard 1976]

Table 4.1 Signaling Disciplines

4.5.4 Comparing SU and SC signals

If a thread executes an SU *signal* to notify another thread that a certain condition is true, this condition remains true when the signaled thread reenters the monitor.

A *signal* operation in an SC monitor is only a “hint” to the signaled thread that it may be able to proceed. Other threads may “barge” into the monitor and make the condition false before the signaled thread reenters the monitor.

That is why SC monitors use a while-loop instead of an if-statement.

- Using a while-loop instead of an if-statement in an SC monitor requires an extra evaluation of condition (*permits == 0*) after a *wait()*.
- On the other hand, the execution of an SU monitor requires additional context switches for managing the signaler threads in the Reentry queue.

Using Signal-and-Exit semantics for a signal operation that appears at the end of an SU method avoids the costs of the extra condition evaluation and the extra context switches.

4.6 Using Semaphores to Implement Monitors

Semaphores can be used to implement monitors with SC, SU, or SE signaling.

Even though Java provides built-in support for monitors, our custom Java monitor classes are still helpful since it is not easy to test and debug built-in Java monitors. Also, we can choose which type of monitor - SC, SU, SE, or USC - to use in our Java programs.

4.6.1 SC Signaling.

The body of each public monitor method is implemented as

```
public returnType F(…) {
   mutex.P();
   /* body of F */
   mutex.V();
}
```

Semaphore *mutex* is initialized to 1. The calls to *mutex.P()* and *mutex.V()* ensure that monitor methods are executed with mutual exclusion.

Java class *conditionVariable* in Listing 4.20 implements condition variables with SC signals:

- since it is not legal to overload final method *wait*() in Java, methods named *waitC*() and *signalC*() are used instead of *wait()* and *signal()*.
- the *signalCall()* operation behaves the same as Java's notifyAll
- operations *empty()* and *length()* are also provided

```
final class conditionVariable {
   private countingSemaphore threadQueue = new countingSemaphore(0);
   private int numWaitingThreads = 0;
   public void waitC() {
      numWaitingThreads++;   // one more thread is waiting in the queue
      threadQueue.VP(mutex); // release exclusion and wait in threadQueue
      mutex.P();             // wait to reenter the monitor
   }
   public void signalC() {
      if (numWaitingThreads > 0) {  // if any threads are waiting
         numWaitingThreads--;//     // wakeup one thread in the queue
         threadQueue.V();
      }
   }
   public void signalCall() {
      while (numWaitingThreads > 0) {   // if any threads are waiting
         --numWaitingThreads;           //  wakeup all the threads in the queue
         threadQueue.V();               //    one-by-one
      }
   }
   // returns true if the queue is empty
   public boolean empty() { return (numWaitingThreads == 0); }
   // returns the length of the queue
   public int length() { return numWaitingThreads; }
}
```

Listing 4.20 Java class *conditionVariable*.

Each *conditionVariable* is implemented using a semaphore named *threadQueue*:

- When a thread executes *waitC()*, it releases mutual exclusion, and blocks itself using threadQueue.VP(mutex). VP() guarantees that threads executing *waitC()* are blocked on semaphore *threadQueue* in the same order that they entered the monitor.
- An integer variable named *numWaitingThreads* is used to count the waiting threads.

The value of *numWaitingThreads* is incremented in *waitC()* and decremented in *signalC()* and *signalCall()*:

- *signalC()* executes *threadQueue.V()* to signal one waiting thread.
- *signalCall()* uses a while-loop to signal all the waiting threads one-by-one.

4.6.2 SU Signaling.

Java class *conditionVariable* in Listing 4.21 implements condition variables with SU signals.

Each SU monitor has a semaphore named *reentry* (initialized to 0), on which signaling threads block themselves.

If *signalC()* is executed when threads are waiting on the condition variable, *reentry.VP(threadQueue)* is executed to signal a waiting thread, and block the signaler in the reentry queue.

Integer *reentryCount* is used to count the number of signaler threads waiting in *reentry*.

- When a thread executes *waitC*(), if signalers are waiting, the thread releases a signaler by executing *reentry.V()*; otherwise, the thread releases mutual exclusion by executing *mutex.V()*.
- Method *signalC()* increments and decrements *reentryCount* as threads enter and exit the *reentry* queue.

The body of each public monitor method is implemented as

```
public returnType F(…) {
  mutex.P();
  /* body of F */
  if (reentryCount >0)
     reentry.V();      // allow a signaler thread to reenter the monitor
  else mutex.V();      // allow a calling thread to enter the monitor
}
```

```
final class conditionVariable {
   private countingSemaphore threadQueue = new countingSemaphore(0);
   private int numWaitingThreads = 0;
   public void signalC() {
      if (numWaitingThreads > 0) {
         ++reentryCount;
         reentry.VP(threadQueue);  // release exclusion and join reentry queue
         --reentryCount;
      }
   }
   public void waitC() {
      numWaitingThreads++;
      if (reentryCount > 0) threadQueue.VP(reentry); // the reentry queue has
      else threadQueue.VP(mutex);      //   priority over entry queue
      --numWaitingThreads;
   }
   public boolean empty() { return (numWaitingThreads == 0); }
   public int length() { return numWaitingThreads; }
}
```

Listing 4.21 Java class *conditionVariable* for SU monitors.

4.7 A Monitor Toolbox for Java

A monitor toolbox is a program unit that is used to simulate the monitor construct. The Java monitor toolboxes are class *monitorSC* for SC monitors and class *monitorSU* for SU monitors.

Classes *monitorSC* and *monitorSU* implement operations *enterMonitor* and *exitMonitor*, and contain a member class named *conditionVariable* that implements *waitC* and *signalC* operations on condition variables.

A regular Java class can be made into a monitor class by doing the following:

1. extend class *monitorSC* or *monitorSU*
2. use operations *enterMonitor()* and *exitMonitor()* at the start and end of each public method
3. declare as many *conditionVariables* as needed
4. use operations *waitC()*, *signalC()*, *signalCall()*, *length()*, and *empty()*, on the *conditionVariables*.

Listing 4.22 shows part of a Java *boundedBuffer* class that illustrates the use of class *monitorSC*.

Simulated monitors are not as easy to use or as efficient as real monitors, but they have some advantages:

- A monitor toolbox can be used to simulate monitors in languages that do not support monitors directly, e.g., C++/Win32/Pthreads.
- Different versions of the toolbox can be created for different types of signals, e.g., an SU toolbox can be used to allow SU signaling in Java.
- The toolbox can be extended to support testing and debugging.

```
final class boundedBuffer extends monitorSC {
   …
   private conditionVariable notFull = new conditionVariable();
   private conditionVariable notEmpty = new conditionVariable();
   …
   public void deposit(int value) {
      enterMonitor();
      while (fullSlots == capacity)
         notFull.waitC();
      buffer[in] = value;
         in = (in + 1) % capacity;
      ++fullSlots;
      notEmpty.signalC();
      exitMonitor();
   }
   …
}
```

Listing 4.22 Using the Java monitor toolbox class *monitorSC*.

4.7.1 A Toolbox for SC Signaling in Java

Listing 4.23 shows a monitor toolbox that uses semaphores to simulate monitors with SC signaling.

Class *conditionVariable* is nested inside class *monitor*, which gives class *conditionVariable* access to member object *mutex* in the *monitorSC* class.

```
public class monitorSC { // monitor toolbox with SC signaling
    private binarySemaphore mutex = new binarySemaphore(1);
    protected final class conditionVariable {
       private countingSemaphore threadQueue = new countingSemaphore(0);
       private int numWaitingThreads = 0;
       public void signalC() {
          if (numWaitingThreads > 0) {
              numWaitingThreads--;
              threadQueue.V();
          }
       }
       public void signalCall() {
          while (numWaitingThreads > 0) {
             --numWaitingThreads;
             threadQueue.V();
          }
       }
       public void waitC() {
          numWaitingThreads++; threadQueue.VP(mutex);    mutex.P();
       }
       public boolean empty() { return (numWaitingThreads == 0); }
       public int length() { return numWaitingThreads; }
    }
    protected void enterMonitor() { mutex.P(); }
    protected void exitMonitor() { mutex.V(); }
}
```

Listing 4.23 Java monitor toolbox *monitorSC* with SC signaling.

4.7.2 A Toolbox for SU Signaling in Java

Listing 4.24 shows a Java monitor toolbox with SU signaling. The SU toolbox provides method *signalC_and_exitMonitor()*, which can be used when a signal operation is the last statement in a method (other than a return statement). When this method is called, the signaler does not wait in the *reentry* queue.

For example, method *deposit*() using *signalC_and_exitMonitor*() becomes:

```
public void deposit(int value) {
   enterMonitor();
   if (fullSlots == capacity)
      notFull.waitC();
   buffer[in] = value;
   in = (in + 1) % capacity;
   ++fullSlots;
   notEmpty.signalC_and_exitMonitor();
}
```

```
public class monitorSU { // monitor toolbox with SU signaling

   private binarySemaphore mutex = new binarySemaphore(1);
   private binarySemaphore reentry = new binarySemaphore(0);
   private int reentryCount = 0;
   proteced final class conditionVariable {
      private countingSemaphore threadQueue = new countingSemaphore(0);
      private int numWaitingThreads = 0;
      public void signalC() {
         if (numWaitingThreads > 0) {
            ++reentryCount;
            reentry.VP(threadQueue);
            --reentryCount;
         }
      }
      public void signalC_and_exitMonitor() { // does not execute reentry.P()
         if (numWaitingThreads > 0)threadQueue.V();
         else if (reentryCount > 0) reentry.V();
         else mutex.V();
      }
      public void waitC() {
         numWaitingThreads++;
         if (reentryCount > 0)   threadQueue.VP(reentry);
         else threadQueue.VP(mutex);
         --numWaitingThreads;
      }
      public boolean empty() { return (numWaitingThreads == 0); }
      public int length() { return numWaitingThreads; }
   }
   public void enterMonitor() { mutex.P(); }
   public void exitMonitor() {
      if (reentryCount > 0) reentry.V();
      else mutex.V();
   }
}
```

Listing 4.24 Java monitor toolbox *monitorSU* with SU signaling.

4.8 A Monitor Toolbox for Win32/C++/Pthreads

Listing 4.25 shows how to use the C++ *monitorSC* toolbox class to define a monitor for the bounded buffer problem.

```
class boundedBuffer : private monitorSC { // Note the use of private inheritance –
private:    // methods of monitorSC cannot be called outside of boundedbuffer.
   int fullSlots;    // # of full slots in the buffer
   int capacity;     // # of slots in the buffer
   int* buffer;
   int in, out;
   conditionVariable notFull;
   conditionVariable notEmpty;
public:
   boundedBuffer(int bufferCapacity);
   ~boundedBuffer();
   void deposit(int value, int ID);
   int withdraw(int ID);
};
```

Listing 4.25 Using the C++ *monitorSC* Toolbox class.

The *conditionVariable* constructor receives a pointer to the monitor object that owns the variable and uses this pointer to access the *mutex* object of the monitor.

```
   boundedBuffer(int bufferCapacity_) : fullSlots(0), capacity(bufferCapacity),
         in(0), out(0), notFull(this), notEmpty(this), buffer( new int[capacity]) {}
```

Monitor methods are written just as they are in Java:

```
    void boundedBuffer::deposit(int value) {
       enterMonitor();
       while (fullSlots == capacity)
          notFull.waitC();
       buffer[in] = value;
       in = (in + 1) % capacity;
       ++fullSlots;
       notEmpty.signalC();
       exitMonitor();
    }
```

4.8.1 A Toolbox for SC Signaling in C++/Win32/Pthreads

Listing 4.26 shows a C++/Win32/Pthreads monitor toolbox with SC signaling. Class *conditionVariable* is a friend of class *monitorSC*. This gives *conditionVariables* access to private member *mutex* of class *monitor*.

4.8.2 A Toolbox for SU Signaling in C++/Win32/Pthreads

Listing 4.27 shows a C++/Win32/Pthreads monitor toolbox with SU signaling.

```
class monitorSC { // monitor toolbox with SC signaling
protected:
   monitorSC() : mutex(1) {}
   void enterMonitor() { mutex.P(); }
   void exitMonitor() { mutex.V(); }
private:
   binarySemaphore mutex;
   friend class conditionVariable;   // conditionVariable needs access to mutex
};
class conditionVariable {
private:
   binarySemaphore threadQueue;
   int numWaitingThreads;
   monitorSC& m;  // reference to monitor that owns this conditionVariable
public:
   conditionVariable(monitorSC* mon) : threadQueue(0),numWaitingThreads(0),
      m(*mon) {}
   void signalC();
   void signalCall();
   void waitC();
   bool empty() { return (numWaitingThreads == 0); }
   int length() { return numWaitingThreads; }
};

void conditionVariable::signalC() {
   if (numWaitingThreads > 0) {
      --numWaitingThreads;
      threadQueue.V();
   }
}
void conditionVariable::signalCall() {
   while (numWaitingThreads > 0) {
      --numWaitingThreads;
      threadQueue.V();
   }
}
void conditionVariable::waitC(int ID) {
   numWaitingThreads++;
   threadQueue.VP(&(m.mutex));
   m.mutex.P();
}
```

Listing 4.26 C++ monitor toolbox for SC signaling.

```
class monitorSU { // monitor toolbox with SU signaling
protected:
   monitorSU() : reentryCount(0), mutex(1), reentry(0) { }
   void enterMonitor() { mutex.P(); }
   void exitMonitor(){
      if (reentryCount > 0) reentry.V();
      else mutex.V();
   }
private:
   binarySemaphore mutex;  binarySemaphore reentry;
   int reentryCount; friend class conditionVariable;
};
class conditionVariable {
private:
   binarySemaphore threadQueue;
   int numWaitingThreads;
   monitorSU& m;
public:
   conditionVariable(monitorSU* mon):threadQueue(0),numWaitingThreads(0),
      m(*mon){}
   void signalC();    void signalC_and_exitMonitor();
   void waitC();
   bool empty() { return (numWaitingThreads == 0); }
   int length() { return numWaitingThreads; }
};
void conditionVariable::signalC() {
   if (numWaitingThreads > 0) {
      ++(m.reentryCount);
      m.reentry.VP(&threadQueue);
      --(m.reentryCount);
   }
}
void conditionVariable::signalC_and_exitMonitor() {
   if (numWaitingThreads > 0) threadQueue.V();
   else if (m.reentryCount > 0) m.reentry.V();
   else m.mutex.V();
}
void conditionVariable::waitC() {
   numWaitingThreads++;
   if (m.reentryCount > 0) threadQueue.VP(&(m.reentry));
   else threadQueue.VP(&(m.mutex));
   --numWaitingThreads;
}
```
Listing 4.27 C++ monitor toolbox for SU signaling.

4.9 Nested Monitor Calls

A thread T executing in a method of monitor M1 may call a method in another monitor M2. This is called a nested monitor call.

If thread T releases mutual exclusion in M1 when it makes the nested call to M2, it is said to be an *open* call. If mutual exclusion is not released, it is a *closed* call. Closed calls are prone to create deadlocks.

```
    class First extends monitorSU {  class Second extends monitorSU {
       Second M2;
       public void A1() {               public void A2() {
         …                                …
         M2.A2();                         wait();          // Thread A is blocked
         …                                …
       }                                }
       public void B1() {               public void B2() {
         M2.B2();                         …
         …                                signal-and-exit();  // wakeup Thread A
       }                                }
}
```

Suppose that we create an instance M1 of the First monitor and that this instance is used by two threads:

Thread A	Thread B
M1.A1();	M1.B1();

- Assume that Thread *A* enters method *A1()* of monitor *M1* first and makes a closed monitor call to *M2.A2()*.
- Assume that Thread *A* is then blocked on the *wait* statement in method *A2()*.
- Thread *B* intends to signal Thread *A* by calling method *M1.B1*, which issues a nested call to *M2.B2()* where the signal is performed.
- But this is impossible since Thread *A* retains mutual exclusion for monitor *M1* while Thread *A* is blocked on the wait statement in monitor *M2*.
- Thus, Thread *B* is unable to enter *M1* and a deadlock occurs.

Open monitor calls are implemented by having the calling thread release mutual exclusion when the call is made and reacquire mutual exclusion when the call returns.

The monitor toolboxes described in the previous section make this easy to do. For example, method *A1()* above becomes:

```
public void A1() {
   enterMonitor();// acquire mutual exclusion
   …
   exitMonitor();  // release mutual exclusion
   M2.A2();
   enterMonitor();// reacquire mutual exclusion
   …
   exitMonitor();  // release mutual exclusion
}
```

This gives equal priority to the threads returning from a nested call and the threads trying to enter the monitor for the first time, since both groups of threads call *enterMonitor()*.

Open calls can create a problem if shared variables in monitor *M1* are used as arguments and passed by reference on nested calls to *M2*.

This allows shared variables of *M1* to be accessed concurrently by a thread in *M1* and a thread in *M2*, violating the requirement for mutual exclusion.