```
import Utilities.*;
import Synchronization.*;

class CountingSemaphoreFromBinary extends MyObject {
   private int count = 0;
   private BinarySemaphore mutex = new BinarySemaphore(1);
   private BinarySemaphore delay = null;

   public CountingSemaphoreFromBinary(int n) {
      if (n < 0) throw new IllegalArgumentException();
      count = n;  // initial value of delay is 0 if count is 0 else 1
      delay = new BinarySemaphore(count>0 ? 1 : 0);
   }

   public void down() {
      P(delay);
      P(mutex);
      count--;
      if (count > 0) V(delay);
      V(mutex);
   }

   public void up() {
      P(mutex);
      count++;
      if (count == 1) V(delay);
      V(mutex);
   }
}
```

17. **Algorithm Animation.** Animate one of Class 4.4, Program 4.8, or Class 4.13 (bounded buffer producer and consumer, sleeping barber, or database readers and writers, respectively).

Chapter 5

Monitors

Semaphores are like `goto`s and pointers in that programming with them is prone to mistakes. Semaphores work but they lack structure and discipline. For example, a disastrous easy-to-make coding mistake is the following.

`V(S);` *critical section*; `V(S);`

And, as described in Section 4.5, the following mistake leads to deadlock.

`P(S);` *critical section*; `P(S);`

Even when semaphores are used correctly in a program for mutual exclusion and condition synchronization, it is difficult to distinguish the two uses without a thorough examination of all the code.

Monitors were devised to help avoid these kinds of mistakes and to provide a higher level programming language construct than semaphores. Monitors are related to objects and object-oriented programming, to abstract data types and data encapsulation. They are the built-in synchronization primitive of Java.

In this chapter, we define monitors and condition variables and describe two signaling disciplines that monitors use: signal and exit and signal and continue. An experiment shows that Java monitors uses the latter. We look at signal-and-exit monitor solutions to the bounded buffer, dining philosophers, and readers and writers classic synchronization problems, using a Java-like pseudocode. Then we look at Java programs having signal-and-continue monitors that also solve the three problems. An examination of the code for the `BinarySemaphore` and `CountingSemaphore` library classes was deferred from the previous chapter. We now see how they are implemented as monitors. A lock is like a binary semaphore with the restriction that the thread releasing a lock must be the one that acquired it. We look at a monitor implementation.

Even though Java lacks individually named condition variables, notification objects substitute for them in many situations. We see the three classic synchronization problems solved with notification objects. Since monitors can be implemented with semaphores, as will be shown, the two tools are equal in power in that they solve the same types of problems. Last, we look at the implementation of the `XtangoAnimation` algorithm animation package. It starts several threads when used, and its code raises many synchronization issues. After reading this chapter, you will understand the advantages monitors have as a higher level construct than semaphores. You will also be aware of the precautions a programmer must take when using Java's signal-and-continue monitors.

5.1 Definitions

A *monitor* ([15], Section 5.3), ([34], Section 6.7), ([36], Section 4.5), ([41], Section 2.2.7), ([43], Section 2.2.6) is a programming language module or object that encapsulates several service or access methods and their global and local variables. The only way to manipulate or access the variables inside the monitor is to call one of the service methods. Only one thread at a time is allowed to be active inside the monitor, that is, executing one of the service methods. This mutually exclusive access prevents race conditions involving the variables inside the monitor that could otherwise occur if several threads were active inside service methods at the same time. Each monitor object has a lock. The compiler generates code at the beginning of each service method to acquire the lock and at the end to release the lock. If the monitor is locked when a thread calls one of the monitor's service methods, the thread blocks and is added to the set of threads waiting to acquire the lock.

Mutual exclusion synchronization is therefore implicit in monitors, guaranteed by the compiler. To provide a mechanism for event or condition synchronization, a monitor may contain *condition variables*. The two operations on a condition variable are *signal* and *wait*. These are analogous to the "up" and "down" operations on binary semaphores. A thread that waits on a condition variable temporarily leaves the monitor and blocks, releasing the lock and joining the set of threads blocked on that condition variable. Each signal on a condition variable awakens one thread from the condition variable's set of blocked threads, not necessarily the one that has been blocked the longest. If the set is empty, the signal is not saved and has no effect, in contrast to the way semaphores work. Since releasing the monitor lock and joining the condition variable queue is an atomic action, there is no danger of lost wakeup signals (Section 3.5.4). A thread awakened by a signal is removed from the set of threads blocked on the condition variable and added to the set of threads waiting to acquire the monitor's lock. Once the lock is reacquired, the thread continues execution of the service method it had called earlier. See Figure 5.1. A monitor condition variable has no value and is best thought of as the name of the set of threads blocked on that condition variable. Note that condition variables and semaphores differ in two ways: a signal on an empty condition variable has no effect whereas a V increments a semaphore on which no thread is blocked, and a wait on a condition variable always blocks the thread until a subsequent signal whereas a P decrements a semaphore whose value is positive without blocking.

Monitor condition variables are implemented with one of several signaling disciplines: *signal and exit*, *signal and wait*, and *signal and continue*. In signal and exit, if a thread executing inside the monitor signals on a condition variable, it is required to leave the monitor immediately after generating the signal by executing a return statement in the service method it invoked. A thread from the wait set

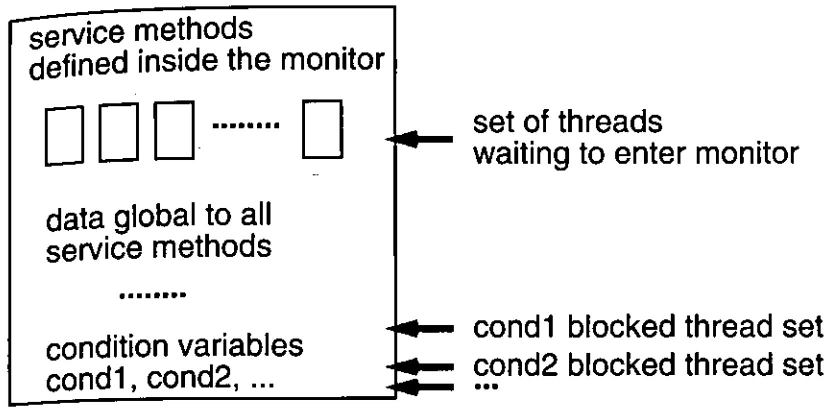

Figure 5.1: Monitor Service Methods, Condition Variables, and Sets.

of that condition variable is awakened and continues executing inside the monitor. It gets priority to execute inside the monitor over all threads waiting to enter the monitor via one of its service methods. In signal and wait, the signaled thread executes inside the monitor while the signaler waits for it to leave the monitor; then the signaler continues. In signal and continue, the signaled thread waits for the signaler to leave the monitor before it resumes execution inside the monitor. Neither signal and wait nor signal and continue requires a thread to execute a return statement after generating a signal. See ([1], Chapter 6) and [12] for much more thorough treatments of monitors and their various signaling disciplines.

Java monitors use the signal-and-continue discipline. We look first at signal and exit since it is easier to understand, particularly for those working with monitors for the first time. Solutions to the classical problems are shown in a Java-like pseudocode. Monitor service methods are indicated with the keyword `synchronized`; only one active thread at a time is permitted. Nonsynchronized methods in a monitor are of a private utility nature or make a sequence of synchronized method calls. The `wait(`*condVar*`)` method is used to wait on a condition variable; `notify(`*condVar*`)` is used to signal. Section 5.3 examines signal and continue, illustrated with Java solutions to the classical problems.

5.2 Signal and Exit

In this section, we make the following two assumptions about condition variables and their signaling discipline.

1. After signaling on a condition variable, the signaler must exit the monitor immediately by executing a return statement so that no variables change before the signaled thread wakes up and continues executing inside the monitor. In this way the signaled thread finds the condition that led to the signal still true when it resumes execution inside the monitor.

2. The signaled thread is given priority to proceed inside the monitor over those threads waiting to enter the monitor through a service method call. Again, the signaled thread finds the condition still true when it resumes execution inside the monitor.

We will see in Section 5.3 that Java does not work like this. However, for convenience we use a pseudocode that resembles Java to express the signal-and-exit monitor examples in this section.

5.2.1 The Bounded Buffer Producer and Consumer

The bounded buffer monitor in Class 5.1 synchronizes a single producer thread and single consumer thread. It is written in a Java-like pseudocode.

```
public synchronized void deposit(double data) {
   if (count == size) wait(notFull);
   buf[rear] = data;
   rear = (rear+1) % size;
   count++;
   if (count == 1) notify(notEmpty);
}
public synchronized double fetch() {
   double result;
   if (count == 0) wait(notEmpty);
   result = buf[front];
   front = (front+1) % size;
   count--;
   if (count == size-1) notify(notFull);
   return result;
}
```

Only one thread executes at a time in the monitor, having called the `fetch` or `deposit` synchronized method. If the buffer is full, the producer thread blocks with the `wait` method on the condition variable `notFull`; the producer is later unblocked by the consumer with `notify` when it frees up a buffer slot. If the buffer is empty, the consumer thread blocks on the condition variable `notEmpty`; the consumer is later unblocked when the producer fills a slot with an item.

The monitor in Class 5.1 could be used, if it were real Java code, with the driver code in Program 3.9 with no changes to either. It is important to note that the `BoundedBuffer` class in Class 4.4 is not the same thing as a monitor; more than one thread at a time may be active inside its methods. We used semaphores inside the methods of that class for synchronization and to prevent race conditions. Mutual exclusion synchronization is implicit in a monitor and condition variables are used for condition synchronization.

5.2.2 The Dining Philosophers

The monitor in Class 5.2 coordinates the dining philosophers.

```
public void takeForks(int i) { hungryAndGetForks(i); }
public void putForks(int i) {
   finishedEating(i);
   checkForkDown(left(i)); checkForkDown(right(i));
}
private void seeIfStarving(int k) {
   if (state[k] == HUNGRY && state[left(k)] != STARVING
         && state[right(k)] != STARVING) {
      state[k] = STARVING;
   }
}
private void test(int k, boolean checkStarving) {
   if (state[left(k)] != EATING && state[left(k)] != STARVING &&
         && (state[k] == HUNGRY || state[k] == STARVING) &&
       state[right(k)] != STARVING && state[right(k)] != EATING)
      state[k] = EATING;
   else if (checkStarving)
      seeIfStarving(k); // simplistic naive check for starvation
}
private synchronized void hungryAndGetForks(int i) {
   state[i] = HUNGRY;
   test(i, false);
   if (state[i] != EATING) wait(self[i]);
}
private synchronized void finishedEating(int i) {
   state[i] = THINKING;
}
private synchronized void checkForkDown(int i) {
   test(i, checkStarving);
   if (state[i] == EATING) notify(self[i]);
}
```

The monitor has an array of condition variables, one array entry for each philosopher to block on if its forks are not both available. A naive form of starvation detection and prevention is implemented. A hungry philosopher enters the "starving" state if it cannot pick up both forks to eat whenever one of its neighbors puts down its forks. A hungry philosopher is not allowed to eat if it has a starving neighbor. The monitor in Class 5.2 could be used, if it were real Java code, with the driver code in Program 4.10 (modified to process a -S command line option that turns on starvation checking).

Since the signal-and-exit discipline requires a thread to exit the monitor immediately after signaling on a condition variable, a thread cannot perform two signal operations on two condition variables during one call to a monitor service method. To handle this situation, the philosopher code calls the public nonsynchronized method putForks when the philosopher puts down its forks. This method makes two calls to the monitor's private synchronized method checkForkDown; each such call generates at most one signal.

5.2.3 The Readers and Writers

The monitor in Class 5.3 solves the readers and writers problem and is based on ([6], Figure 5.5).

```
public synchronized void startRead(int i) {
   if (isWriting) wait(OKtoRead);
   else if (!empty(OKtoWrite)) {
      // new incoming readers cannot starve writers
      wait(OKtoRead);
   }
   numReaders++;
   // when a writer finishes, all waiting readers start
   notify(OKtoRead);
}
public synchronized void endRead(int i) {
   numReaders--;
   if (numReaders == 0) notify(OKtoWrite);
}
public synchronized void startWrite(int i) {
   if (numReaders != 0 || isWriting) wait(OKtoWrite);
   isWriting = true;
}
public synchronized void endWrite(int i) {
   isWriting = false;
   if (!empty(OKtoRead)) notify(OKtoRead);
   else notify(OKtoWrite); // nor do writers starve readers
}
```

It implements a fair, starvation-free solution; that is, a continual stream of arriving readers cannot delay for an arbitrary amount of time a writer from writing. The Boolean method empty(*condVar*) returns true if the condition variable's queue of waiting threads is empty. Whenever a writer finishes, it sweeps into the database all waiting readers, called a platoon or batch, even those that arrived after any waiting writer. (The other approach to preventing starvation is strict serialization. See Exercise 4.6.) The monitor in Class 5.3 could be used, if it were real Java code, with the driver code in Program 4.14 with no changes to either.

5.3 Signal and Continue

The signal-and-continue discipline does not require the signaling thread to leave the monitor after a signal on a condition variable. Nor does the signaled thread have priority to proceed in the monitor over other threads waiting to enter through a service method call. Relaxing the first requirement simplifies the coding of some monitors. In Class 5.2 we were forced to introduce a service method, putForks, that invokes two potentially signal generating private methods in the monitor whenever a philosopher puts down its forks. With signal and continue, only one call is needed, as we will see in Class 5.6 below.

There are disadvantages in relaxing the second requirement on page 132, the one giving signaled threads priority to proceed in the monitor over threads trying to enter through a service method ([12], page 99). We cannot guarantee that the condition leading to the signal is still true when the signaled thread reenters the monitor: before leaving the monitor, the signaling thread could change data fields and alter the state of the monitor after generating the signal. Also threads trying to enter the monitor through a service method can "barge" ahead of signaled threads that were waiting on condition variables, resulting in an unbounded waiting time on a condition variable, a form of starvation.

For example, consider a signal-and-continue bounded buffer monitor for multiple producer and multiple consumer threads. A producer that wants to deposit into a full buffer waits on a condition variable for a signal from a consumer. After getting such a signal, it is possible for another producer to "barge" into the monitor and fill the just-emptied slot. Since we no longer guarantee that the condition leading to the signal is still true when the signaled thread proceeds in the monitor, we need to take the precaution of changing the if statement containing the wait,

```
if (condition) wait();
```

to a while loop,

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

Since potentially more than one thread is signaled before the signaler leaves the monitor, the signal condition may no longer be true when a signaled thread reenters the monitor ([12], page 100). Further, some other thread may have entered the monitor through a service method and modified the condition before the signaled thread reenters the monitor.

If a while loop is used instead of an if by a thread to wait on a condition variable, then a broadcast signal becomes feasible. The signaler broadcasts a signal to all threads waiting on the condition variable as a hint if the signaler thinks but is not positively sure that a waiting thread can proceed. Java supports a broadcast signal with the notifyAll method.

5.3.1 Java Monitors

To construct a Java monitor, use the modifier synchronized for all methods in which only one thread should be executing at a time. These methods are usually public, but may be private if the public access to the monitor consists of calls to several private synchronized methods, as was done in the pseudocode example Class 5.2.

Each Java object has a lock associated with it. A thread invoking a synchronized method in an object must obtain the object's lock before executing the method's code. The thread blocks if the lock is currently held by some other thread. A synchronized method in an object

```
class Name {
   synchronized type method() {
      ...
   }
}
```

is equivalent to a method whose code consists of a single block synchronized on the object.

```
class Name {
   type method() {
      synchronized (this) {
         ...
      }
   }
}
```

Static methods in a class may also be declared synchronized; in this case, a class lock must be obtained. For example, the constructor in class B

```
class B {
   private static int numConstructed = 0;
   private synchronized void inc() {
      numConstructed++;
   }
   public B() {
      inc();
   }
}
```

increments a counter by calling a static synchronized method to avoid lost updates.

Unfortunately, each Java monitor has just one (anonymous) condition variable; all waits and signals refer to it automatically. Use wait() and notify() to wait for and generate a signal, respectively; use notifyAll() to wake up or send signals to all the threads waiting on the anonymous condition variable (broadcast signal).

Java monitors use the signal-and-continue signaling discipline, as the experiment set up in Program 5.4 demonstrates. The sample output illustrates the following differences between the signal-and-exit and signal-and-continue signaling disciplines.

- Java allows a thread trying to get in the monitor through a service method to "barge" ahead and enter before a signaled thread continues inside the monitor.

- A notify() moves to the ready queue one of the waiting threads, an arbitrary one and not necessarily the one that has waited the longest.
- A notifyAll() moves all waiting threads to the ready queue, from which they are scheduled to run on the CPU in an unspecified order.

5.3.2 The Bounded Buffer Producer and Consumer

The Java monitor in Class 5.5 solves the bounded buffer single producer and single consumer problem.

```
public synchronized void deposit(double value) {
   while (count == numSlots)
      try { wait(); } catch (InterruptedException e) {}
   buffer[putIn] = value;
   putIn = (putIn + 1) % numSlots;
   count++;              // wake up the consumer since
   if (count == 1) notify();  // it might be waiting
}
public synchronized double fetch() {
   double value;
   while (count == 0)
      try { wait(); } catch (InterruptedException e) {}
   value = buffer[takeOut];
   takeOut = (takeOut + 1) % numSlots;
   count--;                      // wake up the producer since
   if (count == numSlots-1) notify(); // it might be waiting
   return value;
}
```

This monitor is used with the classes (ProducerConsumer, Producer, and Consumer) in Program 3.9 with no changes. The producer thread waits if the buffer is full, and the consumer thread waits if the buffer is empty. Whenever the producer puts an item into an empty buffer, it signals since the consumer might be waiting. Whenever the consumer extracts an item from a full buffer, it signals since the producer might be waiting. Notice there are no race conditions or lost updates involving the variable count.

5.3.3 The Dining Philosophers

The Java monitor in Class 5.6 coordinates the dining philosophers.

```
class DiningServer ...
public synchronized void takeForks(int i) {
   state[i] = HUNGRY;
   test(i, false);
   while (state[i] != EATING)
```

```
         try {wait();} catch (InterruptedException e) {}
   }
   public synchronized void putForks(int i) {
      state[i] = THINKING;
      test(left(i), checkStarving);
      test(right(i), checkStarving);
      notifyAll();
   }
   private void seeIfStarving(int k) {
      if (state[k] == HUNGRY && state[left(k)] != STARVING
            && state[right(k)] != STARVING) {
         state[k] = STARVING;
      }
   }
   private void test(int k, boolean checkStarving) {
      if (state[left(k)] != EATING && state[left(k)] != STARVING
            && (state[k] == HUNGRY || state[k] == STARVING) &&
          state[right(k)] != STARVING && state[right(k)] != EATING)
         state[k] = EATING;
      else if (checkStarving)
         seeIfStarving(k); // simplistic naive check for starvation
   }
```

The simple starvation prevention scheme of Class 5.2 is used. This monitor is used with the classes in Program 4.10 (modified to process a -S command line option that turns on starvation checking).

Since Java supports only a single anonymous condition variable in its monitors, we cannot use an array of condition variables, one per philosopher, for blocking a hungry philosopher that cannot eat. Instead, a notifyAll() unblocks all waiting philosophers when any philosopher puts down its forks. This is very inefficient since at most two waiting philosophers need be signaled to check their forks.

Two sample runs are shown in Class 5.6. In the first, the thinking and eating times are set up so that philosophers 0 and 2 alternate their eating cycles, in effect starving philosopher 1 between them. The second run turns on starvation checking with -S on the command line; philosopher 1 eats and does not starve.

```
% java DiningPhilosophers \
     -S -R120 1 100 10 1 1 100 1000000 1 1000000 1
DiningPhilosophers:
   numPhilosophers=5, checkStarving=true, runTime=120
DiningServer: checkStarving=true
Philosopher 0 is alive, napThink=1000, napEat=100000
Philosopher 1 is alive, napThink=10000, napEat=1000
Philosopher 2 is alive, napThink=1000, napEat=100000
Philosopher 3 is alive, napThink=1000000000, napEat=1000
Philosopher 4 is alive, napThink=1000000000, napEat=1000
```

```
All Philosopher threads started
age()=110, Philosopher 0 is thinking for 623 ms
age()=170, Philosopher 1 is thinking for 739 ms
age()=170, Philosopher 2 is thinking for 304 ms
age()=220, Philosopher 3 is thinking for 625066794 ms
age()=220, Philosopher 4 is thinking for 852766912 ms
age()=550, Philosopher 2 wants to eat
age()=550, Philosopher 2 is eating for 92594 ms
age()=880, Philosopher 0 wants to eat
age()=880, Philosopher 0 is eating for 30529 ms
age()=990, Philosopher 1 wants to eat
philosopher 1 is STARVING
age()=31360, Philosopher 0 is thinking for 545 ms
age()=31910, Philosopher 0 wants to eat
age()=93150, Philosopher 2 is thinking for 462 ms
age()=93150, Philosopher 1 is eating for 59 ms
age()=93260, Philosopher 1 is thinking for 8682 ms
age()=93260, Philosopher 0 is eating for 22245 ms
age()=93650, Philosopher 2 wants to eat
age()=93650, Philosopher 2 is eating for 92309 ms
age()=101940, Philosopher 1 wants to eat
philosopher 1 is STARVING
age()=115510, Philosopher 0 is thinking for 614 ms
age()=116110, Philosopher 0 wants to eat
age()=120120, time to stop the Philosophers and exit
```

5.3.4 The Readers and Writers

The Java monitor in Class 5.7 synchronizes the database readers and writers and is based on ([6], Figure 5.5).

```
public synchronized void startRead(int i) {
   long readerArrivalTime = 0;
   if (numWaitingWriters > 0 || numWriters > 0) {
      numWaitingReaders++;
      readerArrivalTime = age();
      while (readerArrivalTime >= startWaitingReadersTime)
         try {wait();} catch (InterruptedException e) {}
      numWaitingReaders--;
   }
   numReaders++;
}
public synchronized void endRead(int i) {
   numReaders--;
   okToWrite = numReaders == 0;
   if (okToWrite) notifyAll();
   }
   public synchronized void startWrite(int i) {
      if (numReaders > 0 || numWriters > 0) {
         numWaitingWriters++;
         okToWrite = false;
         while (!okToWrite)
            try {wait();} catch (InterruptedException e) {}
         numWaitingWriters--;
      }
      okToWrite = false;
      numWriters++;
   }
   public synchronized void endWrite(int i) {
      numWriters--;              // ASSERT(numWriters==0)
      okToWrite = numWaitingReaders == 0;
      startWaitingReadersTime = age();
      notifyAll();
   }
```

It implements a fair, starvation-free solution: a continual stream of arriving readers cannot delay for an arbitrary amount of time a writer from writing. A reader wishing to read the database must wait if there is a writer currently accessing the database or a waiting writer. The reader's time of arrival at the database is saved in a local variable. Whenever a writer finishes, it sweeps into the database a platoon of **all** waiting readers. (See Exercise 4.6 for the strict serialization approach.) The platoon consists of all those waiting readers who arrived at the database before the current writer finished writing. This monitor is used with the classes (`ReadersWriters`, `Reader`, and `Writer`) in Program 4.14 with no changes.

From the sample output at times 1210 and 4390, we see that if a reader wants to read the database when there is a waiting writer, the reader must wait. Therefore, writers do not starve.

```
All threads started
age=440, Reader0 wants to read
age=440, Reader0 reading for 1530 ms
age=490, Reader1 wants to read
age=490, Reader1 reading for 93 ms
age=600, Reader1 finished reading
age=600, Reader1 napping for 1931 ms
age=1040, WRITER1 wants to write
age=1210, Reader2 wants to read   reader 2 waits to go after writer 1
age=1320, WRITER0 wants to write
age=1430, Reader3 wants to read
age=1590, WRITER2 wants to write
age=1760, Reader4 wants to read
age=1980, Reader0 finished reading
```

```
age=1980, Reader0 napping for 329 ms
age=1980, WRITER1 writing for 1144 ms
age=2310, Reader0 wants to read
age=2530, Reader1 wants to read
age=3130, WRITER1 finished writing   now all waiting readers can go
age=3130, WRITER1 napping for 774 ms
age=3130, Reader4 reading for 985 ms
age=3180, Reader2 reading for 1007 ms
age=3240, Reader3 reading for 472 ms
age=3240, Reader0 reading for 1436 ms
age=3240, Reader1 reading for 1894 ms
...
```

However, the readers and writers do not enter the database to perform their operations in exactly the same order they arrived and queued up at the database. In other words, the servicing order is not strictly FIFO. When a writer exits the database, **all** waiting readers enter, even those that arrived after a waiting writer.

5.4 Deadlock

Section 4.5 warns that care must be taken to avoid deadlock when semaphores are used to protect nested critical sections or allocate different types of resources. Nested monitor calls and nested synchronized blocks are also subject to deadlock. Suppose class `S` has the structure

```
class S {
  synchronized void f(T t) {
    ...  t.g(...); ...
  }
}
```

and class `T` has the structure

```
class T {
  synchronized void g(S s) {
    ...  s.f(...); ...
  }
}
```

Here `s` and `t` are references to two monitor objects created from classes `S` and `T`, respectively, and shared by two threads A and B. If the following sequence of events occurs,

A: calls **`s.f(t)`**
B: calls **`t.g(s)`**
A: blocks on call to `t.g`
B: blocks on call to `s.f`

the two threads deadlock.

The deadlock prevention scheme described in Section 4.5 can also be used with monitors: globally order all monitor objects and require all threads to lock monitor objects (through synchronized method calls) in the same order. In the case of synchronized code blocks (implicit binary semaphores), the same ordering idea can be used with the objects locked by each block.

5.5 Binary and Counting Semaphore Monitors

Chapter 4 on semaphores uses two classes from this book's `Synchronization` package, `BinarySemaphore` and `CountingSemaphore`, to create semaphore objects for mutual exclusion and condition synchronization. The JDK does not include either a binary or counting semaphore class. However, they are easily implemented as Java monitors with `P` and `V` methods. Both the binary and counting semaphore classes extend the abstract class `Semaphore`, shown in Library Class 5.1.

```
public abstract class Semaphore {
   protected int value = 0;
   protected Semaphore() {value = 0;}  // constructors
   protected Semaphore(int initial) {value = initial;}
   public synchronized void P() {
      value--;
      if (value < 0)
         try { wait(); } catch (InterruptedException e) {}
   }
   public synchronized void V() {
      value++;   if (value <= 0) notify();
   }
}
```

The semaphore's `value` field ranges over both positive and negative integers. A positive or zero `value` corresponds to the conventional value of a semaphore. A negative value means there are threads blocked on the semaphore, waiting to be released with a `V`; the absolute value of the `value` field is the number of blocked threads. This implementation prevents barging: a thread calling `P` always waits if there are other waiting threads (negative `value`), even if it obtains the semaphore object lock before a signaled thread. When a `V` is done, the JVM does not guarantee that the thread waiting the longest is the next one active in the monitor

The `Semaphore` class `P` method uses a `wait()` that is part of an `if` statement instead of a `while` loop, in contrast to the recommendation of Section 5.3. This is because a thread blocked in a `P` is not waiting for some condition to become true but is waiting for permission to proceed, which comes from the `notify()` in the `V` method. The code handling an `InterruptedException` is not shown.

This class provides several methods, not shown above, for semaphores in addition to `P` and `V`: the `toString` method converts the semaphore into a string for `System.out.println`, the `value` method reads the semaphore's value, and the nonblocking

`tryP` method decrements the semaphore's value, if possible, but throws an exception instead of blocking otherwise. The code for the binary and counting semaphore classes extended from `Semaphore` in shown in Library Classes 5.2 and 5.3.

```
public class BinarySemaphore extends Semaphore {
   public BinarySemaphore() {super();}  // constructors
   public BinarySemaphore(int initial)
      {super((initial!=0) ? 1:0);}
   public BinarySemaphore(boolean initial)
      {super(initial ? 1:0);}
   public final synchronized void V() {
      super.V();
      if (value > 1) value = 1; // cap the value
   }
}
public class CountingSemaphore extends Semaphore {
   public CountingSemaphore() {super();}  // constructors
   public CountingSemaphore(int initial) {super(initial);}
}
```

5.6 Locks

In contrast to an implicit binary semaphore in the form of a synchronized block, an explicit binary semaphore may be released by any thread, not necessarily the one that acquired it. The lock implemented in Program 5.8 acts like an explicit binary semaphore except only the owner of the lock is permitted to unlock it and every lock's initial state is unlocked.

```
public synchronized void lock() {
   Thread who = Thread.currentThread();
   if (owner == who) return; // allow owner to relock silently
   else if (owner == null) {
      owner = who;
      return;
   } else /* owner != who && owner != null */ {
      while (owner != null) { // since ``barging'' is possible
         try {    // in Java monitors, a while loop is needed
            wait(); // even though only one thread is awakened
         } catch (InterruptedException e) {}
         // here 'owner' can be null or
         // another thread if it barged in
      }
      owner = who;
      return;
   }
}
```

```
public synchronized void unlock()
      throws IllegalMonitorStateException {
   Thread who = Thread.currentThread(); // allow an unlocked
   if (owner == null) return;  // lock to be unlocked silently
   else if (owner != who) {
      throw new IllegalMonitorStateException();
   } else /* owner == who */ {
      owner = null;
      notify(); // at most one thread can proceed if
   }          // any are waiting so notifyAll() not
}             // needed
```

If a thread not owning the lock tries to unlock it, an IllegalMonitorStateException object is thrown. An unlocked lock may be unlocked and a lock owner may relock the lock it owns, both without generating an exception. One of these locks should be used instead of a synchronized block (implicit binary semaphore) if the lock is acquired in one method and released in another method, both by the same thread.

5.7 Notification Objects

It is possible to use an object somewhat like a (named) condition variable in a Java monitor.

```
shared object:
   Object obj = new Object();
in one thread:
   synchronized (obj) {...
      if (!condition)
         try { obj.wait(); }
         catch (InterruptedException e) {}
      ...
   }
in another thread:
   synchronized (obj) {...
      if (condition) obj.notify();
      ...
   }
```

The shared object obj is used as a *notification* object by the two threads. While inside a synchronized block on obj, one thread checks conditions to see if it should continue; if not, it waits. Another thread changes conditions and notifies the waiting thread. When used inside a monitor, a notification object plays the role of a named condition variable.

```
class Name {
```

```
   Object obj = new Object();
   other data fields;
   public type method1(...) {...
      synchronized (obj) {...
         if (!method2(...))
            try { obj.wait(); }
            catch (InterruptedException e) {}
         ...
      }
      ...
   }
   private synchronized boolean method2(...) {...}
   public type method3(...) {...
      synchronized (obj) {...
         if (method4(...)) obj.notify();
         ...
      }
      ...
   }
   private synchronized boolean method4(...) {...}
}
```

A thread calls a nonsynchronized service method in the monitor, enters a synchronized block on a notification object, and then calls a private synchronized method. If conditions are not right to continue, the thread waits inside the notification object for a signal. Since the threads are performing nested object locking, the programmer must take care to prevent deadlock, as described in Section 5.4. In the above code skeleton, the locks are always requested in the same order.

Class 5.9 moves the code inside the elements and spaces semaphores of Class 4.4 into the bounded buffer implementation.

```
private Object conveyD = new Object();
private Object conveyF = new Object();
...
public void deposit(double value) {
   synchronized (conveyD) {
      spaces--;   // grab a space or wait behind others
      if (spaces < 0) {
         try { conveyD.wait(); }
         catch (InterruptedException e) {}
      }
      buffer[putIn] = value;
      putIn = (putIn + 1) % numSlots;
   }
   synchronized (conveyF) {
      elements++; // signal a waiting consumer if there is one
```

```
         if (elements <= 0) conveyF.notify();
      }
   }
   public double fetch() {
      double value;
      synchronized (conveyF) {
         elements--; // grab an element or wait behind others
         if (elements < 0) {
               try { conveyF.wait(); }
               catch (InterruptedException e) {}
         }
         value = buffer[takeOut];
         takeOut = (takeOut + 1) % numSlots;
      }
      synchronized (conveyD) {
         spaces++;   // signal a waiting producer if there is one
         if (spaces <= 0) conveyD.notify();
      }
      return value;
   }
```

This class works with multiple producer threads and multiple consumer threads. There are two notification objects, one for waiting producers and one for waiting consumers. Notice there are no synchronized methods in this example. The code handling an `InterruptedException` is not shown.

Class 5.10 shows a `DiningServer` class with an array of notification objects, `convey`, one array entry per philosopher.

```
   private Object[] convey = new Object[numPhils];
      for (int i = 0; i < numPhils; i++) convey[i] = new Object();
   ...
   public void takeForks(int i) {
      synchronized (convey[i]) {
         if (hungryAndGetForks(i)) return;
         else try { convey[i].wait(); }
            catch (InterruptedException e) {}
      }
   }
   public synchronized void putForks(int i) {
      state[i] = THINKING;
      test(left(i), checkStarving);
      test(right(i), checkStarving);
      if (state[left(i)] == EATING) forksAvailable(left(i));
      if (state[right(i)] == EATING) forksAvailable(right(i));
   }
   private synchronized boolean hungryAndGetForks(int i) {
```

```
      state[i] = HUNGRY;
      test(i, false);
      return state[i] == EATING;
   }
   private void forksAvailable(int i) {
      synchronized (convey[i]) { convey[i].notify(); }
   }
   private void test(int k, boolean checkStarving) {
      if (state[left(k)] != EATING && state[left(k)] != STARVING
            && (state[k] == HUNGRY || state[k] == STARVING) &&
          state[right(k)] != STARVING && state[right(k)] != EATING)
         state[k] = EATING;
      else if (checkStarving)
         seeIfStarving(k); // simplistic naive check for starvation
   }
   private void seeIfStarving(int k) {
      if (state[k] == HUNGRY && state[left(k)] != STARVING
            && state[right(k)] != STARVING) {
         state[k] = STARVING;
      }
   }
```

If the forks are not available when the philosopher gets hungry, it waits inside its notification object for a signal. This is an example of a notification: the philosopher registers its request with the server, and if the resources are not available, waits for a notification signal inside an object created for this use. An `InterruptedException` is not handled (Exercise 10).

Class 5.11 implements a starvation-free synchronization algorithm for the readers and writers. It uses a local notification object, convey, for each thread to wait inside until it can access the database.

```
   private int numReaders = 0;
   private boolean isWriting = false;
   private Vector waitingReaders = new Vector();
   private Vector waitingWriters = new Vector();
   ...
   public void startRead(int i) {
      Object convey = new Object();
      synchronized (convey) {
         if (cannotReadNow(convey))
            try { convey.wait(); }
            catch (InterruptedException e) {}
      }
   }
   private synchronized boolean cannotReadNow(Object convey) {
      boolean status;
      if (isWriting || waitingWriters.size() > 0) {
         waitingReaders.addElement(convey);  status = true;
      } else {
         numReaders++;  status = false;
      }
      return status;
   }
   public void startWrite(int i) {
      Object convey = new Object();
      synchronized (convey) {
         if (cannotWriteNow(convey))
            try { convey.wait(); }
            catch (InterruptedException e) {}
      }
   }
   private synchronized boolean cannotWriteNow(Object convey) {
      boolean status;
      if (isWriting || numReaders > 0) {
         waitingWriters.addElement(convey);  status = true;
      } else {
         isWriting = true;  status = false;
      }
      return status;
   }
   public synchronized void endRead(int i) {
      numReaders--;
      if (numReaders == 0 && waitingWriters.size() > 0) {
         synchronized (waitingWriters.elementAt(0)) {
            waitingWriters.elementAt(0).notify();
         }
         waitingWriters.removeElementAt(0);
         isWriting = true;
      }
   }
   public synchronized void endWrite(int i) {
      isWriting = false;
      if (waitingReaders.size() > 0) {
         while (waitingReaders.size() > 0) {
            synchronized (waitingReaders.elementAt(0)) {
               waitingReaders.elementAt(0).notify();
            }
            waitingReaders.removeElementAt(0);
            numReaders++;
         }
      } else if (waitingWriters.size() > 0) {
```

```
         synchronized (waitingWriters.elementAt(0)) {
            waitingWriters.elementAt(0).notify();
         }
         waitingWriters.removeElementAt(0);
         isWriting = true;
      }
   }
```

The platoon strategy, as opposed to strict serialization, is used in preventing starvation. A finishing writer signals the notification objects of all waiting readers. Two vectors, `waitingReaders` and `waitingWriters`, are used as queues for the notification objects of waiting readers and writers. An `InterruptedException` is not handled (Exercise 10).

In contrast to the named condition variables of Section 5.2, it is not possible with this notification scheme to wait in the middle of a synchronized monitor service method for a signal and then continue executing inside the monitor service method at that point after receiving the signal. To avoid deadlock, the thread must leave the synchronized method with a return statement before waiting inside the notification object. After a signal, the thread reenters the monitor via a service method call. The notification code closely resembles the Java implementation of semaphores, shown in Section 5.5.

5.8 Implementing Monitors with Semaphores

Semaphores can be implemented with monitors, as shown in Section 5.5. Conversely, monitors can be implemented with semaphores, making the two synchronization techniques equivalent in power. Monitors, though, as mentioned at the beginning of this chapter, are a higher level concept. The implementation shown here is based on ([1], Section 6.6) for signal-and-continue and ([6], Section 5.4) for signal-and-exit. The description in ([6], Section 5.4) for signal and continue is not correct; the one in ([41], pages 52–53) for signal-and-exit is not correct. See Exercise 17.

Each monitor includes a binary semaphore that permits only one thread at a time to be active in the monitor.

```
semaphore mutex = 1
   ...
service method$i$ {
      P(mutex);
         ...
      V(mutex);
      exit the monitor, i.e., return
}
   ...
```

Each condition variable $cond_i$ is implemented with a semaphore SEM_i and an integer counter $COUNT_i$, both initially zero. The following implements the signal-and-continue signaling discipline.

```
wait($cond_i$):
        $COUNT_i++$;
        V(mutex);
        P($SEM_i$);
        P(mutex);

signal($cond_i$):
        if ($COUNT_i > 0$) { $COUNT_i--$; V($SEM_i$); }
        ...continue in the monitor, perhaps send more signals
```

As mentioned earlier, when a signal is generated a thread waiting at P(mutex) to enter the monitor via a service method call may barge in (as a result of a context switch or faster CPU) and enter the monitor before the signaled thread executes P(mutex). The thread that barged in may change the condition; when the signaled thread's P(mutex) succeeds, the condition may no longer be true. To protect against this, waiting threads should recheck the condition after waking up from a signal and reentering the monitor.

The above is not correct for signal and exit because a signaled thread does not have priority to reenter the monitor. To implement signal and exit, we change the above to

```
wait($cond_i$):
        $COUNT_i++$;
        V(mutex);
        P($SEM_i$);

signal($cond_i$):
        if ($COUNT_i > 0$) { $COUNT_i--$; V($SEM_i$); }
        else V(mutex);
        exit the monitor, i.e., return
```

A signaled thread gets priority to proceed in the monitor over any threads waiting to enter the monitor via a service method call. This subtlety is easily overlooked.

5.8.1 Named Condition Variables for Java

The methods for condition variables in Library Class 3.1, `MyObject`,

```
protected static void wait(ConditionVariable cv) {
   cv.waitCV();
}
protected static void notify(ConditionVariable cv) {
   cv.notifyCV();
}
protected static boolean empty(ConditionVariable cv) {
   return cv.emptyCV();
}
```

and the ideas in the pseudocode of Section 5.8 for implementing monitors with semaphores can be used to complete the implementation of the class `ConditionVariable` in Library Class 5.4 (Exercise 20).

```
public class ConditionVariable { // not yet implemented
   public void waitCV(Object monitor) {}
   public void notifyCV(Object monitor) {}
   public boolean emptyCV(Object monitor) {return true;}
}
```

5.9 Algorithm Animation

Library Class 5.5 is a Java implementation of the XTANGO animator command set described in Section 2.7. This package has many thread synchronization issues. The animation surface is implemented with a Java AWT frame having several buttons along the top and a canvas in the center for drawing. All window events, such as button clicks, are handled in the frame's `actionPerformed` method.

All of the public methods in the `XtangoAnimator` class, such as `coords`, `line`, `circle`, and `move`, create an object and call the `doCommand` method in the frame.

```
// Create a circle with the given radius centered at the
// given position.
public void circle(String id, float xpos, float ypos,
      float radius, Color colorval, int fillval) {
   Icircle c =
      new Icircle(id, xpos, ypos, radius, colorval, fillval);
   af.doCommand(c);
}
```

The `XtangoAnimator` class has a `main` method that tests all of these methods. The objects created in the `XtangoAnimator` public methods are instantiated from the `AnimatorCommand` class. It has two subclasses: `AnimatorAction` for commands like `coords` and `move`, and `AnimatorIcon` for icons to be drawn like lines and circles. `AnimatorShape` is a subclass of `AnimatorIcon` for drawn icons with shapes, like circles, triangles, and rectangles.

An `AnimatorIcon` object has `add` and `draw` methods. The `add` method accesses an `icons` vector of all icons to be drawn during the animation and a hash table that associates the object's identifying string with the object's entry in the vector. After checking the hash table for a duplicate string, the method adds the object to the `icons` vector.

```
protected String id;
...
void add(Hashtable ht, Vector icons) {
   Object old = null;
   synchronized (ht) {
         if (ht.containsKey(id)) {
            System.err.println("duplicate id=" + id);
            return;
         }
         old = ht.put(id, this);
      }
      if (old == null) {
         synchronized (icons) { icons.addElement(this); }
      } else { System.err.println("should not happen!"); }
   }
```

The draw method is called by a thread in the canvas. The method is abstract in the AnimatorIcon class and must be implemented by each particular icon class, such as Iline.

```
   synchronized void draw(AnimatorCanvas ac, Graphics g) {
      g.setColor(colorval);
      if (widthval == XtangoAnimator.THIN) {
         g.drawLine(
            ac.scaleX(position.x), ac.scaleY(position.y),
            ac.scaleX(position.x+size.w),
            ac.scaleY(position.y+size.h));
      } else if (widthval == XtangoAnimator.MEDTHICK) {...}
      else if (widthval == XtangoAnimator.THICK) {...}
   }
```

An AnimatorAction object has a perform method that uses the identifying string of the icon being manipulated to retrieve a reference to that icon from the hash table. For example, to move an icon its position, stored inside the icon object, is changed by the perform method.

```
   void perform(AnimatorCanvas ac, Hashtable ht, Vector icons) {
      AnimatorIcon icon = null;
      synchronized (ht) {
         if ((icon = (AnimatorIcon) ht.get(id)) == null) {
            System.err.println("no such id=" + id);
            return;
         }
      }
      synchronized (icon.position) { icon.position(xpos, ypos); }
   }
```

The frame's doCommand method checks the object passed by the public methods in XtangoAnimator to see if the object is an AnimatorAction or an AnimatorIcon.

```
   void doCommand(AnimatorCommand command) {
      if (XtangoAnimator.debug)
```

```
         System.out.println("doCommand: command " + command);
      if (command instanceof AnimatorAction) {
         ((AnimatorAction) command).perform(ac, ht, icons);
      } else if (command instanceof AnimatorIcon) {
         ((AnimatorIcon) command).add(ht, icons);
      } else {
         System.err.println("doCommand: illegal command");
      }
   // try {
   //   int frameDelay = ac.frameDelay;  Thread.sleep(frameDelay);
   // } catch (InterruptedException e) {}
      synchronized (this) {
         if (singleStep)
            try { wait(); } catch (InterruptedException e) {}
      }
   }
```

If the former, doCommand calls the object's perform method, moving an icon for example; if the latter, doCommand calls the object's add method so the object is drawn on the canvas. The doCommand method is not synchronized, permitting commands to be processed in parallel, such as two threads moving icons at the same time. The sleep(frameDelay) is commented out in the current version to allow a group of commands to be done nearly instantaneously. Invoke

```
   xa.delay(frames);
```

after an animation command if its action should be performed as a distinct step in the algorithm animation.

Single-stepping is implemented with a wait() near the end of doCommand, shown above, and a notifyAll() in the actionPerformed method of the frame.

```
   public void actionPerformed(ActionEvent evt) {
      Object o = evt.getSource();
      if (o instanceof Button) {
         String label = ((Button) o).getLabel();
         if (label.equals("Start")) {...
         } else if (label.equals("Single Step Off")) {
            synchronized (this) { singleStep = true; }
            singleStepButton.setLabel("Single Step On");
         } else if (label.equals("Single Step On")) {
            synchronized (this) {
               singleStep = false; // now that single stepping is
               this.notifyAll();   // off, clear out any waiting
            }                      // threads
            singleStepButton.setLabel("Single Step Off");
         } else if (label.equals("Step")) {
            synchronized (this)
            { if (singleStep) this.notifyAll(); }
         } else if ...
      } ...
   }
```

All drawing is done by a thread in the canvas.

```
   public void run() {
      while (true) {
         try {
            Thread.sleep(frameDelay);
         } catch (InterruptedException e) {}
         repaint();               // tell AWT to call paint()
      }
   }
   public synchronized void paint(Graphics g) {
         // activated by repaint()
      offscreenGraphics.setColor(getBackground());
      offscreenGraphics.fillRect(0, 0, widthCanvas, heightCanvas);
      synchronized (icons) {
         int howMany = icons.size();
         for (int i = 0; i < howMany; i++) { ((AnimatorIcon)
            icons.elementAt(i)).draw(this, offscreenGraphics);
         }
      }
      g.drawImage(offscreenImage, 0, 0, this);
   }
```

After a delay time determined by the magnitude of `frameDelay`, the canvas thread uses the hash table and `icons` vector to call each `AnimatorIcon` object's `draw` method. To avoid flicker, the drawing is done in an off-screen image.

To avoid race conditions and corrupted data structures, the drawing thread and the user threads invoking `doCommand` through the public methods in `XtangoAnimator` must be synchronized so that only one thread at a time accesses the hash table and the `icons` vector. An icon should be drawn before or after but not while any of its attributes is being changed by a command. Look for all occurrences of the keyword `synchronized` in the code!

Summary

The monitor is Java's built-in synchronization primitive, specifically, object locks, synchronized blocks, `wait()`, `notify()`, and `notifyAll()`. Monitors are usually easier and safer to use than semaphores because all shared variables and synchronization constructs are encapsulated into a single class definition. Recall from the previous chapter that related semaphore operations may be widely separated, scattered among many classes. Nonetheless, deadlock is a major concern when using either semaphores or monitors.

In this chapter we defined the monitor, its condition variables, and several signaling disciplines that are in use: signal and exit, signal and continue, and signal and wait. Since Java uses signal and continue, a Java monitor designer must be aware of the behavior of this particular signaling discipline. Barging is possible, more than one signal may be generated by a thread active in a monitor, and waiting threads nearly always need to recheck the monitor condition on which they are waiting each time they reenter the monitor after a signal.

`BinarySemaphore` and `CountingSemaphore`, the binary and counting semaphore classes used extensively in Chapter 4, are implemented as Java monitors. We examined their implementation, deferred from the earlier chapter.

We looked at monitor solutions to the three classic synchronization problems (bounded buffer producer and consumer, dining philosophers, database readers and writers) using signal and exit and signal and continue. The latter two problems were solved with a starvation-free algorithm.

Java synchronized blocks and our explicit binary semaphores are closely related. We defined a class, `Lock`, that combines the features of both.

Each Java monitor has only a single nameless condition variable, in contrast to the conventional definition of a monitor. This complicates the design of some monitor synchronization solutions or makes the resulting code less efficient because a `notifyAll()` must be used instead of a signal targeted to a specific group of waiting threads. Consequently, most Java monitor methods follow this design pattern:

```
  public synchronized type method(...) {
     ...
     notifyAll(); // if any wait conditions altered
     while (!condition)
        try { wait(); } catch (InterruptedException e) {}
     ...
     notifyAll(); // if any wait conditions altered
  }
```

Using notification objects helps somewhat in coping with this situation; however, the code may be convoluted and hard to understand because of nested synchronization blocks inside the monitor. And deadlock is of even more concern. We implemented solutions to the three classic synchronization problems using notification objects.

Monitors and semaphores are of equal power in the theoretical sense that each can be implemented with the other. In this chapter, we showed how to do both.

Several algorithm animations were shown in previous chapters, using the `XtangoAnimation` package. We concluded the chapter with a sketch of how the package is implemented, appropriate at this point because of the many synchronization concerns in the package.

5.10 Exercises

1. **Test and Set.** Write a Java monitor that implements the test-and-set instruction described in Section 3.5.3.

2. **Producers and Consumers.** Class 5.1 is a signal-and-exit monitor for a single producer thread and a single consumer thread. It has two condition variables, `notFull` and `notEmpty`, and is written in a Java-like pseudocode. Analyze the correctness of this monitor if one or more of the following changes are made (there are eight combinations in all): there are multiple producer and consumer threads instead of one of each, the signaling discipline is signal and continue instead of signal and exit, there is one nameless (anonymous) condition variable instead of two.

Class 5.5 is a Java bounded buffer monitor for a single producer thread and a single consumer thread. The signaling discipline is signal and continue and there is one nameless (anonymous) condition variable. Explain exactly why the `wait` is part of a `while` loop instead of an `if` statement. The `notify`s in `deposit` and `fetch` are part of `if` statements so that signals are not wasted, that is, not generated when it is impossible any thread is waiting. On the other hand, using this `if` may mean that a needed signal does not get generated. Is this possible? Why or why not?

Modify Class 5.5 so that it can be used with multiple producer and consumer threads, as was done in Exercise 4.2 for the semaphore version. Think carefully about whether `notify`s should be changed to `notifyAll`s and whether signals should be generated unconditionally instead of inside an `if` statement. Explain your decisions.

3. **Bounded Buffer Fetch with Timeout.** Modify your bounded buffer monitor from the previous exercise (works for multiple producer and consumer threads) so that it has an additional method:

```
public double fetch(long maxMillisecondsWait)
   throws WaitTimeoutException
```

Use the `wait(long ms)` method that blocks at most `ms` milliseconds for a `notify()` or `notifyAll()`. *Be sure to recheck the number of full buffer slots and the elapsed* milliseconds each time the `wait` returns in order to determine which event occurred. Throw an exception if the `wait` timed out. Why can we not return 0 or −1? Explain why we must check the elapsed milliseconds each time the `wait` returns?

4. **Fair Baboons.** Write a monitor for the fair baboons program done in Exercise 4.7. Pick either strict serialization or platooning as your strategy.

5. **Sleeping Barbers.** Write a monitor for the sleeping barbers program done in Exercise 4.4.

6. **Fair Dining Philosophers.** Enhance the starvation detection in the monitor in Class 5.2 or 5.6 for the fair dining philosophers so that the two neighbors must *alternate their eating three times before the one in the middle becomes starving.*

7. **Readers and Writers.** Modify Class 5.7 so that the strict serialization strategy is used to prevent starvation. Modify Class 5.11 in the same way.

8. **Bounded Buffer Notification Monitor.** In Classes 3.8 and 3.19, each buffer slot has an `occupied` field used by the threads for busy waiting and suspend/resume

synchronization, respectively. Write a bounded buffer monitor, valid for multiple producer threads and multiple consumer threads, that uses each buffer slot as a notification object in a similar way. Make `notify()` and `if...wait()` feasible by using nested synchronization blocks so that at most one producer or consumer is waiting on a buffer slot at a time.

```
synchronized (buffer[putIn]) {  // executed by a producer
   if (buffer[putIn].occupied) buffer[putIn].wait();
   buffer[putIn].value = value;
   buffer[putIn].occupied = true;
   buffer[putIn].notify();
}
```

9. **Readers and Writers Notification Monitor.** Class 5.11 uses a convey object for each reader to wait inside if it has to delay reading the database due to a waiting writer. For the platoon approach, it is necessary to use only one such object in which all delayed readers wait. A writer exiting the database does a `notifyAll` in this object to release all waiting readers, if any. Make this modification.

10. `InterruptedException` **in Notification Objects.** If a thread calls some object's `wait` method, the thread blocks and waits until removed from the wait set by a call to the object's `notify` method. Invoking the thread's `interrupt` method also removes the thread from the wait set. The `wait` method then throws an `InterruptedException` object. One way to handle an `InterruptedException` object in a monitor having `wait` inside a `while` loop is to check the condition and `wait` again.

```
while (condition) {
   try { wait(); }
   catch (InterruptedException e) { }
}
```

If the `wait` is part of an `if` statement, this technique is not correct: an `interrupt` has the same effect as a `notify`. Instead, we use a `while` loop in conjunction with the `if` statement.

```
if (condition) {
   while (true) {
      try { wait(); break; }
      catch (InterruptedException e) { continue; }
   }
}
```

The only way a waiting thread proceeds in the monitor is after a `notify`; an `interrupt` causes the thread to `wait` again. After leaving the wait set, due to a `notify` or `interrupt`, the thread must compete with other threads to reacquire the object's lock. The `wait` method returns when the lock is obtained after a `notify`. In the case of an interrupt, the catch block is executed when the lock is obtained. An object's lock must also be acquired to call its notify method.

Suppose that the waiting set of the object contains exactly one thread. A race condition occurs in the second technique above for handling an InterruptedException if some thread tries to lock the object to call notify and, at about the same time, some other thread invokes interrupt. If the interrupt occurs first, then the interrupted waiting thread and the notifying thread compete to obtain the object's lock. If the interrupted thread succeeds, it will execute wait again and be notified. However, if the notifying thread obtains the lock first, the signal is lost because the wait set is empty. Both Library Class 5.1 and Program 5.9 contain code to check for this race condition.

```
// code that waits
   value--;
   if (value < 0) {
      while (true) { // we must be notified not interrupted
         try { wait(); break; }
         catch (InterruptedException e) {
            if (value >= 0) break;
            else continue;
         }
      }
   }
...
// code that notifies
   value++;
   if (value <= 0) notify();
```

The waiting thread examines value to see if it missed a signal while trying to lock the object after an interrupt.

This race condition is not detected in the notification objects of Section 5.7, such as in Program 5.10.

```
synchronized (convey[i]) {
   if (hungryAndGetForks(i)) return;
   else while (true)   // we must be notified not interrupted
      try { convey[i].wait();  break; }
      // notify() after interrupt() race condition ignored
      catch (InterruptedException e) { continue; }
}
```

Correct this deficiency.

11. **Deadlock.** Classes 5.10 and 5.11 use nested synchronization blocks that are not always locked in the same order. Is deadlock possible, as discussed in Section 5.4?

12. **Fraternity Party.** Write a monitor for the fraternity party program done in Exercise 4.8.

13. **Bakery.** Write a monitor for the bakery program done in Exercise 4.3.

14. **Amusement Park Multirider Bumper Cars.** Modify your multiple rider bumper car program from Exercise 4.13 so that the coordinator class is coded as a Java monitor instead of with semaphores; that is, all the methods for condition synchronization are synchronized and use wait(), notify(), and/or notifyAll(). Your monitor will have the following service methods called by the rider and car threads: getInLine(rid), takeAseat(rid), waitCarFull(rid), takeAride(rid), load(cid), and unload(cid).

Remember that Java monitors use signal and continue, so think about the ramifications of this as you design your monitor, in particular, the use of while rather than if with wait() and the use of notifyAll() instead of notify(). Remember that "barging" is possible with signal and continue. Do not use nap within a monitor synchronized method. Do you see why?

Here is one approach to such a monitor class. Think of the monitor as maintaining the state of the system and the synchronized methods as making atomic changes to the state of the system. For example, takeAseat checks to see if there is a car loading (or ready to load). If not, it sets a Boolean variable riderWaiting and then calls wait(). If there is a car loading (or ready to load), it adds the rider to the car and checks if the car is full. If the car is full, it signals with notify() or notifyAll(). Similarly, load checks to see if any riders are waiting. If so, it signals with notify() or notifyAll(). If not, it sets a Boolean variable carReadyToLoad and calls wait(). Do not let two cars load at the same time. Do you see why?

A totally different approach is to use a bounded buffer. Remember to use a bounded buffer that is safe for multiple producers and multiple consumers and that has no semaphores. A rider enters the monitor, puts its identifier into a bounded buffer maintained inside the monitor, and then waits. A car enters the monitor to get enough riders out of the bounded buffer to fill it. If not enough riders are currently in the buffer to fill the car, the car waits. If you use this approach, be careful about nested monitor method calls leading to deadlock, that is, a rider or car calling a synchronized coordinator method that calls a synchronized bounded buffer method. Do you see why?

15. **Haunted House with Multiple Touring Cars.** Modify your multiple car Haunted House program from Exercise 4.15 so that the coordinator class is coded as a Java monitor.

16. **Implementing Semaphores with Java Monitors.** Compare the performance of

```
class AnotherBinarySemaphore {

   private boolean locked = false;
```

```
      public AnotherBinarySemaphore() {}  // constructors
      public AnotherBinarySemaphore(boolean initial) {locked = initial;}
      public AnotherBinarySemaphore(int initial) {
         if (initial < 0 || initial > 1)
            throw new IllegalArgumentException("initial<0 || initial>1");
         locked = (initial == 0);
      }

      public synchronized void P() {
         while (locked) {
            try { wait(); } catch (InterruptedException e) {}
         }
         locked = true;
      }

      public synchronized void V() {
         if (locked) notify();
         locked = false;
      }
   }
```

with Library Class 5.2. Then compare the performance of Library Class 5.2 with Program 5.8.

Modify AnotherBinarySemaphore so that it is AnotherCountingSemaphore, as follows.

```
   class AnotherCountingSemaphore {

      private int value = 0;

      public AnotherCountingSemaphore() {}  // constructors
      public AnotherCountingSemaphore(int initial) {
         if (initial < 0) throw new IllegalArgumentException("initial<0");
         value = initial;
      }

      public synchronized void P() {
         while (value == 0) {
            try { wait(); } catch (InterruptedException e) {}
         }
         value--;
      }

      public synchronized void V() {
         if (value == 0) notify();
         value++;
      }
   }
```

Then compare the performance of the latter with Library Class 5.3, in particular,

discover how barging might lead to an inconsistent semaphore state.

17. **Implementing Monitors with Semaphores.** Section 5.8 shows how monitors are implemented with semaphores. The implementation

```
wait($cond_i$):
        $COUNT_i++$;
        V(mutex);
        P($SEM_i$);
        P(mutex);
        $COUNT_i--$;

signal($cond_i$):
        if ($COUNT_i > 0$) V($SEM_i$);
        ...continue in the monitor, perhaps send more signals
```

shown in ([6], Section 5.4) for signal and continue is not correct. Why? Think about what happens if two successive signals are done on a condition variable on which one thread is waiting.

The description in ([41], pages 52–53) for signal and exit is not correct. Why? Think about what happens if the `up(c)` is done in procedure `leave_with_signal` when there are no threads blocked on the semaphore c.

18. **FIFO Semaphores.** If a thread calls `notify()` and the set of waiting threads is not empty, an arbitrary thread from the set is awakened to reacquire the monitor's lock. Using the notification idea from Section 5.7, modify Library Class 5.1 so that threads blocked inside `P` are released in first-come-first-served order by calls to `V`. Use the `Vector` class to contain the waiting threads' notification objects, as is done in Class 5.11.

19. **Flawed Condition Variables.** Consider the following attempt to implement named condition variables in Java. For each condition, create a `Condition` object whose `wait` and `notify` methods are used in a `Monitor` object for synchronization.

```
public class Condition {
}

public abstract class Monitor {

   protected static void wait(Condition c) {
      synchronized (c)
         { try { c.wait(); } catch (InterruptedException e) {} }
   }

   protected static void signal(Condition c) {
      synchronized (c) { c.notify(); }
   }
}
```

The following extends the Monitor class to implement a bounded buffer. Are there any problems with this approach? Look for race conditions and the potential for deadlock.

```
class BoundedBuffer extends Monitor {

   private int numSlots = 0;
   private double[] buffer = null;
   private int putIn = 0, takeOut = 0, count = 0;
   private Condition elements =
      new Condition(), spaces = new Condition();

   public BoundedBuffer(int numSlots) {
      if (numSlots <= 0)
         throw new IllegalArgumentException("numSlots<=0");
      this.numSlots = numSlots;
      buffer = new double[numSlots];
   }

   public synchronized void deposit(double value) {
      while (count == numSlots) wait(spaces);
      buffer[putIn] = value;
      putIn = (putIn + 1) % numSlots;
      count++;
      if (count == 1) signal(elements);
   }

   public synchronized double fetch() {
      double value;
      while (count == 0) wait(elements);
      value = buffer[takeOut];
      takeOut = (takeOut + 1) % numSlots;
      count--;
      if (count == numSlots-1) signal(spaces);
      return value;
   }

   public static void main(String[] args) {
      BoundedBuffer bb = new BoundedBuffer(5);
      bb.deposit(Math.PI);
      System.out.println("fetched " + bb.fetch() + " from bb");
   }
}
```

How about this version?

```
class BoundedBuffer extends Monitor {

   private int numSlots = 0;
   private double[] buffer = null;
   private int putIn = 0, takeOut = 0, count = 0;
```

```
   private Condition elements =
      new Condition(), spaces = new Condition();

   public BoundedBuffer(int numSlots) {
      if (numSlots <= 0)
         throw new IllegalArgumentException("numSlots<=0");
      this.numSlots = numSlots;
      buffer = new double[numSlots];
   }

   public void deposit(double value) {
      while (true) {
         synchronized (this) {
            if (count < numSlots) {
               buffer[putIn] = value;
               putIn = (putIn + 1) % numSlots;
               count++;
               if (count == 1) signal(elements);
               return;
            }
         }
         wait(spaces);
      }
   }

   public double fetch() {
      double value;
      while (true) {
         synchronized (this) {
            if (count > 0) {
               value = buffer[takeOut];
               takeOut = (takeOut + 1) % numSlots;
               count--;
               if (count == numSlots-1) signal(spaces);
               return value;
            }
         }
         wait(elements);
      }
   }

   public static void main(String[] args) {
      BoundedBuffer bb = new BoundedBuffer(5);
      bb.deposit(Math.PI);
      System.out.println("fetched " + bb.fetch() + " from bb");
   }
}
```

20. **Named Condition Variables for Java.** Complete the implementation of the class ConditionVariable in Library Class 5.4 so that it can be used in Java monitors (classes with synchronized methods). See Section 5.8.1. Using the Vector

class, associate a queue with each condition variable; waitCV adds the thread to the condition variable's queue before doing a wait() inside the monitor, which is why this is passed to waitCV in MyObject and why waitCV is not static. When the waiting thread is awakened, it checks to see if it has reached the head of the queue; if not, it waits again. A call to notifyCV invokes notifyAll() inside the monitor; all the waiting threads check to see if they are at the head of the queue. Compare the performance of the technique outlined here with the one in Section 5.7.

The preceding implements signal and continue. Is it possible to implement signal and exit? If so, do it. If not, why not?

Why can we not just call the condition variable object's wait() instead of the monitor's wait(cv)?

21. **More Named Condition Variables.** The condition variable class CondVar, adapted from [33], is designed for use in semaphore programs and resembles a notification object.

```
import Utilities.*;
import Synchronization.*;

public class CondVar extends MyObject {

   public synchronized void wait(BinarySemaphore mutex) {
         V(mutex);              // release the monitor
         try { wait(); }        // atomically before waiting
         catch (InterruptedException e) {}
         P(mutex);              // reacquire the monitor
   }

   public synchronized void signal() {
      notify();                 // release a waiting thread
   }

   public synchronized void signalAll() {
      notifyAll();              // release all waiting threads
   }
}
```

The following example creates a condition variable from this class.

```
import Utilities.*;
import Synchronization.*;

class Allocator extends MyObject {

   private BinarySemaphore mutex =
      new BinarySemaphore(1);
   private int available = 0;
   private CondVar cv = new CondVar();
```

```
   public Allocator(int initial) {
      if (initial <= 0)
         throw new IllegalArgumentException();
      available = initial;
   }

   public void want(int w) {
      P(mutex);         // acquire the monitor
      while (w > available) cv.wait(mutex);
      available -= w;
      V(mutex);         // release the monitor
   }

   public void done(int d) {
      P(mutex);         // acquire the monitor
      available += d;
      cv.signalAll();
      V(mutex);         // release the monitor
   }
}
```

The Allocator class is functionally equivalent to a monitor, using an explicit semaphore mutex instead of synchronized methods for mutual exclusion. The semaphore is passed as a parameter to the condition variable's wait method. Compare the performance of this technique with the ones in Sections 5.7, 5.8, and 5.8.1. Will these condition variables work inside a Java monitor (a class all of whose methods are synchronized)? Why?

22. **Semaphore with a Timeout.** Modify Library Class 5.1 so that the P method is overloaded with one having an argument of type integer that represents a millisecond timeout option. If the P(*ms*) call is still blocked after the timeout period expires, throw an exception.

23. **Four-of-a-Kind Game.** ([25], page 107) Write a monitor that is used in a multithreaded Java program simulating the four-of-a-kind game. The game is played with a deck of cards containing six suits of four cards each. Four players sit around a table, each holding four cards. There are piles of cards between each pair of players, alternating with them around the table. At the start of the game, each pile contains two cards. Each player repeatedly discards a card into the left-hand pile and draws a card from the right-hand pile. The first player to obtain four-of-a-kind wins the game.

24. **Locks.** The lock on a Java object acting as a monitor is obtained by entering a synchronized block on that object or entering a synchronized method in the object. The lock is released when the block or method is exited. Java allows the same thread to lock a monitor object multiple times. This occurs in two nested synchronized blocks on the same object or when a synchronized method in an object invokes another synchronized method in the same object. The Java virtual machine counts the number of times the lock is locked in this fashion and only releases the lock when

the same number of unlocking events occurs. Modify the `Lock` class in Program 5.8 so that it behaves this way. Every time the thread currently owning the lock calls `lock`, increment a counter; every time it calls `unlock`, decrement the counter; if the counter reaches zero, release the lock.

Add a `tryLock` method that obtains the lock if currently free but throws a `WouldBlockException` object if the lock is not free.

Modify the `Lock` class in Program 5.8 so that it does not allow barging. This occurs if a thread calls `lock` after other threads have entered their `wait()` loops and gets the lock before the waiting threads. Also make the lock fair by granting the lock in a first-come-first-served order. Add a `Vector` to contain references to the threads waiting to obtain the lock. Instead of `notifyAll()`, use `notify()` and the notification technique of Section 5.7.

25. **Time Slicing Scheduler Class.** Modify Library Class 3.2, the `Scheduler` class that introduces time slicing, so that at most one object can be instantiated from this class during a program execution. Add a static variable to the class that counts instantiations. Call a static method in the class constructor that increments this count. Does this method need to be `synchronized`?

26. **Algorithm Animation.** Find all remaining race conditions, if any, in Library Class 5.5 and fix them. Give a specific sequence of events that leads to each race condition you find.

Add an `Iimage` icon class to Library Class 5.5 that associates a string `id` with an image stored in a GIF or JPEG file.

Add a `Cgroup` command class that associates a string `id` with a group (an array) of other `id`s so that commands such as move and change color can be applied to the whole group.

Add a `CnewId` command class that gives an icon a new string `id`.

Add a `Cwidth` class that changes the `widthval` of a line icon.

Add a `Crotate` class that rotates an icon.

Add an arrow option to the `Iline` class.

The `vis`, `raise`, and `lower` methods deal with viewing planes and the order icons are drawn on top of one another. Add a `Chide` class and a `hide` method that toggles the visibility of an icon in the sense of drawing the icon or not, regardless of its viewing plane. Add a Boolean field to the `AnimatorIcon` class; if it is false, the drawing loop in the canvas thread skips the icon.

Chapter 6

Message Passing and the Rendezvous

The previous two chapters show how a collection of threads sharing one address space in a uniprocessor or shared-memory multiprocessor uses semaphores or monitors for mutual exclusion and event synchronization. Even though two threads in two different address spaces cannot interfere with each other's variables and are therefore not subject to race conditions, they may still need to synchronize their execution or communicate data. Suppose in the bounded buffer producer and consumer problem, the producer and consumer threads are in separate address spaces. Items built by the producer need to be delivered (communication) to the consumer. The consumer needs to block if the buffer is empty, and the producer needs to block if the buffer is full (synchronization). As another example, suppose a client thread executing on a diskless workstation wants to read some data from a file on a disk attached to a file server machine. The client must somehow send its request to the file server and then block until the data are returned.

However, two threads in two different address spaces cannot share semaphores or monitors for synchronization or communication. For example, threads spread across a distributed system, such as a collection of machines, each with its own CPU and memory, connected together on a LAN, cannot share semaphores or monitors. Computer science researchers are devising *distributed shared memory* (DSM) systems ([42], Chapter 6), so one day such sharing might be possible; however, DSM systems are not yet practical. Therefore, we must use another technique for thread communication and synchronization in distributed systems. This chapter describes message passing and two tools implemented with message passing, the rendezvous and the remote procedure call. Message passing may be used by threads within one address space as an alternative to semaphores and monitors.

We examine several varieties of message passing: synchronous, asynchronous, capacity controlled, and conditional. Message types supported are integer numbers (`int`), floating-point numbers (`double`), and objects. Within one address space, a pipe holds sent but not yet received messages. Between address spaces, particularly between machines, a socket is used. Two message passing applications are presented, distributed mutual exclusion and the distributed dining philosophers. We next see