Chapter 3

Synchronization Primitives

3.1 Introduction

All of our previous solutions to the mutual exclusion problem were wasteful in one regard. If a process is unable to enter the critical section, it repeatedly checks for the entry condition to be true. While a process is doing this, no useful work is accomplished. This way of waiting is called *busy wait*. Instead of checking the entry condition repeatedly, if the process checked the condition only when it could have become true, it would not waste CPU cycles. Accomplishing this requires support from the operating system.

In this chapter we introduce synchronization primitives that avoid busy wait. Synchronization primitives are used for mutual exclusion as well as to provide order between various operations by different threads. Although there are many types of synchronization constructs in various programming languages, two of them are most prevalent: semaphores and monitors. We discuss these constructs in this chapter.

3.2 Semaphores

Dijkstra proposed the concept of *semaphore* that solves the problem of busy wait. A semaphore has two fields, its `value` and a `queue` of blocked processes, and two operations associated with it — $P()$ and $V()$. The semantics of a binary semaphore is shown in Figure 3.1. The `value` of a semaphore (or a binary semaphore) can be only *false* or *true*. The `queue` of blocked processes is initially empty and a process may add itself to the queue when it makes a call to $P()$. When a process calls $P()$ and `value` is *true*, then the value of the semaphore becomes *false*. However, if the value of the semaphore is *false*, then the process gets blocked at line 7 until it becomes *true*. The invocation of `Util.myWait()` at line 8 achieves this. The class `Util` is shown in the appendix, but for now simply assume that this call inserts the caller process into the queue of blocked processes.

When the value becomes *true*, the process can make it *false* at line 9 and return from $P()$. The call to $V()$ makes the value *true* and also notifies a process if the queue of processes sleeping on that semaphore is nonempty.

Now, mutual exclusion is almost trivial to implement:

```
BinarySemaphore mutex = new BinarySemaphore(true);
mutex.P();
criticalSection();
mutex.V();
```

```
1  public class BinarySemaphore {
2      boolean value;
3      public BinarySemaphore(boolean initValue) {
4          value = initValue;
5      }
6      public synchronized void P() {
7          while (value == false)
8              Util.myWait(this);// in the queue of blocked processes
9          value = false;
10     }
11     public synchronized void V() {
12         value = true;
13         notify();
14     }
15 }
```

Figure 3.1: Binary semaphore

Another variant of semaphore allows it to take arbitrary integer as its value. These semaphores are called *counting semaphores*. Their semantics is shown in Figure 3.2.

```
public class CountingSemaphore {
    int value;
    public CountingSemaphore(int initValue) {
        value = initValue;
    }
    public synchronized void P() {
        while (value == 0) Util.myWait(this);
        value--;

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

Figure 3.2: Counting semaphore

Semaphores can be used to solve a wide variety of synchronization problems. Note that Java does not provide semaphores as basic language construct, but they can easily be implemented in Java using the idea of `monitors`, which we will cover later. For now we simply assume that semaphores are available to us and solve synchronization problems using them.

3.2.1 The Producer-Consumer Problem

We first consider the producer-consumer problem. In this problem, there is a shared buffer between two processes called the *producer* and the *consumer*. The producer produces items that are *deposited* in the buffer and the consumer *fetches* items from the buffer and consumes them. Since the buffer is shared, each process must access the buffer in a mutually exclusive fashion. We use an array of `Object` of size `size` as our buffer. The buffer has two pointers, `inBuf` and `outBuf`, which point to the indices in the array for depositing an item and fetching an item, respectively. The variable `count` keeps track of the number of

items currently in the buffer. Figure 3.3 shows the buffer as a circular array in which `inBuf` and `outBuf` are incremented modulo `size` to keep track of the slots for depositing and fetching items.

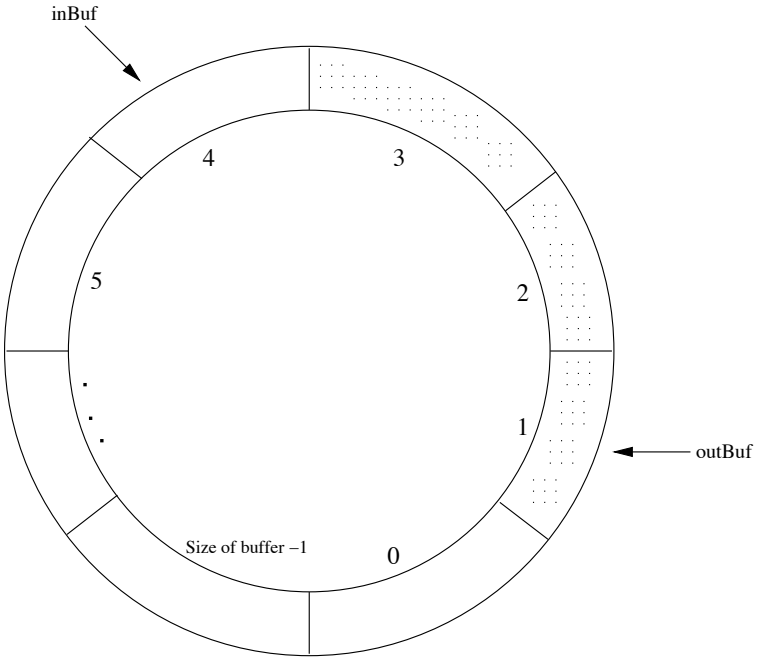

Figure 3.3: A shared buffer implemented with a circular array

In this problem, we see that besides mutual exclusion, there are two additional synchronization constraints that need to be satisfied:

1. The consumer should not fetch any item from an empty buffer.

2. The producer should not deposit any item in the buffer if it is full. The buffer can become full if the producer is producing items at a greater rate than the rate at which the items are consumed by the consumer.

Such form of synchronization is called *conditional synchronization*. It requires a process to wait for some condition to become true (such as the buffer to become nonempty) before continuing its operations. The class `BoundedBuffer` is shown in Figure 3.4. It uses `mutex` semaphore to ensure that all shared variables are accessed in mutually exclusive fashion. The counting semaphore `isFull` is used for making a producer wait in case the buffer is full, and the semaphore `isEmpty` is used to make a consumer wait when the buffer is empty.

In the method `deposit`, line 10 checks whether the buffer is full. If it is, the process making call waits using the semaphore `isFull`. Note that this semaphore has been initialized to the value `size`, and therefore in absence of a consumer, first `size` calls to `isFull.P()` do not block. At this point, the buffer would be full and any call to `isFull.P()` will block. If the call to `isFull.P()` does not block, then we enter the critical section to access the shared buffer. The call `mutex.P()` at line 11 serves as entry to the critical section, and `mutex.V()` serves as the exit from the critical section. Once inside the critical section, we deposit the value in `buffer` using the pointer `inBuf` at line 12 (see Figure 3.4). Line 15 makes a call to `isEmpty.V()` to wake up any consumer that may be waiting because the buffer was empty. The method `fetch` is dual of the method `deposit`.

The class `BoundedBuffer` can be exercised through the producer-consumer program shown in Figure

```
1  class BoundedBuffer {
2      final int size = 10;
3      Object[] buffer = new Object[size];
4      int inBuf = 0, outBuf = 0;
5      BinarySemaphore mutex = new BinarySemaphore(true);
6      CountingSemaphore isEmpty = new CountingSemaphore(0);
7      CountingSemaphore isFull = new CountingSemaphore(size);
8
9      public void deposit(Object value) {
10         isFull.P();// wait if buffer is full
11         mutex.P(); // ensures mutual exclusion
12         buffer[inBuf] = value; // update the buffer
13         inBuf = (inBuf + 1) % size;
14         mutex.V();
15         isEmpty.V();  // notify any waiting consumer
16     }
17     public Object fetch() {
18         Object value;
19         isEmpty.P(); // wait if buffer is empty
20         mutex.P();  // ensures mutual exclusion
21         value = buffer[outBuf]; //read from buffer
22         outBuf = (outBuf + 1) % size;
23         mutex.V();
24         isFull.V(); // notify any waiting producer
25         return value;
26     }
27 }
```

Figure 3.4: Bounded buffer using semaphores

3.5. This program starts a `Producer` thread and a `Consumer` thread, repeatedly making calls to `deposit` and `fetch`, respectively.

3.2.2 The Reader-Writer Problem

Next we show the solution to the reader-writer problem. This problem requires us to design a protocol to coordinate access to a shared database. The requirements are as follows:

1. *No read-write conflict:* The protocol should ensure that a reader and a writer do not access the database concurrently.

2. *No write-write conflict:* The protocol should ensure that two writers do not access the database concurrently.

Further, we would like multiple readers to be able to access the database concurrently. A solution using semaphores is shown in Figure 3.6. We assume that the readers follow the protocol that they call `startRead` before reading the database and call `endRead` after finishing the read. Writers follow a similar protocol. We use the `wlock` semaphore to ensure that either there is a single writer accessing the database or only readers are accessing it. To count the number of readers accessing the database, we use the variable `numReaders`.

The methods `startWrite` and `endWrite` are quite simple. Any writer that wants to use the database locks it using `wlock.P()`. If the database is not locked, this writer gets the access. Now no other reader or writer can access the database until this writer releases the lock using `endWrite()`.

Now let us look at the `startRead` and the `endRead` methods. In `startRead`, a reader first increments `numReaders`. If it is the first reader (`numReaders` equals 1), then it needs to lock the database; otherwise,

```
import java.util.Random;
class Producer implements Runnable {
    BoundedBuffer b = null;
    public Producer(BoundedBuffer initb) {
        b = initb;
        new Thread(this).start();
    }
    public void run() {
        Double item;
        Random r = new Random();
        while (true) {
            item = r.nextDouble();
            System.out.println("produced item " + item);
            b.deposit(item);
            Util.mySleep(200);
        }
    }
}
class Consumer implements Runnable {
    BoundedBuffer b = null;
    public Consumer(BoundedBuffer initb) {
        b = initb;
        new Thread(this).start();
    }
    public void run() {
        Double item;
        while (true) {
            item = (Double) b.fetch();
            System.out.println("fetched item " + item);
             Util.mySleep(50);
        }
    }
}
class ProducerConsumer {
    public static void main(String[] args) {
        BoundedBuffer buffer = new BoundedBuffer();
        Producer producer = new Producer(buffer);
        Consumer consumer = new Consumer(buffer);
    }
}
```

Figure 3.5: Producer-consumer algorithm using semaphores

there are already other readers accessing the database and this reader can also start using it. In `endRead`, the variable `numReaders` is decremented and the last reader to leave the database unlocks it using the call `wlock.V()`.

This protocol has the disadvantage that a writer may starve in the presence of continuously arriving readers. A starvation-free solution to the reader-writer problem is left as an exercise.

```
class ReaderWriter {
    int numReaders = 0;
    BinarySemaphore mutex = new BinarySemaphore(true);
    BinarySemaphore wlock = new BinarySemaphore(true);
    public void startRead() {
        mutex.P();
        numReaders++;
        if (numReaders == 1) wlock.P();
        mutex.V();
    }
    public void endRead() {
        mutex.P();
        numReaders--;
        if (numReaders == 0) wlock.V();
        mutex.V();
    }
    public void startWrite() {
        wlock.P();
    }
    public void endWrite() {
        wlock.V();
    }
}
```

Figure 3.6: Reader-writer algorithm using semaphores

3.2.3 The Dining Philosopher Problem

This problem, first posed and solved by Dijkstra, is useful in bringing out issues associated with concurrent programming and symmetry. The dining problem consists of multiple philosophers who spend their time thinking and eating spaghetti. However, a philosopher requires shared resources, such as forks, to eat spaghetti (see Figure 3.7). We are required to devise a protocol to coordinate access to the shared resources. A computer-minded reader may substitute processes for philosophers and files for forks. The task of eating would then correspond to an operation that requires access to shared files.

Let us first model the process of a philosopher. The class `Philosopher` is shown in Figure 3.8. Each philosopher P_i repeatedly cycles through the following states — *thinking, hungry*, and *eating*. To eat, a philosopher requires resources (forks) for which it makes call to `acquire(i)`. Thus, the protocol to acquire resources is abstracted as an interface `Resource` shown in Figure 3.9.

The first attempt to solve this problem is shown in Figure 3.10. It uses a binary semaphore for each of the forks. To acquire the resources for eating, a philosopher i grabs the fork on its left by using `fork[i].P()` at line 12, and the fork on the right by using `fork[(i+1) % n].P()` at line 13. To release the resources, the philosopher invokes `V()` on both the forks at lines 16 and 17.

This attempt illustrates the dangers of *symmetry* in a distributed system. This protocol can result in deadlock when each philosopher is able to grab its left fork and then waits for its right neighbor to release its fork.

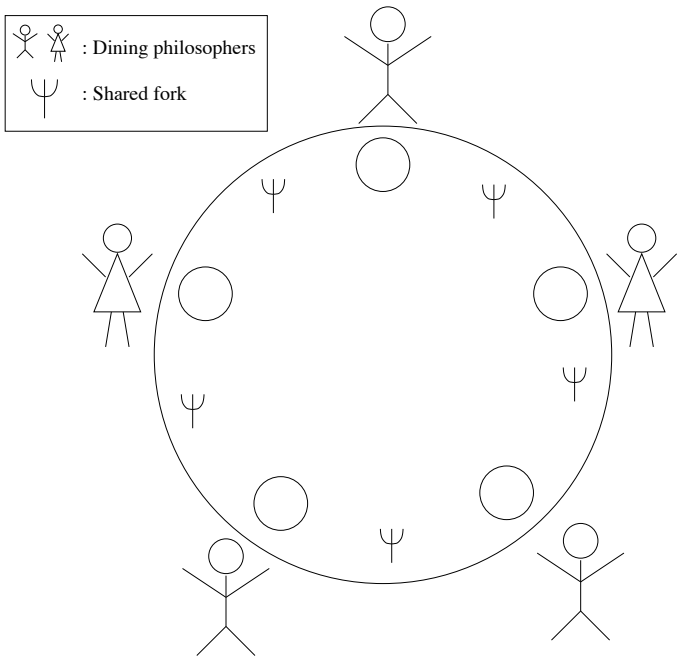

Figure 3.7: The dining philosopher problem

```
class Philosopher implements Runnable {
    int id = 0;
    Resource r = null;
    public Philosopher(int initId, Resource initr) {
        id = initId;
        r = initr;
        new Thread(this).start();
    }
    public void run() {
        while (true) {
            try {
                System.out.println("Phil " + id + " thinking");
                Thread.sleep(30);
                System.out.println("Phil " + id + " hungry");
                r.acquire(id);
                System.out.println("Phil " + id + " eating");
                Thread.sleep(40);
                r.release(id);
            } catch (InterruptedException e) {
                return;
            }
        }
    }
}
```

Figure 3.8: Dining Philosopher

```
interface Resource {
    public void acquire(int i);
    public void release(int i);
}
```

Figure 3.9: `Resource` Interface

```
1  class DiningPhilosopher implements Resource {
2      int n = 0;
3      BinarySemaphore[] fork = null;
4      public DiningPhilosopher(int initN) {
5          n = initN;
6          fork = new BinarySemaphore[n];
7          for (int i = 0; i < n; i++) {
8              fork[i] = new BinarySemaphore(true);
9          }
10     }
11     public void acquire(int i) {
12         fork[i].P();
13         fork[(i + 1) % n].P();
14     }
15     public void release(int i) {
16         fork[i].V();
17         fork[(i + 1) % n].V();
18     }
19     public static void main(String[] args) {
20         DiningPhilosopher dp = new DiningPhilosopher(5);
21         for (int i = 0; i < 5; i++)
22             new Philosopher(i, dp);
23     }
24 }
```

Figure 3.10: Dining philosopher using semaphores

There are many ways that one can extend the solution to ensure freedom from deadlock. For example:

1. We can introduce asymmetry by requiring one of the philosophers to grab forks in a different order (i.e., the right fork followed by the left fork instead of vice versa).

2. We can require philosophers to grab both the forks at the same time.

3. Assume that a philosopher has to stand before grabbing any fork. Allow at most four philosophers to be standing at any given time.

It is left as an exercise for the reader to design a protocol that is free from deadlocks.

The dining philosopher problem also illustrates the distinction between deadlock freedom and starvation freedom. Assume that we require a philosopher to grab both the forks at the same time. Although this eliminates deadlock, we still have the problem of a philosopher being starved because its neighbors continuously alternate in eating. The reader is invited to come up with a solution that is free from deadlock as well as starvation.

3.3 Monitors

The *Monitor* is a high-level object-oriented construct for synchronization in concurrent programming. A monitor can be viewed as a `class` that can be used in concurrent programs. As any `class`, a monitor has data variables and methods to manipulate that data. Because multiple threads can access the shared data at the same time, monitors support the notion of *entry* methods to guarantee mutual exclusion. It is guaranteed that at most one thread can be executing in any entry method at any time. Sometimes the phrase "thread t is inside the monitor" is used to denote that thread t is executing an entry method. It is clear that at most one thread can be in the monitor at any time. Thus associated with every monitor object is a queue of threads that are waiting to enter the monitor.

As we have seen before, concurrent programs also require *conditional synchronization* when a thread must wait for a certain condition to become true. To address conditional synchronization, the monitor construct supports the notion of *condition variables*. A condition variable has two operations defined on it: *wait* and *notify* (also called a *signal*). For any condition variable x, any thread, say, t_1, that makes a call to $x.wait()$ is blocked and put into a queue associated with x. When another thread, say, t_2, makes a call to $x.notify()$, if the queue associated with x is nonempty, a thread is removed from the queue and inserted into the queue of threads that are eligible to run. Since at most one thread can be in the monitor, this immediately poses a problem: which thread should continue after the notify operation—the one that called the $notify$ method or the thread that was waiting. There are two possible answers:

1. One of the threads that was waiting on the condition variable continues execution. Monitors that follow this rule are called *Hoare* monitors (or, *signal-and-wait* monitors).

2. The thread that made the notify call continues its execution. When this thread goes out of the monitor, then other threads can enter the monitor. This is the semantics followed in Java and is called *signal-and-continue*.

One advantage of Hoare's monitor is that the thread that was notified on the condition starts its execution without intervention of any other thread. Therefore, the state in which this thread starts executing is the same as when the $notify$ was issued. On waking up, it can assume that the condition is true. Therefore, using Hoare's mointor, a thread's code may be

```
if (!B) x.wait();
```

Assuming that t_2 notifies only when B is true, we know that t_1 can assume B on waking up. In Java-style monitor, even though t_2 issues the $notify$, it continues its execution. Therefore, when t_1 gets its turn to execute, the condition B may not be true any more. Hence, when using Java, the threads usually wait for the condition as

```
while (!B) x.wait();
```

The thread t_1 can take a `notify()` only as a hint that B may be true. Therefore, it explicitly needs to check for truthness of B when it wakes up. If B is actually false, it issues the `wait()` call again.

In Java, we specify an object to be a monitor by using the keyword `synchronized` with its methods. To get conditional synchronization, Java provides

1. `wait()`: which inserts the thread in the wait queue. For simplicity, we use `Util.myWait()` instead of `wait()` in Java. The only difference is that `myWait` catches the `InterruptedException`.

2. `notify()`: which wakes up a thread in the wait queue.

3. `notifyAll()`: which wakes up all the threads in the wait queue.

In Java, by default, associated with each object is a single `wait` queue for conditions. This is sufficient for most programming needs. A pictorial representation of a Java monitor is shown in Figure 3.11. There are two queues associated with an object—a queue of threads waiting for the lock associated with the monitor and another queue of threads waiting for some condition to become true.

If one needs multiple condition queues, then Java provides ReenetrantLocks which can have multiple conditions variables associated with them. We discuss ReentrantLocks in Section 3.3.2.

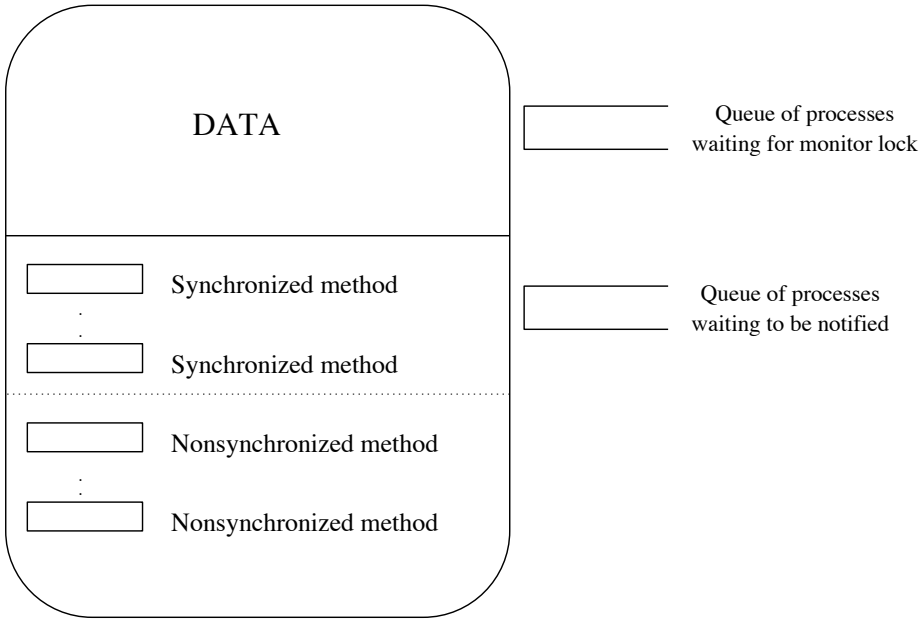

Figure 3.11: A pictorial view of a Java monitor

Let us solve some synchronization problems with Java monitors. We first look at the producer-consumer problem. The `BoundedBufferMonitor` shown in Figure 3.12 has two entry methods: `deposit` and `fetch`. This means that if a thread is executing the method `deposit` or `fetch`, then no other thread can execute `deposit` or `fetch`. The `synchronized` keyword at lines 5 and 14 allows mutual exclusion in access of

shared variables and corresponds to acquiring the *monitor lock*. Let us now look at the method `deposit`. At line 6, if the buffer is full, (i.e., `count` is equal to `size`), then the thread that called `deposit` must wait for a slot in the `buffer` to be consumed. Therefore, it invokes the method `myWait()`. When a thread waits for the condition, it goes in a *queue* waiting to be notified by some other thread. It also has to release the monitor lock so that other threads can enter the monitor and make the condition on which this thread is waiting true. When this thread is notified, it has to acquire the monitor lock again before continuing its execution.

Now assume that the condition in the `while` statement at line 6 is false. Then the `value` can be deposited in the buffer. The variable `inBuf` points to the tail of the circular buffer. It is advanced after the insertion at line 9 and the `count` of the number of items in the buffer is incremented at line 10. We are not really done yet. While designing a monitor, one also needs to ensure that if some thread may be waiting for a condition that may have become true, then that thread must be notified. In this case, a consumer thread may be waiting in the method `fetch` for some item to become available. Therefore, if `count` is 1, we notify any waiting thread at line 12.

The method `fetch` is very similar to `deposit`.

```
1  class BoundedBufferMonitor {
2      final int size = 10;
3      Object[] buffer = new Object[size];
4      int inBuf = 0, outBuf = 0, count = 0;
5      public synchronized void deposit(Object value) {
6          while (count == size) // buffer full
7              Util.myWait(this);
8          buffer[inBuf] = value;
9          inBuf = (inBuf + 1) % size;
10         count++;
11         if (count == 1) // items available for fetch
12             notify();
13     }
14     public synchronized Object fetch() {
15         Object value;
16         while (count == 0) // buffer empty
17             Util.myWait(this);
18         value = buffer[outBuf];
19         outBuf = (outBuf + 1) % size;
20         count--;
21         if (count == size - 1) // empty slots available
22             notify();
23         return value;
24     }
25 }
```

Figure 3.12: Bounded buffer monitor for one producer and one consumer

The class `BoundedBufferMonitor` works correctly only when there is exactly one producer and one consumer. Can you determine why the code does not work for multiple producers and multiple consumers? For simplicity, consider the case when `size` equals 1 and *count* equals 0. Suppose that two consumer threads C_1 and C_2 call the method `fetch()`. They are now waiting for producers. Now suppose that two producers P_1 and P_2 arrive. When P_1 leaves, it notifies consumers. However, before C_1 enters the monitor, P_2 enters the monitor and also starts waiting because the buffer is full. Now C_1 enters the monitor. At this point, there is one consumer C_2 and one producer P_2 waiting for their conditions to become true. When C_1 executes `notify()`, it is poissible that the thread C_2 gets the notification. When thread C_2 enters the monitor, it finds *count* equal to 0 and goes to sleep again. At this point, we have both a producer and a

consumer thread waiting for notifications. The problem is that the notification from thread C_1 was really meant for P_2. However, since `notify()` wakes up a single thread, it ended up notifying the wrong thread. One way to solve this problem is to use `notifyAll()` and remove the `if` check before the notification. A solution for the bounded buffer in presence of multiple producers and consumers with this approach is given in Figure 3.13. Another method is based on using `Condition` variables discussed in Section 3.3.2.

```
1  class MultiBoundedBufferMonitor {
2      final int size = 10;
3      double[] buffer = new double[size];
4      int inBuf = 0, outBuf = 0, count = 0;
5      public synchronized void deposit(double value) {
6          while (count == size) // buffer full
7              Util.myWait(this);
8          buffer[inBuf] = value;
9          inBuf = (inBuf + 1) % size;
10         count++;
11         notifyAll();
12     }
13     public synchronized double fetch() {
14         double value;
15         while (count == 0) // buffer empty
16             Util.myWait(this);
17         value = buffer[outBuf];
18         outBuf = (outBuf + 1) % size;
19         count--;
20         notifyAll();
21         return value;
22     }
23 }
```

Figure 3.13: Bounded buffer monitor for multiple producers and consumers

Now let us revisit the dining philosophers problem. In the solution shown in Figure 3.14, a philosopher i uses the method `test` at line 18 to determine if any of neighboring philosophers is eating. If not, then this philosopher can start eating. Otherwise the philosopher must wait for the condition `(state[i] == eating)` at line 19 to begin eating. This condition can become true when one of the neighboring philosophers finishes eating. After eating, the philosopher invokes the method `release` to check at lines 24 and 25, whether the left neighbor or the right neighbor can now eat. If any of them can eat, this philosopher wakes up all the waiting philosophers by invoking `notifyAll()` at line 32. This solution guarantees mutual exclusion of neighboring philosophers and is also free from deadlock. However, it does not guarantee freedom from starvation. The reader should devise a protocol for starvation freedom.

3.3.1 Implementing Monitors Using Semaphores

Binary semaphores can easily be implemented on top of Java monitor. Figure 3.1 gives one such implementation. Is the converse true? We now show that any concurrent program that is written using monitors can also be written using binary semaphores. There are two aspects of monitors: mutual exclusion and conditional synchronization. Providing mutual exclusion is simple. We keep a `mutex` binary semaphore initialized as `true` for each object. Before any synchronized method, we use `mutex.P()` to acquire the monitor lock, and at exit use `mutex.V()` to release the monitor lock. We now describe how to implement `wait()` and `notify()` using binary semaphores. We will use another binary semaphore `cond` initialized as `false` to implement conditional synchronization. When a thread invokes `wait()`, it must first release the monitor lock, and then wait for notification to evaluate the condition again. Once it receives the

```
1  class DiningMonitor implements Resource {
2      int n = 0;
3      int state[] = null;
4      static final int thinking = 0, hungry = 1, eating = 2;
5      public DiningMonitor(int initN) {
6          n = initN;
7          state = new int[n];
8          for (int i = 0; i < n; i++) state[i] = thinking;
9      }
10     int left(int i) {
11         return (n + i - 1) % n;
12     }
13     int right(int i) {
14         return (i + 1) % n;
15     }
16     public synchronized void acquire(int i) {
17         state[i] = hungry;
18         test(i);
19         while (state[i] != eating)
20             Util.myWait(this);
21     }
22     public synchronized void release(int i) {
23         state[i] = thinking;
24         test(left(i));
25         test(right(i));
26     }
27     void test(int i) {
28         if ((state[left(i)] != eating) &&
29         (state[i] == hungry) &&
30         (state[right(i)] != eating)) {
31             state[i] = eating;
32             notifyAll();
33         }
34     }
35     public static void main(String[] args) {
36         DiningMonitor dm = new DiningMonitor(5);
37         for (int i = 0; i < 5; i++)
38             new Philosopher(i, dm);
39     }
40 }
```

Figure 3.14: Dining philosopher using monitors

notification, it must first acquire the monitor lock. Hence `wait()` is implemented as:
`mutex.V();` // release the monitor lock
`cond.P();` // wait on `cond`
`mutex.P();` // reacquire the monitor lock.

The construct `notify()` would be implemented as `cond.V()`.

3.3.2 Reenetrant Locks and Condition Variables

In the multiple producer-consumer example, conditional synchronization would have been easier if a producer could signal only the consumers and vice-versa. We implement this idea by using the notion of condition variables. In Java, instead of using `synchronized` keywork to protect the monitor, we use explicit lock class called `ReentrantLock`. Any ReentrantLock has two methods associated with it `lock()` and `unlock()`. The qualifier `Reentrant` indicates that if a thread that already has the ReentrantLock, calls the method `lock()`, then it is not blocked and can reenter the critical section. This feature is useful when a thread accesses an object using nested methods. We can associate any number of conditions with a ReentrantLock by invoking the method `newCondition()` of a ReentrantLock. For example, the following code snippet declares a ReentrantLock and associates two condition variables with it.

```
Lock mutex = new ReentrantLock();
Condition notFull = mutex.newCondition();
Condition notEmpty = mutex.newCondition();
```

Associated with each condition variable are three important methods, `await(), signal() and signalAll()`. Now the programmer has flexibility to signal threads waiting for some specific condition.

Fig. 3.15 gives a solution for multiple producer consumer problem using ReentrantLocks and Conditions.

Note the use of `lock()` and `unlock()` in this example. The `unlock()` method is used in the `finally` clause so that the lock is released even when some exception is thrown.

3.4 Other Examples

In this section we give another example of concurrent programming in Java. Figure 3.16 shows a thread-safe implementation of a queue that is based on a linked list. The class `Node` declared at line 2 contains a String as data and the reference to the next node in the linked list. To `enqueue` data, we first create a `temp` node at line 8. This node is inserted at the `tail`. If the linked list is empty, this is the only node in the linked list and both `head` and `tail` are made to point to this node at lines 11–13. To `dequeue` a node, a thread must wait at line 22 if `head` is null (the linked list is empty). Otherwise, the data in the `head` node is returned and `head` is moved to the `next` node.

As mentioned earlier, whenever a thread needs to execute a synchronized method, it needs to get the monitor lock. The keyword `synchronized` can also be used with any statement as `synchronized (expr) statement`. The expression `expr` must result in a reference to an object on evaluation. The semantics of the above construct is that the `statement` can be executed only when the thread has the lock for the object given by the `expr`. Thus a synchronized method

```
public synchronized void method() {
    body();
}
```

can simply be viewed as a short form for

```
1  import java.util.concurrent.locks.*;
2
3  class MBoundedBufferMonitor {
4     final int size = 10;
5     final ReentrantLock monitorLock = new ReentrantLock();
6     final Condition notFull  = monitorLock.newCondition();
7     final Condition notEmpty = monitorLock.newCondition();
8
9     final Object[] buffer = new Object[size];
10    int inBuf=0, outBuf=0, count=0;
11
12    public void put(Object x) throws InterruptedException {
13      monitorLock.lock();
14      try {
15        while (count == buffer.length)
16          notFull.await();
17        buffer[inBuf] = x;
18        inBuf = (inBuf + 1) % size;
19        count++;
20        notEmpty.signal();
21      } finally {
22        monitorLock.unlock();
23      }
24    }
25
26    public Object take() throws InterruptedException {
27      monitorLock.lock();
28      try {
29        while (count == 0)
30          notEmpty.await();
31        Object x = buffer[outBuf];
32        outBuf = (outBuf + 1) % size;
33        count--;
34        notFull.signal();
35        return x;
36      } finally {
37        monitorLock.unlock();
38      }
39    }
40 }
```

Figure 3.15: Bounded Buffer Using ReentrantLocks and Conditions

```
1  public class ListQueue {
2      class Node {
3          public String data;
4          public Node next;
5      }
6      Node head = null, tail = null;
7      public synchronized void enqueue(String data) {
8          Node temp = new Node();
9          temp.data = data;
10         temp.next = null;
11         if (tail == null) {
12             tail = temp;
13             head = tail;
14         } else {
15             tail.next = temp;
16             tail = temp;
17         }
18         notify();
19     }
20     public synchronized String dequeue() {
21         while (head == null)
22             Util.myWait(this);
23         String returnval = head.data;
24         if (head == tail) tail = null;
25         head = head.next;
26         return returnval;
27     }
28 }
```

Figure 3.16: Linked list

```
public void method() {
  synchronized (this) {
     body();
  }
}
```

Just as nonstatic methods can be `synchronized`, so can the static methods. A `synchronized` static method results in a classwide lock.

One also needs to be careful with inheritance. When an extended class overrides a `synchronized` method with an unsynchronized method, the method of the original class stays synchronized. Thus, any call to `super.method()` will result in synchronization.

3.5 Dangers of Deadlocks

Since every synchronized call requires a lock, a programmer who is not careful can introduce deadlocks. For example, consider the following class that allows a cell to be swapped with the other cell. An object of class `BCell` provides three methods: `getValue`, `setValue` and `swap`. Although the implementation appears correct at first glance, it suffers from deadlock. Assume that we have two objects, p and q, as instances of class `BCell`. What happens if a thread t_1 invokes `p.swap(q)` and another thread, say, t_2, invokes `q.swap(p)` concurrently? Thread t_1 acquires the lock for the monitor object p and t_2 acquires the lock for the monitor object q. Now, thread t_1 invokes `q.getValue()` as part of the `swap` method. This invocation has to wait because object q is locked by t_2. Similarly, t_2 has to wait for the lock for p, and we have a deadlock!

```
class BCell { // can result in deadlocks
    int value;
    public synchronized int getValue() {
        return value;
    }
    public synchronized void setValue(int i) {
        value = i;
    }
    public synchronized void swap(BCell x) {
        int temp = getValue();
        setValue(x.getValue());
        x.setValue(temp);
    }
}
```

The program that avoids the deadlock is given below. It employs a frequently used strategy of totally ordering all the objects in a system and then acquiring locks only in increasing order. In this program, both `p.swap(q)` and `q.swap(p)` result in either `p.doSwap(q)` or `q.doSwap(p)`, depending on the `identityHashCode` value of the objects `p` and `q`.

```
class Cell {
    int value;
    public synchronized int getValue() {
        return value;
    }
    public synchronized void setValue(int i) {
        value = i;
    }
    protected synchronized void doSwap(Cell x) {
        int temp = getValue();
        setValue(x.getValue());
        x.setValue(temp);
    }
    public void swap(Cell x) {
        if (this == x)
            return;
        else if (System.identityHashCode(this)
                < System.identityHashCode(x))
            doSwap(x);
        else
            x.doSwap(this);
    }
}
```

Another frequent reason for deadlock is when a thread waits for a condition inside a *nested* monitor. For example, suppose that a thread t_1 is inside a monitor for an object `firstObj` and it calls a synchronized method for another object `secondObj`. Now if the thread invokes a `wait()` inside this method, by the semantics of Java monitor, it will release the lock on `secondObj`. However, it will continue to hold the monitor lock of `firstObj`. Suppose that thread t_2 can notify thread t_1, but thread t_2 needs the lock of `firstObj`, we have a deadlock. Thus, one needs to be extra careful when nested monitor calls are made.

3.6 Other Useful Methods in `Thread` Class

. Some other useful methods in Java `Thread` class are as follows:

1. The `interrupt()` method allows a thread to be interrupted. If thread t_1 calls $t_2.interrupt()$, then t_2 gets an `InterruptedException`.

2. The `yield()` method allows a thread to yield the CPU to other threads temporarily. It does not require any interaction with other threads, and a program without `yield()` would be functionally equivalent to `yield()` call. A thread may choose to `yield()` if it is waiting for some data to become available from say `InputStream`.

3. The method `holdsLock(x)` returns true if the current thread holds the monitor lock of the object x.

3.7 Problems

3.1. Show that if the `P()` and `V()` operations of a binary semaphore are not executed atomically, then mutual exclusion may be violated.

3.2. Show that a counting semaphore can be implemented using binary semaphores. (*Hint*: Use a shared variable of type integer and two binary semaphores)

3.3. Give a starvation-free solution to the reader-writer problem using semaphores.

3.4. Show that `BoundedBufferMonitor` does not work for multiple producers and multiple consumers.

3.5. The following problem is known as the *sleeping barber* problem. There is one thread called *barber*. The barber cuts the hair of any waiting *customer*. If there is no customer, the barber goes to sleep. There are multiple customer threads. A customer waits for the barber if there is any chair left in the barber room. Otherwise, the customer leaves immediately. If there is a chair available, then the customer occupies it. If the barber is sleeping, then the customer wakes the barber. Assume that there are n chairs in the barber shop. Write a Java class for SleepingBarber using semaphores that allows the following methods:

```
runBarber() // called by the barber thread; runs forever
hairCut() // called by the customer thread
```

How will you extend your algorithm to work for the barber shop with multiple barbers?

3.6. Give a deadlock-free solution to the dining philosophers problem using semaphores. Assume that one of the philosophers picks forks in a different order.

3.7. Assume that there are three threads—P, Q, and R—that repeatedly print "P", "Q", and "R" respectively. Use semaphores to coordinate the printing such that the number of "R" printed is always less than or equal to the sum of "P" and "Q" printed.

3.8. Write a monitor for the sleeping barber problem.

3.9. How will you implement `wait()`, `notify()` and `notifyAll()` using binary semaphores?

3.10. Show how condition variables of a monitor can be implemented in Java.

3.11. Write a monitor class `counter` that allows a process to sleep until the counter reaches a certain value. The `counter` class allows two operations: `increment()` and `sleepUntil(int x)`.

3.12. Write a Java class for `BoundedCounter` with a minimum and a maximum value. This class provides two methods: `increment()` and `decrement()`. Decrement at the minimum value and increment at the maximum value result in the calling thread waiting until the operation can be performed without violating the bounds on the counter.

3.13. Write a Java class `FifoSemaphore` that is identical to `BinarySemaphore` except that it maintains blocked threads in a *fifo queue*, and the call to `V()` notifies the thread at the head of the queue.

3.14. Implement class `ReaderWriter` using Java monitors.

3.15. Write a Java class for `Multiroom` that is supposed to coordinate access to a set of m rooms. There are two requirements on synchronization.

(a) Multiple threads can be in the same room but no more than one room can ever be occupied. For example, suppose that a thread T_1 is in room[1]. Now if another thread T_2 requests to enter room[2], it must wait. However, if it requests to enter room[1], then it can enter the room.

(b) If a thread is waiting to enter a room, then no thread that arrives later can succeed in entering its room before this thread does.

3.8 Bibliographic Remarks

The semaphores were introduced by Dijkstra [Dij65a]. The monitor concept was introduced by Brinch Hansen [Han72] and the Hoare-style monitor, by Hoare [Hoa74]. Solutions to classical synchronization problems in Java are also discussed in the book by Hartley [Har98]. The example of deadlock and its resolution based on resource ordering is discussed in the book by Lea [Lea99].