Chapter 7 Thread Programming

7.1 Processes and Threads

Processes are fundamental in any computing system. A **process** is a program in execution, plus the data, the stack, registers, and all the resources required to run the program. We can create several processes from the same program and each of them is considered as an independent execution unit. A **multitasking** system allows several processes coexist in the system's memory at the same time. A process has different states, which can be one of the following as shown in Figure 7-1:

1. **New**: when the process is created.
2. **Running**: when instructions are executed, consuming CPU time.
3. **Blocked**: when the process is waiting for some event such as receiving a signal or completing an I/O task to happen.
4. **Ready**: when the process is temporarily stopped, letting another process run and waiting to be assigned to a processor.
5. **Terminated**: when the process has finished execution.

The names are not unique and may be called differently in different systems but the states they represent exist in every system. A system maintains a process table to keep track of the states of the system's processes.

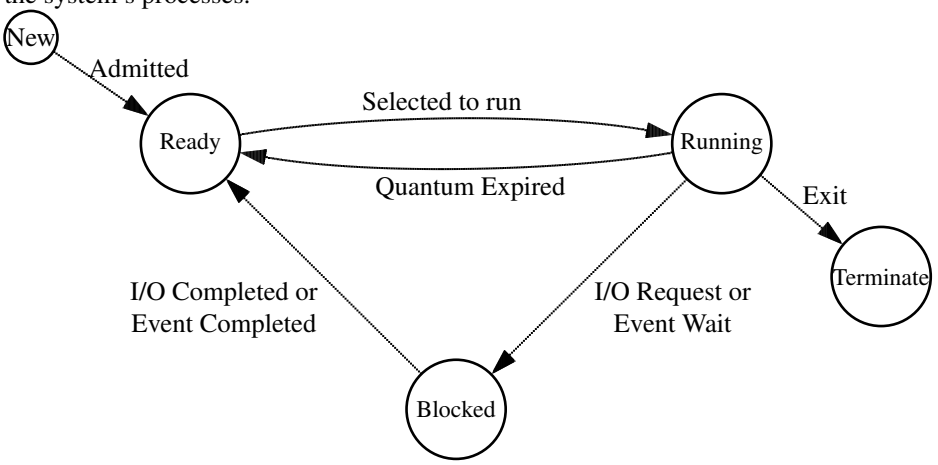

Figure 7-1. States of a Process or a Thread

A process could be bulky, consuming a lot of resources and has its own address space. It performs its own **thread** of operation. Most modern operating system, including Android, extends the process concept discussed to let a process to have multiple threads of operations, allowing it to perform multiple tasks concurrently.

A **thread**, sometimes referred to as a **lightweight process** (LWP), is a basic unit of CPU utilization. It has a thread ID to identify itself, a program counter (PC) to keep track of the next instruction to be executed, a register set to hold its current working variables, and a stack to store the execution history. A thread must execute in a process and shares with its peers the resources allocated to the process. Threads make a system run much more effectively. For example, a web browser might have one thread displaying text and images while another retrieving data from a

remote database. A word processor may have one thread accepting text inputs while another performing spelling check in the background. A modern computing system can run thousands of threads at the same time easily but running thousands of processes concurrently will consume so much resources that it might make the system come to a halt. Figure 7-2 below compares a traditional single-threaded process with a multi-threaded process.

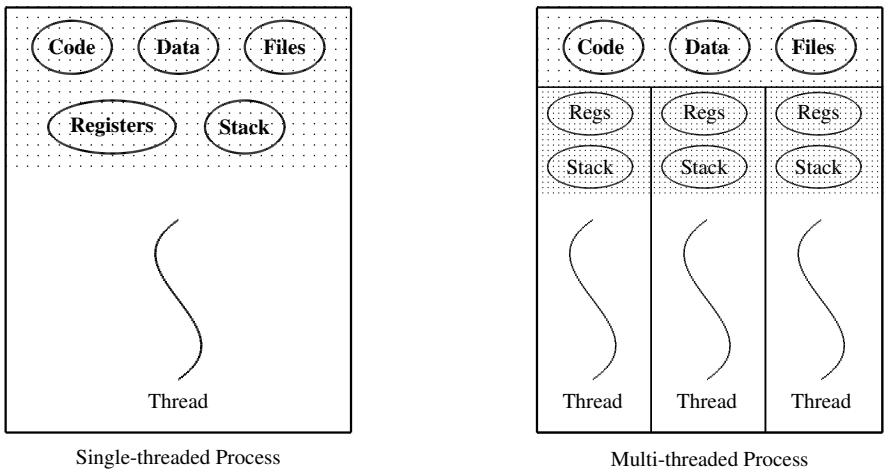

Figure 7-2. Single-threaded and Multi-threaded Processes

The Android official site provides a discussion of Android processes and threads at

http://developer.android.com/guide/components/processes-and-threads.html

7.2 Java Threads

Since Android applications are developed using Java, Android threads inherit the properties of the Java language, which includes direct support for threads at the language level for thread creation and management. However, by design, Java does not support asynchronous behavior. For example, if a Java program tries to connect to a server, the client is blocked, suspending its activities and waiting for the establishment of a connection or the occurrence of a timeout. Typically, the Java app creates a communication thread which attempts to make a connection to the server and a timer thread which will sleep for the timeout duration. When the timer thread wakes up, it checks to see whether the communication thread has finished establishing the connection. If not, the timer thread generates a signal to stop the communication thread from continuing to try.

All Java programs comprise at least one single thread of control, consisting of a **main**() method, running in the Java Virtual Machine (JVM). Correspondingly, a single-thread process (activity) of Android contains the **onCreate**() method.

7.2.1 Thread Creation by Extending *Thread* Class

One way to create a thread is to extend the *Thread* class and to override its **run**() method as shown in the following code:

```
public class CreateThread extends Activity
{
  @Override
  public void onCreate(Bundle savedInstanceState)
```

```
    {
      super.onCreate(savedInstanceState);
      Athread t1 = new Athread();

      t1.start();
      System.out.println ( "I am the main thread!" );
    }
  }

  class Athread extends Thread
  {
    public void run()
    {
      System.out.println ("I am a thread!");
    }
  }
```

If we create and run this program in the Eclipse IDE, we shall see outputs in the **LogCat** that are similar to the following:

TID	Application	Tag	Text
1650	example.createthread	System.out	I am a thread!
1650	example.createthread	System.out	I am the main thread!

7.2.2 Thread Creation by Implementing *Runnable* Interface

Another way to create a thread is to implement the *Runnable* interface directly, which is defined as follows:

```
  public interface Runnable
  {
    public abstract void run();
  }
```

The *Thread* class that our program in the last section extends also implements the *Runnable* interface.

This is why a class that extends *Thread* also needs to provide a **run**() method.

The following code shows a complete example of using this technique to create thread:

```
public class CreateThread1 extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
      super.onCreate(savedInstanceState);
      Athread1 t1 = new Athread1();

      t1.run();
      System.out.println ( "I am the main thread!" );
    }
}

class Athread1 implements Runnable
{
  public void run()
  {
```

```
      System.out.println ("I am a thread!");
    }
  }
```

Again, if we run the code in the Eclipse IDE, we shall obtain outputs similar to those of the previous section.

Each of the two techniques of creating threads has drawbacks and advantages and both are equally popular. The advantage of the first technique, extending the *Thread* class, is that it can use all the methods of the *Thread* class. However, since Java does not support multiple inheritance, if a class has already extended another class, it will not be allowed to extend *Thread*, and we need to use the second technique to create a thread. Of course, if a class implements the *Runnable* interface and does not extend the *Thread* class, none of the the methods provided by *Thread* can be used in the new class. In the above code, we cannot even use the **start**() method to start running the thread!

The *Thread* class provides a few methods to manage threads, including:

1. **stop**(): Terminates the thread. Once a thread has been terminated, it cannot be resumed or restarted.
2. **suspend**(): Suspends execution of the running thread. A suspended thread can be resumed to run.
3. **sleep**(): Puts the running thread to sleep for a specified amount of time.
4. **resume**(): Resumes execution of the suspended thread.

If we construct directly an instance of a class that implements *Runnable* like what we did above, we cannot use any of these methods; a better way to construct a thread with this technique is to pass the object variable of the class as an input parameter to the *Thread* class constructor as follows:

```
public class CreateThread1 extends Activity
{
  @Override
  public void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);

    Runnable runner = new Athread1 ();
    Thread t1 = new Thread ( runner );
    t1.start();
    System.out.println ( "I am the main thread!" );
  }
}
```

Besides the threads created by users, a Java environment has a few threads running asynchronously on behalf of the Java Virtual Machine (JVM). These system threads carry out housekeeping tasks such as memory management and memory controls. In particular, the garbage-collector (GC) thread examines objects in the system to check whether they are alive. Anything that is not alive is designated as garbage and is returned to the memory heap. This garbage collection mechanism allows developers to create objects without worrying about allocation and deallocation of memory. This would eliminate memory-leak problems and help developers create more robust programs in a shorter time. Other interesting system threads include the timer thread, which can be used to schedule tasks and handle time events, and the graphics control threads, which can be used to update the screen and control user interface events such as clicking a button.

7.2.3 Wait for a Thread

Very often after a parent has created a thread, the parent would like to wait for the child thread to complete before its own exit. This can be done using the **join**() method of the *Thread* class. The method forces one thread to wait for the completion of another. Suppose t is a Thread object. The statement

```
t.join();
```

causes the currently executing thread to pause execution until Thread t terminates. A programmer may specify a waiting period by overloading the **join**() method. Like **sleep**, **join** depends on the OS for timing. So we should not assume that **join** will wait for the exact amount of time that we have specified.

Like **sleep**, when **join** is interrupted, it exits with an *InterruptedException*.

We present below an example of multi-threaded programming and the usage of **join**. In this example, we use a number of threads to multiply two matrices. The multiplication of an $M \times L$ matrix A and an $L \times N$ matrix B gives an $M \times N$ matrix C, and is given by the formula,

$$C_{ij} = \sum_{k=0}^{L-1} A_{ik} B_{kj} \quad 0 \le i < M,\ 0 \le j < N \tag{7.1}$$

Basically, each element C_{ij} is the dot product of the i-th row vector of A with the j-th column vector of B. We use one thread to calculate a dot product. Therefore, we totally need $M \times N$ threads to calculate all the elements of matrix C. The calculating threads are created by the main thread, which must wait for all of them to complete. The following is the complete code for the program.

Program Listing 7-1 Multithreaded Matrix Multiplication

```
public class MatMulActivity extends Activity
{
  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    double a[][] = {{1, 2, 3}, {4, 5, 6}};
    double b[][] = { {1, -1}, {-1, 1}, {1, 1} };
    final  int numRows = a.length;
    final  int numCols = b[0].length;
    final  int nThreads = numRows * numCols;
    double c[][] = new double[numRows][numCols];
    int k = 0;

    System.out.println ("Matrix a: ");
    printMat ( a );
    System.out.println ("Matrix b: ");
    printMat ( b );

    Thread threads[] = new Thread[nThreads];
    System.out.println ( "I am the main thread!" );

    for ( int i = 0; i < numRows; i++ )      //row
```

```
      for ( int j = 0; j < numCols; j++ ) {  //column
        threads[k] = new MatMul ( i, j, a, b, c );
        threads[k].start();
        k++;
      }
    try {
      for ( int i = 0; i < k; i++ )
        threads[i].join();     //wait for threads[i] to complete
    } catch ( InterruptedException e ) {}

    System.out.println ("Matrix c = a x b: ");
    printMat ( c );
  }

  //print a matrix
  public static void printMat ( double a[][] )
  {
    int numRows = a.length;
    int numCols = a[0].length;

    for ( int i = 0; i < numRows; i++ ) {
      for ( int j = 0; j < numCols; j++ ) {
        System.out.printf ( "%6.2f, ", a[i][j] );
      }
      System.out.printf("\n");
    }
  }
}

//Thread calculates dot product of a row-vector and a column-vector
class MatMul extends Thread
{
  private int row, col;
  private double a[][];
  private double b[][];
  private double c[][];

  //Calculate dot product of row-vector a0[row0] and column vector
  //  b0[][col0]. Save result in c0[row0][col0].
  public MatMul(int row0, int col0, double a0[][], double b0[][],
                                                    double c0[][])
  {
    row = row0;
    col = col0;
    a = a0;
    b = b0;
    c = c0;
  }

  public void run()
  {
    System.out.println ("I am a MatMul  thread!");
    double sum = 0.0;
    int n = a[0].length;      //number of columns
```

```
      for ( int i = 0; i < n; i++ )
        sum += a[row][i] * b[i][col];

      c[row][col] = sum;
    }
}
```

When we run the program in Eclipse IDE, we will see in the **LogCat** some outputs similar to the following:

```
System.out: Matrix a:
System.out:   1.00,   2.00,   3.00,
System.out:   4.00,   5.00,   6.00,
System.out: Matrix b:
System.out:   1.00,  -1.00,
System.out:  -1.00,   1.00,
System.out:   1.00,   1.00,
System.out: I am the main thread!
System.out: I am a MatMul  thread!
System.out: I am a MatMul  thread!
System.out: I am a MatMul  thread!
System.out: I am a MatMul  thread!
System.out: Matrix c = a x b:
System.out:   2.00,   4.00,
System.out:   5.00,   7.00,
```

7.3 Synchronization

In a multitasking system, several threads may be running at the same time (concurrently). There are situations that the activities of the threads need to be **synchronized** so that the activity of one thread will not interfere or disturb the activity of another. For example, we certainly do not want two threads to print something simultaneously using the same printer. Figure 7-3 shows a World War I Fighter Aircraft, which can be used to illustrate the concept of synchronization. In the days of World War I, the engineering of fighter aircrats was relatively primitive. A machine gun was mounted in front of the pilot but behind whirling propeller. At the beginning of the war, a pilot would not shoot enemy aircrafts which were at the same altitude as the bullets might damage the propeller of his own fighter. Later, the Germans could fly their more advanced aircrafts higher; a German pilot would turn off the engine at higher altitude, diving at the enemy fighter and firing without hitting the propeller. This is an example of **mutual exclusion** where only one event can occur at a time, either running the propeller or firing the machine gun. This would be called **coarse-grained synchronization**. The Germans later developed even more advanced technologies that automatically synchronized the propeller whirling and gun firing, so that bullets were shot only when their blades were not in the way. This would be called **fine-grained synchronization**.

7.3.1 Mutual Exclusion

In a computing system, very often resources are shared among threads. Some resource such as a printer, a memory segment or a file may allow only one thread to access it at a time. The requirement to ensure that only one thread or process is accessing such a shared resource is referred to as **mutual exclusion**. Of course, a thread is a running program and it executes a certain code segment to access the resources. To deny a thread to access a resource means to deny the thread

to execute the corresponding piece of code segment that access the resource. A code segment in a process in which a shared resource is access is referred to as a **critical section**. When the program is about to execute the code segment, we literally say that it is entering the critical section, and when it is finishing its execution, we say that it is leaving the critical section. Situations where two or more threads are competing to enter a critical section is referred to as **race conditions**. Mutual exclusion is often abbreviated to **mutex**.

Figure 7-3 World War I Fighter Aircraft

Achieving mutual exclusion usually involves some locking mechanisms. It is in analogy of the situation when we access a small public restroom that allows only one a person to use it at one time such as one in a small Starbuck or fast food restaurant. When a lady wants to use the restroom, she first examines whether it is *vacant* (unlocked) or *occupied* (locked). If it is locked, she will wait until it is unlocked. If it is unlocked, she will lock it and use it; when she has finished using it, she unlocks the restroom and leaves it. In general, mutual exclusion for accessing a critical section is achieved by following the steps:

1. Create a lock to protect the shared resource.
2. Acquire the lock.
3. Enter the critical section (accessing the shared resource).
4. Release the lock.

In practice, all threads should be able to enter a critical section in finite time. That is, no thread will be in a **starvation** state, waiting infinitely for the desired resource. Another important requirement to achieve mutual exclusion is that the locking and unlocking process must be **atomic**, which means that the action is indivisible. Once it is started, it will not be interrupted by any other thread. In our restroom analogy, when a lady is locking the restroom halfway, no other customer is allowed to enter the restroom to do the same thing. In a computing system, the mechanism is usually achieved with the help of special hardware instructions.

Java uses a concept called **monitor** to achieve synchronization. Every Java object is associated with a lock. Any attempt to lock a monitor that has been locked causes the thread to wait until the lock is released (unless the owner of the lock itself is trying to acquire the lock again). However, Java synchronization is **block-based**. That is, instead of locking the whole object, a thread is able to lock a block of code. This implies that normally the lock is ignored when the object is being accessed; any method or block of code can be accessed as usual unless it is declared as **synchronized**.

A Java thread can always put a lock on a synchronized block of code whenever it needs and releases the lock as it desires. But if a synchronized block has been locked by a thread, no other

thread can access any synchronized method of the object. This mechanism is achieved by using the keyword **synchronized** in the declaration of a method or a block of code in the class. The JVM ensures that only one thread is allowed to lock a method or a block of statements at any instance while other threads remain active, being able to access other availble methods and objects.

A thread calling a synchronized method must first own the lock before it can access the method. If the lock has been acquired by another thread, the calling thread blocks itself and is put in a waiting set, waiting to be waken up to acquire the lock. The following is an example of using the keyword **synchronized**:

```
public class Score
{
  private double score;
  private double total;

  public synchronized void setScore ( double s )
  {
    score = s;
  }

  public synchronized void addScore ( double s )
  {
    total += s;
  }

  public double getScore ()
  {
    return score;
  }
  .....
}
```

In the example, at any instance only one thread is allowed to access the method **setScore**() or **addScore**)(). When a thread is accessing **setScore**(), another thread cannot access the other synchronized method **addScore**() but it can access **getScore**(). Here the method **getScore**() is not synchronized, so multiple threads can access it at the same time without acquiring the lock. This is in analogy of clients accessing the rooms of a building, where some rooms have a lock and others do not. Only one key exists, held by a receptionist at the entrance of the building, and the key can open all lock-rooms. A client can always enter any room that does not have a lock. However, if she wants to enter any room that has a lock, she must first acquire the key. If the key has been checked out, she must wait. When she is done using the lock-room, she would return the key to the receptionist. In this analogy, the same key is used to access all the rooms that has a lock corresponding to the situation that only one synchronized method can be accessed within an object. In other words, only one thread is active inside a monitor. This defeats the purpose of concurrency and could be a drawback of of using a monitor to achieve synchronization in some applications; this shortcoming may be remedied using a tool called *serializer*, which is similar to a monitor except that it contains a special code section called *hollow region*, where threads can be concurrently active. The discussion of serializer is beyond the scope of this book.

As an example of using **synchronized** methods, we use the **synchronized** keyword to write a *Lock* class that can lock any statement or a block of statements:

```
public class Lock
{
  private boolean locked = false;
```

```
  public synchronized void lock () throws InterruptedException
  {
    while ( locked )
      wait();
    locked = true;
  }

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

The **wait** method is to wait for a signal to wake up the thread. When a thread calls **wait**(), the state of the thread is set to **Blocked** (see Figure 7-1), and the thread is put in a waiting set for the synchronized block. The **notify** method is to send a signal to wake up a thread chosen randomly from the waiting set and the awaken thread is placed in the ready queue shown in Figure 7-1. Using this *Lock* class, we can implement the *Score* class like the following without using the **synchronized** keyword explicitly:

```
public class Score
{
  private double score;
  private Lock lock = new Lock();

  public void setScore ( double s )
  {
    try {
      lock.lock();
    } catch ( InterruptedException e ) {
      e.printStackTrace();
    }
    score = s;
    lock.unlock();
  }
  .....
}
```

Starting from Java 5, Java provides a lock package that implements a number of locking mechanisms. So you may not need to implement your own locks. To use the package, you just need to add the import statement:

```
import java.util.concurrent.locks.*;
```

7.3.2 Semaphore

Mutual exclusion ensures that only one thread is accessing a shared resource at one time. It is a very common form of synchronization. However, in many applications we may want to allow more than one thread to access a resource simultaneously but will block further threads from accessing when the current number of threads using the resource has reached a certain number. For example, we want to limit the number of clients reading a web site simultaneously but would let more than one client to read it at the same time. Problems like this and many others cannot be handled by mutual exclusion. They can be solved using the *semaphore*, a synchronization operation introduced by the renown Computer Scientist *E.W. Dijkstra* in the 1960s.

A semaphore S is an integer variable, which, apart from initialization, is accessed through two standard atomic operations called **down** and **up**. Many people also call the two operations **wait** and **signal** or **P** and **V** respectively. Less often, some people refer to the two operations as **lock** and **unlock**. The names **P** and **V** were given by Dijkstra who was Dutch and in Dutch, they may be the initials of two words meaning *decrement* and *increment*:

down(S):	an atomic operation that waits for semaphore S to become positive, then decrements it by 1; also referred to as **wait**(S) or **P**(S).
up(S) :	an atomic operation that increments semaphore S by 1; also referred to as **signal**(S) or **V**(S).

We use the following notations to describe the atomic operations **down** and **up**. The **down**(S) operation is given by:

when ($S > 0$) [
  $S = S - 1;$
]

In the above code, the statements enclosed by the square brackets are referred to as the *command sequence*, and the expression following **when** is referred to as the *guard*. The *command sequence* is executed only when the *guard* is true. The entire code segment is known as a *guarded command*, which is always executed atomically. That is, no other operations could interfere with it while it is executing.

The **up**(S) operation is simpler, simply incrementing S by 1:

$[S = S + 1;]$

Apart from initialization, there is no other way for manipulating the value of a semaphore besides these two operations. Therefore, if S is initially 1 and two threads concurrently operate on it, one executing *down*, the other *up*, then the resulting value of S is always 1. If its initial value is 0 and the two concurrent threads are executing *down* and *up*, the thread executing *down* must wait until the other thread finishes the *up* operation that makes the semaphore's value positive. Then the waiting thread will finish its *down* operation, decrementing the semaphore's value to 0.

If a semaphore's value is restricted to 0, and 1, it is referred to as a *binary semaphore*. Regular semaphores with no such restriction are usually called *counting semaphores*. Binary semaphores are implemented in the same way as regular semaphores except multiple **up** operations will not increase the semaphore value to anything greater than 1.

When a thread executes the **down** operation and finds that the semaphore value is not positive, it must wait. However, rather than busy waiting, the thread can block itself, placing itself in a waiting queue associated with the semaphore, and the state of the thread is switched to the sleeping state (blocked state). The sleeping thread is restarted by a signal (wakeup) associated with the **up** operation.

In Java, a semaphore can be implemented by making use of the **synchronized** keyword. Listing 7-2 below shows an example of implementing semaphore.

Program Listing 7-2 Sample Semaphore Implementation

```
public final class Semaphore
{
  private int S;

  // default initial value is 0
  public Semaphore() {
```

```
    S = 0;
  }

  // initialization of semaphore value
  public Semaphore(int v) {
    S = v;
  }

  // keyword synchronized makes the method 'atomic' and
  //  mutual-exclusive; only one object can access it at one time
  public synchronized void down() {
    while (S == 0) {
       try {
          wait(); // Causes current thread to wait until another
                  //  thread invokes the notify() method or the
                  //  notifyAll() method for this object.
       }
       catch (InterruptedException e) { }
    }
    S--;          // decrement semaphore value
  }

  public synchronized void up() {
    S++;     // increment semaphore value
    notify();     // wakes up a single thread that is waiting on
                  //  this object's monitor.
  }
}
```

In the code, the keyword **final** means the class cannot be further extended. The keyword **synchronize** guarantees mutual exclusion and that when a thread executes the **up** or **down** operation, no other thread can interfere.

Listing 7-3 presents a demo program showing how to use a semaphore to access a simulated critical section.

Program Listing 7-3 Simulating Usage of Semaphore

```
public class SemaphoreTest extends Activity
{
  @Override
  protected void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_semaphore_test);
    //initialize semaphore with value 1
    Semaphore semaphore = new Semaphore ( 1 );

    final int nThreads =  10;

    Athread[] threads = new Athread[nThreads];
    /*
      Semaphore semamphore is shared among the Athread objects.
      Because of the synchronized keyword, only one Athread
```

```
      can 'access' it at a time.
    */
    for (int i = 0; i < nThreads; i++)
       threads[i] = new Athread ( semaphore, "Athread " +
                                  (new Integer(i)).toString() );

    for (int i = 0; i < nThreads; i++)
       threads[i].start();
  }
}

class Tasks
{
   // simulate a critical section
   public static void criticalSection() {
     try {
       Thread.sleep( (int) (Math.random() * 3000) );
     }
     catch (InterruptedException e) { }
   }

   // simulate a noncritical section
   public static void nonCriticalSection() {
     try {
       Thread.sleep( (int) (Math.random() * 3000) );
     }
     catch (InterruptedException e) { }
   }
}

class Athread extends Thread
{
  private Semaphore s;
  private String tname;

  public Athread ( Semaphore s0, String name ) {
    tname = name;
    s = s0;
  }

  public void run()
  {
    while (true) {
      System.out.println ( tname + " trying to enter CS" );
      s.down();
      System.out.println ( tname + " entering CS" );
      Tasks.criticalSection();
      System.out.println ( tname + " exited CS" );
      s.up();

      Tasks.nonCriticalSection();
    }
  }
}
```

When we run the program in the Eclipse IDE, we will see in the **LogCat** outputs similar to the following:

```
Athread 0 trying to enter CS
Athread 0 entering CS
Athread 1 trying to enter CS
Athread 2 trying to enter CS
.....
Athread 0 exited CS
Athread 1 entering CS
Athread 1 exited CS
.....
```

7.3.3 Producer-Consumer Problem

The producer-consumer problem is a common paradigm for thread synchronization and we use it as an example to illustrate the usage of semaphores in synchronization. This problem also will be used to solve practical problems in later chapters.

In the problem, a *producer* thread produces information which is consumed by a *consumer* thread. This is in analogy with whats happening in a fast-food restaurant. The chef produces food items and put them on a shelf; the customers consume the food items from the shelf. If the chef makes food too fast and the shelf is full, she must wait. On the other hand, if the customers consume food too fast and the shelf is empty, the customers must wait.

To allow producer and consumer threads to run concurrently (simultaneously), we must make available a buffer (like the shelf in our fast-food analogy) that can hold a number of items and be **shared** by the two threads; the producer fills the buffer with items while the consumer empties it. A producer can produce an item while the consumer is consuming another item. Trouble arises when the producer wants to put a new item in the buffer, which is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and finds it empty, it goes to sleep until the producer puts something in the buffer and wakes the consumer up. The **unbounded-buffer** producer-consumer problem places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items without waiting. The **bounded-buffer** producer-consumer problem puts a limit on the buffer size; the consumer must wait when the buffer is empty, and the producer must wait when the buffer is full.

The approach sounds simple enough, but if not properly handled, the two threads may **race** to access the buffer and the final outcome depends on who runs first.

In many practical applications, we may use a circular queue to hold more than one item at a time. The producer inserts an item at the tail of the queue and the consumer removes an item at the head of it. We advance the tail and head pointers after an insert and a remove operation respectively. The pointers wrap around when they reach the "end" of the queue. If the tail reaches the head, the queue is full and the producer has to sleep. If the head catches up with the tail, the queue is empty and the consumer has to sleep. Actually, such a queue may handle the situation of multiple producers and multiple consumers. This concept is illustrated in Figure 7-4 below.

Physically, a bounded-buffer is a circular queue. However, logically, we can consider it as a linear queue extending to infinity as shown in Figure 7-5; using this model, when $head = tail$, the bounded-buffer is empty, and when $tail - head = buffer\ size$ (which is 8 in this example), the buffer is full. We increment $tail$ when an item is inserted and increment $head$ when an item is removed from the buffer. The increment operations should be done mutual-exclusively. Therefore, we can define two synchronized methods in the same class to perform the operations.

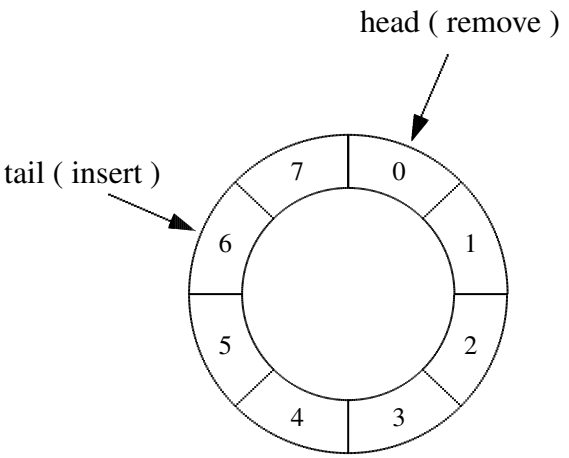

Figure 7-4. Circular Queue with Eight Slots

To simplify things, we make all operations on the tail and the head of the circular buffer **synchronized**. That is, only one thread is allowed to perform an operation at one time. We also assume that the variables $head$ and $tail$ have large enough bit fields that they never overflow in the applications.

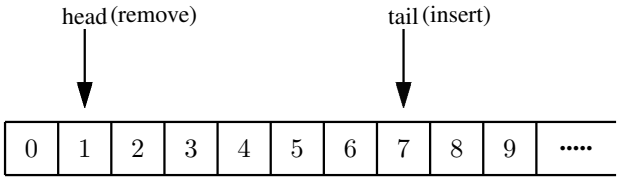

Figure 7-5. Infinite Linear Queue

Listing 7-4 below presents a simple solution to the problem. Certainly this is not the best solution as it only allows one thread to operate on a slot of the buffer at a time. A better solution would allow multiple threads to access the buffer and operate on the slots simultaneously as long as they work on different slots. An object of the *BoudedBuffer* class is to be shared by a number of producer and consumer threads.

Program Listing 7-4 Shared *BoundedBuffer* in the Producer-Consumer Problem

```
// Buffer shared by threads
public class BoundedBuffer
{
   public double buffer[];
   private int head;
   private int tail;
   public int length;  //number of slots in buffer

   //default constructor
   BoundedBuffer ()
   {
     head = tail = 0;
     length = 1;
```

```
    buffer = new double[length];
  }

  BoundedBuffer ( int len )
  {
    head = tail = 0;
    if ( len > 0 )
      length = len;
    else
      length = 1;
    buffer = new double[length];
  }

  public synchronized int insert ( double item )
  {
    // insert only if the buffer is not full
    while ( tail >= head + length ){
      try {
        System.out.println( "Buffer full, producer waits." );
        wait();
      } catch ( InterruptedException e ) {
       e.printStackTrace();
      }
    }
    int t = tail % length;
    buffer[t] = item;      //insert item at tail of queue
    tail++;                //advance tail
    notifyAll();           //wake up all waiting threads

    return t;      //returns slot position of insertion
  }

  //remove only if buffer not empty,returns value in item0[0]
  public synchronized  int remove ( double item0[] )
  {
    while ( tail <= head ){    //Buffer empty
      try {
        System.out.println( "Buffer empty, consumer waits." );
        wait();
      } catch ( InterruptedException e ) {
        e.printStackTrace();
      }
    }
    int h = head % length;
    item0[0] = buffer[h];  //delete at head
    head++;
    notifyAll();           //wakeup all waiting threads

    return h;        //returns slot position of removal
  }
}
```

Listing 7-5 below shows a typical usage of the shared *BoundedBuffer* in a producer-consumer

case. In the example, the buffer has five slots, and a *Producer* class and a *Consumer* class are presented. Four *Consumer* threads and three *Producer* threads are created to simulate the production and consumption operations.

Program Listing 7-5 An Example of Producer-Consumer Problem using *BoundedBuffer*

```
public class ProducerConsumerMain extends Activity
{
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    final int n1 =  4, n2 = 3;
    final int length = 5;
    //shared buffer
    BoundedBuffer buffer = new BoundedBuffer ( length );

    //Create n1 Consumer threads and n2 Producer threads
    Consumer [] cons = new Consumer[n1];
    Producer [] prods = new Producer[n2];
    for (int i = 0; i < n1; i++)
      cons[i] = new Consumer(buffer,"Consumer " +
                          (new Integer(i)).toString());
    for (int i = 0; i < n2; i++)
      prods[i] = new Producer(buffer,"Producer " +
                           (new Integer(i)).toString());

   //start the threads
   for (int i = 0; i < n1; i++)
     cons[i].start();
   for (int i = 0; i < n2; i++)
     prods[i].start();
 }

 class Producer extends Thread
 {
   private BoundedBuffer buffer;
   private String tname;

   public Producer (BoundedBuffer b,  String name)
   {
     buffer = b;
     tname = name;
   }

   double produceItem()
   {
     double x = Math.random();
     return x;
   }

   public void run()
   {
     int p;
     while (true) {
```

```
        double item = produceItem();

        int t = buffer.insert ( item );
        System.out.printf("%s inserted into slot %d value:\t%1.3f\n",
                                                    tname, t, item);
         try {
           Thread.sleep( (int) (Math.random() * 2000) );
         } catch (InterruptedException e) { }
     }
    }
  }

  class Consumer extends Thread
  {
    private BoundedBuffer buffer;
    private String tname;

    public Consumer ( BoundedBuffer b,  String name ) {
      buffer = b;
      tname = name;
    }

    public void run()
    {
      double [] item = new double[1]; //holds one item
      int h;
      while (true) {
        h = buffer.remove( item );
        System.out.printf("%s got from slot %d value:\t%1.3f\n",
                                              tname,h,item[0]);
        try {
          Thread.sleep( (int) (Math.random() * 3000) );
        } catch (InterruptedException e) { }
      }
    }
  }
```

When we run the program using Eclipse IDE, we will see in the **LogCat** console messages similar to the following:

```
Buffer empty, consumer waits.
Buffer empty, consumer waits.
Buffer empty, consumer waits.
Buffer empty, consumer waits.
Producer 0 inserted into slot 0 value:  0.024
Buffer empty, consumer waits.
Consumer 3 got from slot 0 value:       0.024
Buffer empty, consumer waits.
Buffer empty, consumer waits.
Producer 2 inserted into slot 1 value:  0.550
Producer 1 inserted into slot 2 value:  0.883
Consumer 0 got from slot 1 value:       0.550
Consumer 1 got from slot 2 value:       0.883
Buffer empty, consumer waits.
```

```
Producer 2 inserted into slot 3 value:  0.779
Consumer 2 got from slot 3 value:       0.779
Producer 0 inserted into slot 4 value:  0.224
Producer 1 inserted into slot 0 value:  0.540
Producer 1 inserted into slot 1 value:  0.078
......
```

7.3.4 Condition Variable

We have seen that semaphores are elegant tools, which can be conveniently used to solve many synchronization problems. However, for many other problems, it is cumbersome to use semaphores and their solutions expressed in semaphores could be complex. Therefore, many computing systems, including Android, provide an additional construct called *condition variable* for concurrent programming. A *condition variable* is a queue of threads (or processes) waiting for some sort of notifications. This construct has been supported by POSIX , SDL (Simple DirectMedia Layer), and Win-32 events in C/C++ programming environments. The Java utility library and the Android platform both provide support of this construct.

A condition variable queue can only be accessed with two methods associated with its queue. These methods are typically called **wait** and **signal**. The **signal** method is also referred to as **notify** in Java. This tool provides programmers a convenient way to implement guarded commands. Threads waiting for a guard to become true enter the queue. Threads that change the guard from false to true could wake up the waiting threads in the queue.

The following code segments show the general approach presented in guarded commands and an outline of implementation using Android Java with exception code omitted.

Guarded Command

```
When ( guard ) [
    statement 1
    .....
    statement n
]
```

Android Implementation

```
class SharedResource
{
   final Lock mutex = new ReentrantLock();
   final Condition condVar = mutex.newCondition();
   boolean condition = false;
   .....
   public void methodA() throws InterruptedException
   {
      mutex.lock();
      while ( !condition );
         condVar.await();
     statement 1
      .....
      statement n
      mutex.unlock();
   }
   .....
}
```

The above code shows that the execution of statements is protected by a guard. In the Android implementation, to evaluate a guard safely, a thread must mutually exclude all other threads evaluating it. This is accomplished by declaring a condition variable, (*condVar* in the example), which always associate with a lock (*mutex* in the example). The thread first locks the lock *mutex* to achieve mutual exclusion. If the guard is true, the thread can execute the command sequence, still

locking *mutex*. It unlocks *mutex* only when all statements of the command sequence have been executed.

An interesting situation arises when the guard is false, which makes the thread execute the **await**() method of the condition variable *condVar*; the operation puts the thread in the queue of the condition variable and the thread is suspended. It seems that the thread would wait forever in the queue as the guard is locked by *mutex* and no other thread can access it. But what we want is that the thread waits until the guard becomes true. Here is what the condition variable comes into play. Right before the thread enters the queue and gets suspended, it unlocks *mutex* temporarily so that another thread can change the value of the guard. The thread that changes the guard from false to true is also responsible for waking up the waiting thread.

So the **await**() method of a condition variable works in the following way.

1. It causes the current thread to wait until it is signaled or interrupted.
2. The lock associated with the condition variable is atomically released and the current thread is suspended until one of four events happens:
   (a) Some other thread executes the **signal**() method of this condition variable and the current thread happens to be selected from the queue as the thread to be awakened.
   (b) Some other thread executes the **signalAll**() method of this condition variable, which wakes up all waiting threads in the queue.
   (c) The current thread is interrupted by some other thread, and interruption of thread suspension is supported.
   (d) An event of *spurious wakeup* occurs.

   In any of the four cases when the current thread wakes up, before the method returns, the thread must re-acquire the lock associated with the condition variable. This guarantees that the thread works in the same way when the guard is true at the beginning.

The following code segment shows the situation that the guard is modified by a thread.

Guarded Command

```
//code modifying guard
    .....




]
```

Android Implementation

```
class SharedResource
{
   .....
   public void methodB() throws InterruptedException
   {
      mutex.lock();
      condition = true; //or code modifying guard
      .....
      condVar.signal();
      mutex.unlock();
   }
}
```

Android provides two classes, *Condition* and *ConditionVariable*, to create condition variable objects. The *Condition* class is provided by the Java utility library. It uses the method **signal**() to wake up one waiting thread, and **signalAll**() to wake up all waiting threads.

The class *ConditionVariable*, which implements the condition variable paradigm, is unique to Android, not supported by traditional Java. Its methods **open**, **close**, and **block** are sticky, its exact function depending on the state of the thread. The method **open** corresponds to the traditional **signalAll** method that releases all threads that are blocked (waiting). The method **block** corresponds to the traditional **wait** method, which blocks the current thread until the condition

becomes true (or opened in the Android's documentation). If **open** is called before **block**, the method **block** will not block but instead returns spontaneously.

As a condition variable always associates with a lock, to use the *Condition* class, we need to include in our program the import statments:

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
```

To use *ConditionVariable*, we need to include:

```
import android.os.ConditionVariable;
```

Now we can present a solution to the producer-consumer problem using condition variables. We rewrite the *BoudedBuffer* class discussed above with use of condition variables. Here, for simplicity and clarity of presentation, we hard-code the buffer size, omit the print statements and do not return the current head and tail positions of the queue. We use two condition variables *notFull* and *notEmpty* for the producer threads to wait until the buffer is not full and the consumer threads to wait until the buffer is not empty. The lock *mutex* is to ensure that only one thread is examining the status buffer; the thread releases *mutex* if it has to wait for the condition to become true and re-acquire it when it is awakened. The complete code of the class is shown in Listing 7-6 below.

Program Listing 7-6 Implementation of *BoundedBuffer* with Condition Variables

```
class BoundedBuffer
{
  final Lock mutex = new ReentrantLock();
  final Condition notFull  = mutex.newCondition();
  final Condition notEmpty = mutex.newCondition();
  final int length = 100;
  final Object[] items = new Object[length];
  int tail, head;

  public void insert ( Object x ) throws InterruptedException
  {
    mutex.lock();        //Exclude others while examining guard
    try {
      while ( tail >= head + length )  //buffer full
      notFull.await();   //wait until buffer not full
      int t = tail % length;
      items[t] = x;      //insert item at tail
      tail++;            //advance tail
      notEmpty.signal();
    } finally {
      mutex.unlock();
  }
 }

  public Object remove() throws InterruptedException
  {
    mutex.lock();
    try {
      while ( tail == head )  //buffer empty
        notEmpty.await();     //wait until buffer not empty
```

```
            int h = head % length;
            Object x = items[h];
            head++;               //Advance head
            notFull.signal();
            return x;
        } finally {
            mutex.unlock();
        }
    }
}
```

7.3.5 Readers-Writers Problem

The readers-writers problem concerns the access of a shared file or database, where some threads, known as *readers*, may want to read the shared resource, whereas others, *known as writers*, may want to write to it. The problem allows concurrent reads but mutual exclusion must be maintained when a writer modifies the data. This is a good example of a synchronization problem that lacks an elegant solution using semaphores only, but can be solved conveniently using condition variables. The following code presents a solution expressed using guarded commands:

```
void reader() {
    when ( nWriters == 0 ) [
        nReaders++;
    ]

    // read

    [nReaders–;]
}
```

```
void writer() {
    when(nReaders==0 && nWriters==0)[
        nWriters++;
    ]

    // write

    [nWriters–]
}
```

In the code, *nReaders* is the number of readers, the number of threads that are currently reading, and *nWriters* is the number of writers that are currently writing. Listing 7-7 below shows an Android implementation of this solution.

Program Listing 7-7 Readers-Writers Problem with Condition Variables

---

```
class ReaderWriter
{
  final Lock mutex = new ReentrantLock();
  final Condition readerQueue=mutex.newCondition();//cond variable
  final Condition writerQueue=mutex.newCondition();//cond variable

  int nReaders = 0;  //number of reader threads
  int nWriters = 0;  //number of writer threads (0 or 1)

  void reader() throws InterruptedException
  {
    mutex.lock();   //mutual exclusion
    while ( !(nWriters == 0) )
```

```
      readerQueue.await();//wait in readerQueue till no more writers
    nReaders++;     //one more reader
    mutex.unlock();
    //read
    //........
    //finished reading
    mutex.lock();   //need mutual exclusion
    if ( --nReaders == 0 )
      writerQueue.signal(); //wake up a waiting writer
    mutex.unlock();
  }

  void writer() throws InterruptedException
  {
    mutex.lock();
    while ( !((nReaders == 0) && (nWriters == 0)) )
      writerQueue.await();  //wait in writerQueue
                            //  until no more writer & readers
    nWriters++;             //one writer
    mutex.unlock();
    //write
    //........
    //finished writing
    mutex.lock();            //need mutual exclusion
    nWriters--;               //only one writer at a time
    writerQueue.signal();     //wake up a waiting writer
    readerQueue.signalAll();  //wake up all waiting readers
    mutex.unlock();
  }
}
```

Here *readers* wait on the condition variable *readerQueue* when the number of *writers* is not 0, indicating a thread is modifying the data. The *writers* wait on the condition variable *writerQueue* when either *nReaders* or *nWriters* is not zero, indicating the presence of some threads that are reading or a thread that is writing. When a reader thread has finished reading and finds that no more thread is reading, it wakes up all the first waiting *writers* and all the waiting *readers*. The awakened threads will all try to acquire the *mutex* lock. If one of the *readers* gets the lock first, reading occurs with the awakened writer thread waiting, otherwise writing takes palce and all *readers* have to wait.

Some readers might have noticed that if *readers* arrive frequently, the writer thread may be in **starvation**, never getting a chance to write. This problem is actually referred to as the readers-writers problem with readers priority. This is in analogy of the situation that a female janitor tries to clean a gentlemen restroom. She has to wait until all users have left. If the restroom is heavily used and gentlemen arrive continuously, she will never have a chance to clean it. In practice, to resolve the issue, she would put up a sign to block new entries and enter the men's room after the last current user has left.

In the same way, the starvation issue in the readers-writers problem can be solved by giving *writers* higher priorities, and the problem is now referred to as readers-writers problem with writers priority. The writers-priority solution is to let the writer just wait for current *readers* to finish and block newly arrived *readers*. The following code presents a solution in guarded commands:

```
void reader() {
    when ( nWriters == 0 ) [
        nReaders++;
    ]

    // read

    [nReaders–;]
}
```

```
void writer() {
    nWriters++;
    when(nReaders==0 && nActiveWriters==0)[
       nActiveWriters++;
    ]

    // write

    [nWriters–; nActiveWriters;]
}
```

In this code, *nWriters* represents the number of *writers* that have arrived, either currently writing or waiting to write. The new variable *nActiveWriters*, which can either be 0 or 1, represents the number of *writers* that are currently writing. *Readers* must now wait until no more *writers*, either writing or waiting to write, has arrived. *Writers* wait as before, until no other threads are reading or writing.

The Android implementation of this solution is presented in Listing 7-8 below:

Program Listing 7-8 Solution of Readers-Writers Problem with Writers-Priority

```
class ReaderWriterPriority
{
  final Lock mutex = new ReentrantLock();
  final Condition readerQueue = mutex.newCondition(); //cond variable
  final Condition writerQueue = mutex.newCondition(); //cond variable

  int nReaders = 0;  //number of reader threads
  int nWriters = 0;  //number of writer threads (0 or 1)
  int nActiveWriters = 0;  //number of threads currently writing

  void reader() throws InterruptedException
  {
    mutex.lock();    //mutual exclusion
    while ( !(nWriters == 0) )
      readerQueue.await();//wait in readerQueue until no more writers
    nReaders++;      //one more reader
    mutex.unlock();
    //read
    //........
    //finished reading
    mutex.lock();   //need mutual exclusion
    if ( --nReaders == 0 )
      writerQueue.signal();   //wake up a waiting writer
    mutex.unlock();
  }

  void writer() throws InterruptedException
  {
    mutex.lock();
    nWriters++;               //a writer has arrived
    while ( !((nReaders == 0) && (nActiveWriters == 0)) )
      writerQueue.await();    //wait in writerQueue
```

```
                                //  until no more writer & readers
      nActiveWriters++;         //one active writer
      mutex.unlock();
      //write
      //........
      //finished writing
      mutex.lock();             //need mutual exclusion
      nActiveWriters--;         //only one active writer at a time
      if ( --nWriters == 0 )    //no more waiting writers, so wake
        readerQueue.signalAll();// up all waiting readers
      else                      //has waiting writer
        writerQueue.signal();   //wake up one waiting writer
      mutex.unlock();
    }
  }
```

7.4 Deadlocks

Suppose you have a turkey dinner with your siblings and cousins and all of you are highly civilized like what is shown in Figure 7-6. There is a cooked whole turkey on the table. Also on the table are a public knife and a public fork, which will be shared by all of you. If you want to eat turkey meat you must first acquire the public knife and the public fork to cut a piece and put it on your own private plate. After returning the knife and the fork to the table, you can begin to enjoy eating your turkey meat using your own private utensils. If each of you acquires the tools (knife and fork) one by one, you may run into a situation in which you have acquired the knife and your little brother has acquired the fork. If no one wants to share the usage of the public tools, then you have to wait for him to release the fork and he he has to wait for you to release the knife. If none wants to yield, no one can proceed to get any turkey. Both of you have to wait forever and this situation is called **deadlock**.

Figure 7-6 Resource Sharing (Image downloaded from *http://disney-clipart.com/Thanksgiving/)*

In this simple example, *is there any way to prevent deadlock by setting some rules of using the tools*? Actually, there a simple way to prevent deadlock by ordering the acquiring of tools.

For example, you can set a simple rule of using the tools: *if you need to use both the knife and the fork, you must first hold (have acquired) the knife before you are allowed to pick up the fork.* Under this rule, deadlock will not occur as only one of you can hold the knife at any instance. Of course, while you are holding the knife, if your little brother has acquired the fork, he is certainly allowed to use it to get some other food such as mashed potato and beans that do not need a knife to fetch. After he has finished fetching those food, he has to put the fork back on the table before he is allowed to get the knife to get any turkey meat and now you can pick it up to fetch the turkey meat. In computer systems, ordering shared resources and enforcing the rule of acquiring them in order is a simple way to prevent deadlock. The following two examples demonstrate this method. In the examples, two threads will acquire two *Mutex* variables to access a critical section. In the first example, the two threads do not acquire the *Mutex* objects in the same order, and deadlock occurs, while in the second example, they acquire *Mutex* objects in the same order and the system is deadlock free.

The following program creates two threads that result in a deadlock state:

Program Listing 7-9 Deadlocked Threads

```
// A demo of the existence of deadlock
package thread.deadlock;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.Menu;

public class MainActivity extends Activity
{
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Mutex mutexX = new Mutex( "mutexX" );
    Mutex mutexY = new Mutex( "mutexY" );

    // Thread A tries to acquire mutexX, then mutexY
    Athread A = new Athread( mutexX, mutexY, "A");
    // Thread B tries to acquire mutexY, then mutexX
    Athread B = new Athread( mutexY, mutexX, "B");

    A.start();
    B.start();
  }
}

class Mutex
{
  public String mname;
  public Mutex( String name )
  {
    mname = name;
  }
}

class Athread extends Thread
{
```

```
   private Mutex first, second;
   private String tname;

   public Athread(Mutex f, Mutex s, String name ) {
      first = f;
      second = s;
      tname = name;
   }

   public void run() {
     synchronized (first) {
         // do something
   try {
         Thread.sleep( ((int)(Math.random()+1))*1000);
   } catch (InterruptedException e) {}
       Log.v ("Thread Info", tname + " thread got " + first.mname);
       synchronized (second) {
         // do something
         Log.v("Thread Info", tname + " thread got " + second.mname );
       }
     }
   }
}
```

In the example, each of the threads tries to acquire both of the mutex objects, *mutexX* and *mutexY* but they try to acquire them in different orders. Consequently, each can only obtain one mutex and wait for the other one, resulting in a deadlock state. When we run the program, we will see in the log output the following statements:

```
Thread Info: A thread got mutexX
Thread Info: B thread got mutexY
```

This indicates that neither thread has acquired both mutex objects and they have to wait forever.

By modifying the code so that both threads acquire the mutex objects in the same order, we can ensure that the two threads are deadlock free. The following code listing shows this situation:

Program Listing 7-10 Deadlock-Free Threads

```
// Deadlock free code
package thread.deadlockfree;

public class MainActivity extends Activity
{
  @Override
  public void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    Mutex mutexX = new Mutex( "mutexX" );
    Mutex mutexY = new Mutex( "mutexY" );

    // Both Thread A and  try to acquire mutexX, then mutexY
    Athread A = new Athread( mutexX, mutexY, "A");
    Athread B = new Athread( mutexX, mutexY, "B");
```

```
    A.start();
    B.start();
  }
}
// Rest of code same as Listing 7-9
```

When we run the code, we obtain the following log outputs; both threads have acquired both *Mutex* objects successfully.

```
Thread Info: A thread got mutexX
Thread Info: A thread got mutexY
Thread Info: B thread got mutexX
Thread Info: B thread got mutexY
```

This means that no deadlock has occurred. In developing muti-threaded applications, one has to be careful to avoid deadlock situations.