Chapter 9: Concurrency

Structures that support concurrent execution are important in programming languages. Two models for invoking concurrent units of execution are the process model and the thread model. Also important are the structures used to share resources among concurrent units, facilities for communication between units, and ways to synchronize the execution of different units.

9.1 Concurrent Structures

Thus far we have viewed a program as a sequential process, working on exactly one task at a time, therefore performing several tasks in sequence, one after another. In this section, we introduce the concept of concurrency—that is, the ability to do several processes at once—and discuss the manner in which concurrency can be represented in a programming language.

There are several actual processor configurations used to provide concurrent execution capabilities. Multiprogramming is the configuration used by a time-sharing system. Here, a single processor is shared by a number of processes, with the processes executing alternately under control of the operating system. Multiprocessing requires that separate processors be available so that multiple processes can each be executing on its own processor, exhibiting true simultaneous operation. All processors share a common memory, which can be used as a communication medium between the processes. Distributed processing also requires separate processors, but here each processor has its own memory. Distributed processors are connected via communication lines, such as an ethernet network.

For the purpose of our discussion of programming language features, the configuration used to implement concurrency is irrelevant. The language will provide an abstract method for viewing and implementing concurrency that can be implemented on any of the three configurations, although some language constructs fit more easily into one configuration than the others.

Concurrent units can be compared to procedural abstraction. Like procedures and methods, they are units of abstraction having their own, often sequential, definition. Like procedures and methods, their use in a program enhances the modularity and level of abstraction in a program by defining their own unit of binding and scope. Concurrent units are more general than procedures and methods, because they have their own thread of control that is independent of any “caller” or invoking mechanism. However, because of the closeness of the procedural and concurrent unit abstractions, concurrent units require the same considerations as other procedural abstractions, namely definition, invocation, and data sharing.

In any case, the major distinguishing feature is that once it is invoked, the invoking unit can proceed with its execution without waiting for the invoked unit to be completed, as illustrated in Figure 9.1. Therefore, in addition to the preceding considerations, two further considerations with concurrent units are the need for communication between concurrent units and the need to synchronize concurrent executions with respect to shared resources.

9.2 Concurrent Units

There are three primary models for the definition of concurrent units. The first is defining the concurrent unit at the control structure level. This was observed in Chapter 6, where control structures for parallel execution were introduced. Figure 9.2(a) shows an example of such a definition in Concurrent Pascal.

The second model for the definition of concurrent units is that of a procedural abstraction. In
this model, the definition is given as a procedure definition. When this procedure is called, it is called into concurrent execution rather than sequential execution. Figure 9.2(b) gives an example of this type of definition.

The third model of concurrent unit definition is that of a data abstraction or class definition. In this case, the definition of the concurrent unit takes the form of a class definition in the language with each instantiation of the class capable of concurrently executing its execution unit upon receipt of the appropriate message. Figure 9.2(c) gives an example of this type of concurrency definition using Java Threads as an example. Here the class bigCalcThread is created and each object that is instantiated from that class becomes a possible concurrent execution unit. The main program in Figure 9.2(c) creates ten such Threads and starts each of them executing by sending them the run() message.

**Figure 9.1** Concurrent Threads of Control

![Concurrent Threads of Control](image)

**Figure 9.2** Examples of Concurrent Unit Definition

(a)  
```
procedure concurrent;
    var sum, i: integer;
begin
    sum := 0;
    cobegin
        for i := 1 to 10 do
            bigCalc;
    coend;
end;
```

(b)  
```
task type bigCalc;
task bigCalc is
    begin
        ...
    end bigCalc;

procedure concurrent is
    process : array(1..10) of bigCalc;
begin
    -- task begins now!
end concurrent;
```

(c)  
```
class bigCalcThread extends Thread {
```
9.3 Invocation of Concurrent Units

The invocation of concurrent units can either be implicit or explicit. Implicit invocation assumes the concurrent process belongs to the program unit in which it is defined. Whenever a program unit begins execution, all concurrent units that belong to that unit (i.e., are defined in that unit) begin their execution simultaneously. When the master unit terminates, it will wait until all its concurrent units terminate before it returns to its invoking unit. This implicit invocation process is illustrated in Figure 9.3. Ada employs implicit invocation semantics.

Concurrent units that are invoked explicitly are invoked in one of two ways. For units declared explicitly as procedures or as types, their invocation is just like that for procedures—that is, as a statement abstraction through reference to the name to which the unit is bound by its definition. For units declared as a class, invocation is accomplished through sending a message to an instantiation of that class to initiate concurrent execution. Both of these methods permit invocation of the concurrent unit at any point within the program unit in which it is defined.

Most methods of invoking a process have a “fork-join” type of behavior. The invoking master unit “forks” the invoked processes, splitting its own thread of control into multiple threads and becoming a process itself as it executes its own sequential code. When it completes its own parallel execution, it must wait for the “join” of its child processes—that is, the termination of these processes—before it can continue further sequential execution.

9.4 Data Sharing

It is frequently necessary for concurrently executing units to share data with each other. Two models are commonly used to accomplish this. The first is the use of shared memory, which is discussed in this section. The second model for data sharing, through information passing from one unit to another, is discussed in the next section.

When multiple concurrent units share memory, data exist that are global to all units in the sense that several units have access to that data. These data typically take the form of variables that all units can access or data structures held in memory for all units to use.
**Figure 9.3** Implicit Invocation of Concurrent Units

```
procedure P;
  concurrent unit C1;
  concurrent unit C2;
  concurrent unit C3;
begin ...
end;
```

Invocation of P

- Execution of P
  - Wait for Termination of all concurrent units
    - Termination of P
      - Return to Invoking unit

- Execution of C1
  - Termination of C1

- Execution of C2
  - Termination of C2

- Execution of C3
  - Termination of C3
Program Concurrent;
  var N : integer;

  concurrent unit P1;
  begin
    N := N + 1;
  end;

  concurrent unit P2;
  begin
    N := N + 2;
  end;

  begin
    N := 3;
    cobegin
      P1;
      P2;
    coend;
    writeln(N);
  end.

This type of sharing is illustrated in Figure 9.4 by Concurrent Pascal syntax. The example is made up of two concurrent units, P1 and P2, each of which uses the global variable N. It is important to note that the results of the program execution can vary, depending on the timings of the executions of the concurrent units. If the sequence is P1 followed by P2, then N results in the value 6. But, consider the sequence in Figure 9.5. Here P1 is executing $N := N + 1$ at exactly the same time that P2 is executing $N := N + 2$, and the result value is not the expected 6 but rather 4.

Figure 9.5 illustrates a potential problem that affects all concurrent units that share memory. Access conflict can occur when concurrent access of shared data is allowed. Thus, such access needs to be done with care, because it could lead to inaccurate results and affect the rest of the program’s execution. The prevention of this kind of problem with shared variables is one of the major applications of synchronization, explained in Section 9.6.
Figure 9.5 Execution of the Program in Figure 9.4

<table>
<thead>
<tr>
<th>Time</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fetch N</td>
<td>Add 2</td>
</tr>
<tr>
<td></td>
<td>Fetch N</td>
<td>Add 2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Laboratory: Write a program that demonstrates the corruption of data through poor implementation of data-sharing protection. Then correct your program and demonstrate that your corrected version protects the data properly.

9.5 Interprocess Communication

The second way that data is shared is through information passing from one unit to another, a method typically referred to as **interprocess communication** (IPC). We discuss IPC separately from shared memory, although a shared variable provides a very simple way for concurrent units to communicate. As with all ways of exchanging information, IPC requires a message sender and a message receiver. The different ways these two work together on exchanging the message result in two models for passing information between concurrent units: the mail model and the phone model. The difference between these two models lies in whether or not the receiving unit needs to attend to the message before the sending unit may proceed.

Under the mail model, information is sent by unit S to unit R and placed in a mailbox. Unit S can then proceed with its execution and unit R can come and retrieve the message at any later time. If more than one message is sent to the same mailbox, the messages are usually queued up within the mailbox so unit R can successively retrieve messages until the mailbox is empty.

There are three ways the mailbox might be identified, providing three different versions of the mail model. These three ways are distinguished by the way in which communication takes place and are illustrated in Figure 9.6. The many-to-one version is analogous to the way a typical post office mailbox operates, with messages arriving from any of a number of processes but destined for only one specific process. Therefore, the sender specifies the receiver of the message, but the receiver retrieves messages without needing to specify the sender.

The one-to-one version accepts messages from exactly one sender. Here the sender not only must specify the receiver, but when the receiver retrieves the message, it must specify the identity of the sender as well. A given mailbox then is identified with both the sending and the receiving processes. This situation is similar to a mailbox used to pass information from a boss to a secretary, where all messages come from the same sender and all go to the same receiver.
The many-to-many version accepts messages from many processes, and these messages may be retrieved by many processes. A sending process therefore places the message in the mailbox without specifying whom the receiver is to be. The next process to retrieve from that mailbox is then the receiving process. This is similar to a mailbox in an office with many bosses and many secretaries, where a boss puts a job to be done in a mailbox and the next available secretary retrieves the message from the box and does the job specified.

Under the mail model, the sender simply sends the message and does not wait for message receipt. The second model for passing information, the phone model, requires that the sending unit wait for the receiving unit to accept the message before proceeding. This is analogous to placing a phone call where the caller must wait for the person called to respond before the message can be sent. The phone model is also known as the rendezvous model, where two people meet together at a prearranged location to pass information. By its nature, the phone, or rendezvous, model also synchronizes the two processes, because they must wait to make a simultaneous contact for the message to be sent.

There are two forms of the phone model, each with different views of waiting. In the first form, the sender waits for notification from the receiver only that the message had been received, and upon this notification, both units continue with their concurrent execution. In the second form, the rendezvous version, the sender waits for both message receipt and message processing. This second version of the phone model is similar to a procedure call: the caller calls the procedure, sending parameters, and must wait until this procedure returns, possibly with modified parameters. The analogy is so strong, in fact, that this second form of the phone model is typically referred to as a remote procedure call (RPC).

As with the mail model, the phone model might be Many-to-one, one-to-one, or many-to- many.
Figure 9.6 Three Types of Mailboxes for Concurrent Communication

*Many-to-One Mailbox*

Sending Processes

Receiving Process

*One-to-One Mailbox*

Sending Process

Receiving Process

*Many-to-Many Mailbox*

Sending Processes

Receiving Processes
Discuss: Is there a type of problem to which the mail model applies more than the phone model?

Laboratory: Determine the IPC mechanism used in a language assigned to you and write a program to demonstrate it.

Laboratory: Can you “simulate” other IPC mechanisms in your language? If you can demonstrate how to do this, write some demonstration programs.

9.6 Synchronization

Two processes that are executing concurrently frequently need to be synchronized. Often, this is necessary if several processes share the same resource, but only one can access the resource at a time. This is called mutual exclusion, an example of which is a set of processes accessing the same data. In order to protect the integrity of the data, only one process at a time should be permitted access to the data. We illustrated a problem with the lack of mutual exclusion in Figure 9.5.

In other situations, one process may need to wait until another process has completed some operation on a shared resource before the first process can proceed. This is called mutual dependency. An example of mutual dependency is one process collecting data while another operates on it. This case is illustrated by a pipeline, where the latter process should not be permitted to begin its operational activity until the collecting process has “filled the pipe” with sufficient data.

There are two common general approaches used by programming languages to specify the synchronization of processes. We will call these the token and the gate approaches.

The token approach makes use of a hypothetical token, which only one process may possess at a time. There are two operations, get-token and replace-token. If a process P executes a get-token command on token T and the token is not presently possessed by any other process, process P becomes the owner of T. If token T is already owned by some other process, then process P is suspended and must wait until the token is available again. Usually, when several processes are waiting for a token, they are placed in a queue, so the earliest arriving process gets possession of the token first. The process that possesses the token can execute a replace-token command, giving the token to the next process on the queue and permitting it to proceed. If there is no process on the queue, the token then becomes available.

The token approach can be used to implement mutual exclusion in a natural way. A token is associated with each shared resource. When a process wishes to use the resource, it asks for the resource’s token with a get-token command. When a process has possession of the token, it has exclusive permission to use the resource. During the time a process has possession of the resource, any other process issuing a get-token command will be placed on a queue to await its turn to use the resource. After a process is finished with the resource, it makes the token available again through a replace-token command. Therefore, ownership of the token represents permission to access the shared resource.

The token approach can also be used to implement mutual dependency. Here the procedure performing the required activity on the shared resource will take the token and not replace it until that activity is completed to the point where the other process may proceed. The dependent
process will request the token before proceeding to the part of its task that requires that the activity of the other process be completed.

The gate approach to synchronization works as follows. A hypothetical gate exists, which blocks the execution of a process. That process may proceed beyond the gate only if some other process comes along and opens the gate.

Gate synchronization is based on two primitive commands: wait and open. When a process executes a wait command, it will wait at the gate until the gate is opened, after which it may proceed with its execution. If other processes are presently waiting at the gate, this process joins them at the rear of a queue.

When a process executes an open command, it permits the process at the front of the queue to proceed and the rest of the queue to move up one position. After a single process passes through the gate, the gate recloses until another open is executed. If no process is waiting when the open command is reached, no action is taken. The gate simply swings open and then immediately recloses with no process passing through. In any case, the process executing the open command continues execution.

The gate synchronization allows for direct implementation of mutual dependency, with the dependent process executing a wait and the other process executing an open when the required activity is completed. The implementation of mutual exclusion requires the presence of a Boolean primitive that we call waiting, which is true if at least one process is waiting for the gate to open and false if none are waiting. A mutual exclusion can then be implemented by polling gate G and by including the following template in each participating process:

```c
if busy(G) then wait(G);
mutually-exclusive actions;
open(G);
```

As we mentioned in the preceding section, an additional form of synchronization is automatically implemented by the rendezvous version of the phone model of information passing, which requires the sending and receiving processes to synchronize before the message can be sent. We examine this further in Chapter 18, because it is the method used by Ada.

**Research:** At several places in concurrent processing, a queue is used. For example, messages that are passed under the mail model may be queued, and processes waiting to synchronize may be queued. Examine the possibility of making these priority queues, where the originating process assigns a priority to the activity that is queried. How might this be useful? How might this be implemented?

**Discuss:** An additional problem can arise in the token approach to the mutual exclusion problem. If two processes request the token simultaneously and the token is a shared resource, there may be an associated mutual exclusion problem in accessing the token. Describe how this might happen and how it could be prevented.
Discuss: Some languages -- Ada is one of them -- allow parallel features to be used regardless of the type of processor, even on uniprocessors. Discuss how this is possible. What is required of the implementation? Would this be easier if the operating system were multitasking?

Reinforce: Describe how you would simulate the gate approach to synchronization using only the token operations.

Reinforce: Describe how you would simulate the token approach to synchronization using only gate operations.

Reinforce: Divide-and-conquer algorithms can be nicely implemented concurrently. Describe how a concurrent quicksort might be written using any of the models introduced in this chapter.
One way to implement the token model is to have a variable called a semaphore mimic the action of the token. If the variable is 0, any process may take the token; it must make that variable value 1 to indicate the token is in use.

a. Write a program in Concurrent Pascal demonstrating this concept.
b. As mentioned above in the Discussion Questions, the semaphore is itself a shared resource and is subject to the same mutual exclusion problems as other shared resources. Write a program in Concurrent Pascal to illustrate this.
c. Can you write a section of code that guarantees mutual exclusion -- both for a critical section and for semaphores used?

Laboratory: Does the language assigned to you use the token or gate model for synchronization? Write a program to illustrate the other method.

9.7 Concurrency in Java

Java provides concurrency through threads. A thread is a flow of control that is implemented in Java by the Thread class. This section puts the concurrency features of Java threads into the framework of concurrency as introduced in the preceding sections.

9.7.1 Concurrent Units

Java follows the class model for the definition of concurrent units. In particular, a concurrent unit is defined as an object of the Thread class or of some class derived from the Thread class.

Every Thread object can have a run() method defined for its class. That run() method is the concurrent action associated with that thread. There are two ways of defining threads in Java. First, a thread may have its run() method attached to it by a constructor that has as its parameter an object of a class that implements the Runnable interface. The Runnable interface is an interface that requires a run() method, and the run() method for the constructed thread will be the run() method of the Runnable object that is attached through the constructor.

An example of such a Java implementation is found in Figure 10.1. The class Hello is created with a Runnable interface. It includes instance variables name and count, which are given values by its constructor. The run() method prints the hello message repeatedly for a total of count times. The main program in HelloTest constructs two Thread objects that use the Hello
run() method. When the start() message is sent to a Thread object, the Thread object is initialized and then its run() method is called implicitly. The method start() is a member of the Thread class. The HelloTest main method in Figure 9.7 results in the threads one and two running concurrently. After starting threads one and two, main then completes its execution, terminating when its two running threads terminate their execution.

Figure 9.7

```java
public class Hello implements Runnable {
    protected String name;
    protected long count;

    public Hello(String name, long count) {
        this.name = name;
        this.count = count;
    }

    public void run() {
        for (int i=0; i<count; i++)
            System.out.println("Hello from " + name + i);
    }
}

public class HelloTest {
    public static void main(String args[]) {
        Thread one = new Thread(new Hello("one",1000));
        Thread two = new Thread(new Hello("two",5));
        one.start();
        two.start();
        System.out.println("Hello from Main");
    }
}
```

A second approach to defining threads in Java is to define the run() method by deriving a class from class Thread and including a definition for the run() method in that class definition. This approach is illustrated in Figure 9.8. This example duplicates the functionality of Figure 9.7, but creates a new class derived from Thread, HelloThread, with the run() method defined within that class. Note that this approach moves the initialization of the instance variables from the constructor of the Runnable to the constructor of the Thread.

Figure 9.8

```java
public class HelloThread extends Thread {
    protected String name;
    protected long count;

    public HelloThread(String name, long count) {
        this.name = name;
        this.count = count;
    }

    public void run() {
```
for (int i=0; i<count; i++)
    System.out.println("Hello from " + name + i);
}

public class HelloThreadTest {
    public static void main(String args[]) {
        HelloThread one = new HelloThread("one",1000);
        HelloThread two = new HelloThread("two",5);
        one.start();
        two.start();
        System.out.println("Hello from Main");
    }
}

9.7.2 Invocation of Concurrent Units

The preceding section shows that a Java Thread is invoked by calling its run() method indirectly by sending it a start() message. In this section, we look at this invocation process in more detail.

When the start() message is sent to a Thread object, the object initiates a concurrent execution of its run() method. The thread of execution that sends the start() message then continues its execution. The newly started thread will continue execution until its run() method completes or until another message to the Thread object intervenes. There are two ways this intervention can occur.

A Thread object can be halted by a stop() message. Upon receipt of this message, the thread is irrevocably halted. It is also possible to halt a thread temporarily by a suspend() message. A suspended thread can be continued by a resume() message. Figure 9.9 shows examples of stop(), suspend(), and resume() message calls. Threads one and two begin concurrent execution, after which two is temporarily suspended. Thread one is then halted permanently by the stop() message. Following this, two resumes its execution to completion.

Figure 9.9

public class HelloTest3 {
    public static void main(String args[]) {
        Thread one = new Thread(new Hello("one",1000));
        Thread two = new Thread(new Hello("two",1000));
        one.start();
        two.start();
        two.suspend();
        System.out.println("two has been suspended");
        one.stop();
        System.out.println("one has been stopped");
        two.resume();
        System.out.println("two has been resumed");
    }
}
Another message that controls execution of a thread is `sleep()`. The `sleep()` message has a single `long` parameter and causes the receiving thread to suspend for the number of milliseconds specified by the `long` parameter. Any Java thread of execution is suspended for `m` milliseconds when it executes the statement

```
Thread.sleep(m);
```

This is a call to the static method `sleep()` of class `Thread`.

Finally, a thread can suspend itself until another thread is completed. This is done by the message

```
aThread.join();
```

This suspends the thread that executes this call until `aThread` is completed, either through normal termination or through a `stop()` call.

Figure 9.10 gives examples of `sleep()` and `join()` calls. The method `join` can also be called with a single `long` parameter `m`. This causes the executing thread to suspend until `aThread` completes or until it has waited `m` milliseconds, whichever occurs first. The single parameter version of `join` is not illustrated in Figure 9.10.

**Figure 9.10**

```java
public class DrowsyThread extends Thread {
    protected String name;
    protected long count;
    protected long nap;

    public DrowsyThread(String name, long count, long nap) {
        this.name = name;
        this.count = count;
        this.nap = nap;
    }

    public void run() {
        for (int i=0; i<count; i++) {
            try {
                Thread.sleep(nap);
            } catch (Exception ex) {}
            System.out.println(name + ":" + i);
        }
    }
}

public class DrowsyTest {
    public static void main(String args[]) {
        DrowsyThread one = new DrowsyThread("one",5,200);
        DrowsyThread two = new DrowsyThread("two",3,1000);
        one.start();
        two.start();
        try {
            one.join();
        } catch (Exception ex) {}
        System.out.println("end of main");
    }
}
```
9.7.3 Data Sharing

Java threads support data sharing in two ways. First, data can be shared by passing the same data object into a number of threads as a parameter to its constructor. This permits each thread to operate on that object, therefore sharing the data. An example of this type of data sharing is given in Figure 9.11. Here, each thread of class PutTake can put or take data from a common object of the class Vector. Each thread decides on its action by generating a random integer. If the generated integer is even, the thread puts an item, and if it is odd and the Vector is non-empty, the thread takes an item from the Vector. Otherwise, the thread sleeps for 100 milliseconds.

Figure 9.11
import java.util.*;
import java.util.Vector;

public class PutTake extends Thread {
    protected Vector v;
    protected String name;
    protected static Random rng = new Random();

    public PutTake(String name, Vector v) {
        this.name = name;
        this.v = v;
    }

    public void run() {
        for (int i=0; i<100; i++) {
            if (rng.nextInt()%2 == 0) {
                v.addElement(name + i);
                System.out.println("put " + name + i);
            } else if (v.size()>0) {
                System.out.println(name + ":take " + (String)v.firstElement());
                v.removeElementAt(0);
            } else try {sleep(100);} catch (Exception ex) {} 
        }
    }
}

import java.util.Vector;

public class PutTakeTest {
    public static void main(String args[]) {
        Vector v = new Vector();
        PutTake one = new PutTake("one",v);
        PutTake two = new PutTake("two",v);
        PutTake three = new PutTake("three",v);
        one.start();
two.start();
}
The second approach is to pass a second Java thread to a given thread via its constructor parameter, thus giving the constructed thread access to the data space of the passed thread. Figure 9.12 shows a very simple example of this kind of data sharing. Here two Thread classes, First and Second, are defined so that they share a single int data value. The shared data is a member of an instance of Second that is passed through the First constructor as a parameter, giving each First instance a partner instance of Second with which it shares data.

Figure 9.12

class First extends Thread {
    Second partner;

    First(Second partner) {
        this.partner = partner;
    }

    void setValue(int value) {
        partner.setValue(value);
    }

    int getValue() {
        return partner.getValue();
    }
}

class Second extends Thread {
    int value;

    void setValue(int i) {
        value = i;
    }

    int getValue() {
        return value;
    }
}

9.7.4 Synchronization

Synchronization in Java occurs through the presence of a synchronization lock that is associated with each Java object. The lock for an object can be requested by any thread that has access to that object. If the lock is already in the possession of another thread, the requesting thread waits until the lock is made available, at which time it competes for possession with all other threads waiting on the lock. If the lock is available when requested, the requesting thread takes possession of the lock and holds it until the lock is released by the thread.
The synchronization lock for an object may be requested in either of two ways. First, it may be requested for the scope of an executable block. The syntax for such a request is

```java
synchronized (anObject) {
    /* block where lock for anObject is held*/
} // Lock for anObject is released upon exit from the block.
```

The synchronization lock for `anObject` is first requested when the `synchronized` clause is encountered. If the lock is available, the thread proceeds to execute the block. Upon completion of the block, the thread relinquishes the lock. If the lock is not available when the `synchronized` clause is reached, the thread waits until it can obtain the lock before proceeding with the block.

Figure 9.13 is a synchronized version of the `PutTake` class in Figure 9.11. This implementation requires that a thread acquire the synchronization lock for Vector `v` before it performs an operation on `v`. This prevents the occurrence of a “race condition” where two threads perform operations on `v` simultaneously, giving unreliable results.

**Figure 9.13**

```java
import java.util.*;
import java.util.Vector;

public class PutTake extends Thread {
    protected Vector v = new Vector();
    protected String name;
    protected static Random rng = new Random();

    public PutTake(String name, Vector v) {
        this.name = name;
        this.v = v;
    }

    public void run() {
        for (int i=0; i<100; i++) {
            if (rng.nextInt()%2 == 0) {
                synchronized(v) {
                    v.addElement(name + i);
                }
                System.out.println("put " + name + i);
            } else if (v.size()>0) {
                synchronized(v) {
                    System.out.println(name + ":take " + (String)v.firstElement());
                    v.removeElementAt(0);
                }
            } else try {sleep(100);} catch (Exception ex) {} 
        }
    }

    import java.util.Vector;
```
public class PutTakeTest {
    public static void main(String args[]) {
        Vector v = new Vector();
        PutTake one = new PutTake("one", v);
        PutTake two = new PutTake("two", v);
        PutTake three = new PutTake("three", v);
        one.start();
        two.start();
        three.start();
    }
}

The second way a synchronization lock can be requested in Java is by a thread calling a synchronized method. A method is tagged as synchronized by the keyword synchronized appearing in its definition. A synchronized method, when called, requests the lock for the receiving object. If it is available, that lock then belongs to the calling thread for the duration of the execution of the method. When the synchronized method terminates, the lock for the receiving object is made available again. The PutTake version using method synchronization is given in Figure 9.14. Here a new class, DataShare, is provided, and each method call on an object of this class is synchronized, resulting in an implicit request for that object’s synchronization lock.

Figure 9.14

import java.util.*;
import java.util.Random;
import java.util.Vector;

public class PutTake extends Thread {
    protected DataShare ds;
    protected String name;
    protected static Random rng = new Random();

    public PutTake(String name, DataShare ds) {
        this.name = name;
        this.ds = ds;
    }

    public void run() {
        for (int i=0; i<100; i++) {
            if (rng.nextInt()%2 == 0) {
                ds.addElement(name + i);
                System.out.println("put " + name + i);
            } else if (ds.size()>0) {
                System.out.println(name + ":take " + (String)ds.firstElement());
                ds.removeElementAt(0);
            } else try {sleep(100);} catch (Exception ex) {} 
        }
    }
}

import java.util.*;
import java.util.Vector;

public class DataShare {
    protected Vector v = new Vector();

    public synchronized void addElement(Object o) {
        v.addElement(o);
    }

    public synchronized Object firstElement() {
        return v.firstElement();
    }

    public synchronized void removeElementAt(int i) {
        v.removeElementAt(i);
    }

    public synchronized int size() {
        return v.size();
    }
}

public class PutTakeTest {
    public static void main(String args[]) {
        DataShare ds = new DataShare();
        PutTake one = new PutTake("one",ds);
        PutTake two = new PutTake("two",ds);
        PutTake three = new PutTake("three",ds);
        one.start();
        two.start();
        three.start();
    }
}

9.7.5 Interprocess Communication

Java implements interprocess communication through the wait() and notify() methods. The implementation of these methods is integrally related to the synchronization lock, and these methods can be used to implement both the mail and phone models of interprocess communication.

In order to illustrate the rather complex semantics of wait() and notify() in Java, we consider a simple example involving two threads, t1 and t2, and an object x. A summary of this simple interaction is shown in Figure 9.15. Here we assume that thread t1 has somehow obtained the lock for x. At some point, while holding that lock, t1 executes

    x.wait();

The execution of this method causes 1) t1 to relinquish the lock on x, and 2) t1 to enter a sleep state where it will remain until another thread executes x.notify().

We assume here that the notifying thread is t2. Therefore, at some later time t2 obtains the lock on x, since it cannot send x a notify() message otherwise. When t2 executes the call x.notify(), one of the threads that is in a sleep state due to a wait() on object x is awakened.
For our example, we assume that the awakened thread is \( t_1 \), though this may not be the case if more than one thread is waiting on \( x \). When \( t_1 \) is awakened, it immediately and implicitly submits a request for the lock of \( x \), but it does not immediately receive that lock since that lock is still held by \( t_2 \). When \( t_2 \) later relinquishes the lock on \( x \), \( t_1 \) competes for that lock with all the other threads that have active requests. Eventually, \( t_1 \) will receive that lock. At that point, \( t_1 \) resumes execution at the point immediately after the \( x\.wait() \) call.

One other feature of the \( wait() \) method is that a version is provided that has a single \( long \) parameter. This parameter specifies a number of milliseconds after which the waiting thread will awaken in the event that it is not awakened before that time by a \( notify() \) call.

Also, there is a \( notifyAll() \) method that, when called, awakens all threads currently sleeping as a result of a \( wait() \) call on the receiver. Once again, none of these awakened threads will obtain the lock immediately, but they will all compete for it when the thread calling \( notifyAll() \) relinquishes the lock.

**Figure 9.15 Illustration of wait() and notify() calls**

<table>
<thead>
<tr>
<th>x lock belongs to</th>
<th>State of t1</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>executing</td>
</tr>
<tr>
<td>t1 obtains lock for x</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>t1 executing</td>
</tr>
<tr>
<td>t1 executes x.wait()</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>none sleeping</td>
</tr>
<tr>
<td>t2 obtains lock for x</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>t2 sleeping</td>
</tr>
<tr>
<td>t2 executes x.notify()</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>t2 waiting for lock</td>
</tr>
<tr>
<td>t2 relinquishes lock for x</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>none waiting for lock</td>
</tr>
<tr>
<td>t1 obtains lock for x and resumes execution immediately after wait() call</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>t1 executing</td>
</tr>
</tbody>
</table>

Figure 9.16 illustrates a Java simulation of the mail model of interprocess communication. Two Thread classes are shown in this figure, Sender, which sends messages to a MessageBuffer, and Receiver, which retrieves the messages. The mail model requires no synchronization between the Sender and Receiver, so the Sender proceeds to the completion of its send() message as soon as it has placed the message in the MessageBuffer.
Figure 9.16 Sender/Receiver Threads using Mail Model Interprocess Communication

```java
public class Sender extends Thread {
    protected MessageBuffer mb;

    public Sender(MessageBuffer mb) {
        this.mb = mb;
        mb.register(this);
    }

    public void send(String test) {
        mb.setValue(test);
    }

    public void terminate() {
        mb.unregister(this);
    }
}

public class Receiver extends Thread {
    MessageBuffer mb;

    public Receiver(MessageBuffer mb) {
        this.mb = mb;
    }

    public void run() {
        String message;
        while (mb.has_senders() || mb.has_value()) {
            if (mb.has_value()) {
                message = mb.getValue();
                process(message);
            }
        }
    }

    public void process(String m) {
        // code used by Receiver to process the message
    }
}
```

Figure 9.17 illustrates the phone model of interprocess communication. In this case, the Sender thread obtains the lock for test, the String that it is sending, before test is sent. After sending test, Sender sleeps until a notify() for test is executed. The Receiver thread, after obtaining a String from the MessageBuffer, sends notify() to the received message String, reawakening its sending thread. Figure 9.17 illustrates the phone model that does not allow the sending thread to proceed until the receiving thread has finished processing the message. To implement the other model, where the receiver acknowledges as soon as receiving the message
and permits the sending thread to continue executing as soon as that acknowledgement occurs, the run() method of Receiver would be as in Figure 9.18.

Figure 9.17 Phone Model with Acknowledgement after processing

```java
public class Sender extends Thread {
    protected MessageBuffer mb;

    public Sender(MessageBuffer mb) {
        this.mb = mb;
        mb.register(this);
    }

    public void send(String test) {
        synchronized(test) {
            mb.setValue(test);
            try {test.wait();} catch (Exception ex) {}        
        }
    }

    public void terminate() {
        mb.unregister(this);
    }
}

public class Receiver extends Thread {
    MessageBuffer mb;

    public Receiver(MessageBuffer mb) {
        this.mb = mb;
    }

    public void run() {
        String message;
        while (mb.has_senders() || mb.has_value()) {
            if (mb.has_value()) {
                message = mb.getValue();
                process(message);
                synchronized(message) {
                    message.notify();
                }
            }
        }
    }

    public void process(String m) {
        try {Thread.sleep(150);} catch (Exception ex){}        
        System.out.println(m + " received");
    }
}
```
Figure 9.18 Phone Model with acknowledgement before processing

```java
public void run() {
    String message;
    while (mb.has_senders() || mb.has_value()) {
        if (mb.has_value()) {
            message = mb.getValue();
            synchronized(message) {
                message.notify();
            }
            process(message);
        }
    }
}
```

In the case of the examples of Figures 9.16 through 9.18, the MessageBuffer class, though not shown here, would have all of its methods synchronized. This would guarantee that whenever a thread is performing an operation on a MessageBuffer, no other thread will try to operate on the same MessageBuffer while the first operation is in progress. Note that this synchronization will result in the threads obtaining the lock on their MessageBuffers, while the other synchronization we have discussed dealt with locks on the String making up the message itself.

**Discuss:** What types of problems are best solved with a concurrent solution? What types of problems are best solved with a sequential solution?

**Laboratory:** A classic problem in concurrency is the dining philosophers problem, first suggested by E. W. Dijkstra in a paper published in 1971. The problem goes something like this:

Five philosophers wish to philosophize together. As they philosophize, they also must eat to keep their strength up (thinking is a taxing activity). Thus, each goes through a cycle of eating and thinking, continually and asynchronously. Consider a table such as the one shown for the philosophers:
Five bowls and five chopsticks (or, in another version, forks) are arranged alternately around a circular table. Each philosopher has his or her own place at the table. The center bowl of rice is continually replenished. To eat, each philosopher needs two forks. Only the two forks on either side of the bowl may be used. A fork may be used by only one philosopher at a time. The problem is to coordinate the use of the forks so that no philosopher dies of starvation.

This problem neatly exemplifies many problems and solutions found in concurrent situations:

Deadlock can easily occur. If all philosophers pick up the left fork together, they will all wait for their neighbors to relinquish the other fork they require. They will all die waiting.

Starvation can occur. If two philosophers conspire against another, even if cooperation to avoid deadlock exists, philosophers can starve -- not even get the chance to eat.

Mutual exclusion/synchronization is exemplified by the rule that no two philosophers may use a fork together. The one-at-a-time rule requires synchronization.

Implement a solution to the dining philosophers problem.

Laboratory: Another classic problem is the producer-consumer problem. For this problem, two processes, a producer and a consumer, execute concurrently. The producer produces data one element at a time and places each element into a buffer of size n. The consumer retrieves elements one at a time from the buffer in the same order they were placed there by the producer. Implement a solution to this problem in your language and then add a twist: a “bucket-brigade” version. This version has producer a consumer, and an “intermediary” -- a process that consumes from one queue and produces for another, as shown in the following diagram.:

![Diagram of producer, intermediary, and consumer]

Your “producer” may be very simple -- such as producing a sequence of numbers. Your consumer should display the results that are retrieved from the queue. Your intermediary does not need to produce any output, although you will probably need some for debugging purposes.

Chapter 9 Terms
concwncry
multiprogramming
multiprocessing
distributed processing
interprocess communication
remote procedure call
mutual exclusion
mutual dependency