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

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

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

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

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

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

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

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

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

Add a `Crotate` class that rotates an icon.

Add an arrow option to the `Iline` class.

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

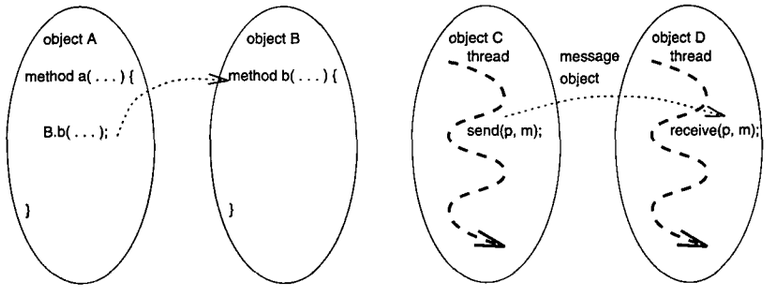
Chapter 6

Message Passing and the Rendezvous

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

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

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



(a) Object A invokes a method in object B. (b) Thread in C sends a message to thread in D.

Figure 6.1: Method Invocation Compared to Message Passing.

how two threads in the same address space or in different address spaces establish a rendezvous and then perform one or more transactions. This is the basis of client-server interaction. A conditional or guarded rendezvous is also implemented. Java's remote method invocation package handles remote procedure calls. A simple example illustrates how to use this package. The chapter concludes with two message passing animations, the distributed dining philosophers and a parallel version of quicksort.

6.1 Message Passing Definitions

Sometimes the phrase "sending a message to an object" is used in object-oriented programming to describe a thread in an object calling or invoking a method in another object. However, this is inaccurate. "Passing" or "sending a message" should be used to describe a thread in one object sending a message to a *thread* executing in another object, where the message itself is an object. Figure 6.1 illustrates the difference.

Primitive operations executed by threads to perform message passing are send(to, message) and receive(from,message), where from and to specify some sort of source and destination addressing, respectively, and message is a reference to the object containing the message. The terms *port*, *channel*, and *mailbox* are used to describe where messages are sent and from where they are received.

Message passing is *blocking* or *nonblocking*, These design choices are also called *synchronous* and *asynchronous*, respectively. If receives are blocking and there is no message available, the receiver blocks until one arrives. If sends are blocking, the sender blocks if there is no receiver ready to receive the message. If sends are nonblocking, a send returns control immediately after the message is queued by the messaging system. If receives are nonblocking, a receive returns a failure indicator if no message is available. Instead of sharing variables, threads in different address spaces share message passing channels implemented by the messaging system. Sending a message is analogous to a thread making a remote assignment of a value to a variable in another address space. If sends are blocking, then the sending thread

delays until the remote assignment is made. Asynchronous message passing is an extension or enhancement of the semaphore P and V operations to include data. In fact, asynchronous message passing of empty messages is equivalent to (potentially remote) semaphore P and V operations.

If both send and receive are blocking, then a *simple rendezvous* occurs when the message is transferred or copied from the sender to the receiver. Both the sending thread and the receiving thread are at known points in their code at the same time and a unidirectional flow of information takes place.

We also have *buffered* and *nonbuffered* design choices. A nonbuffered send blocks until a receiver executes a receive. A buffered send usually implies nonblocking. But there is probably an upper limit on the buffer size above which the sender blocks. There is not much functional difference between blocking buffered and blocking nonbuffered sends.

So we really have three varieties of each. All send/receive design choice combinations below are feasible except 2/3, 3/2, and 3/3. For send,

1. Blocking: waits for receiver or receiver is already waiting;
2. Buffered, nonblocking: message is buffered but the receiver has not necessarily gotten it yet;
3. Nonbuffered, nonblocking: returns an error if no receiver is ready or waiting, returns okay if message is sent and received.

For receive,

1. Blocking: waits for message, i.e., sender to send;
2. Buffered, nonblocking: returns an error if no message is waiting, else returns okay;
3. Nonbuffered, nonblocking: returns an error if no sender is ready or waiting, returns okay if message is sent and received.

Two other design considerations are naming or addressing and the sharing of channels by multiple threads. How does the sending thread address the destination thread? How does the receiving thread specify the desired message source? Instead of the sender and receiver addressing each other directly, a channel, also called a port or mailbox, may be used. The sender and receiver refer to the channel in their message passing operations.

```
shared by sender and receiver:
   channel chan;
sender:
   send(chan, message);
receiver:
   message = receive(chan);
```

client	server
...	...
send (request)	receive (service request)
... [wait]	... [do requested service]
receive (results)	send (back results)
...	...

Figure 6.2: Client-to-Server Remote Procedure Call.

A channel shared by a single sender and a single receiver is called a *one-to-one* communication link between the two threads. If several senders share a channel to communicate to a single receiver, the channel is *many-to-one*. Channels may also be *one-to-many* and *many-to-many*.

An *extended rendezvous* occurs if

- the sender does a blocking send followed by a blocking receive,
- and the receiver does a blocking receive followed by a blocking send.

Both threads are at known points in their code, and a bidirectional flow of information takes place. See Figure 6.2. Typically a *client* thread uses this technique to communicate with a *server* thread and request a service to be performed on its behalf. A similar situation is a *worker* thread contacting a coordinator, administrator, or *master* thread, asking for more work to do.

An extended rendezvous is sometimes called a *remote procedure call* (RPC), particularly if the sender and receiver are on two different machines connected by a LAN and if a new thread is created on the remote machine to execute the code in the remote procedure on behalf of the sender. In this case, tools may be provided to the programmer to make the code for the rendezvous look like a simple procedure call. Such tools are called RPC libraries or packages. For more information, see ([34], Section 16.3.1), ([36], Section 12.5), ([41], Section 10.3), ([42], Section 2.4). The extended rendezvous and remote procedure call are ideal tools for client-server programming. They appear functionally equivalent to the client: the client blocks while the server performs some steps to handle the client's request. The difference is in the implementation on the server side. A rendezvous is accepted by an explicit server thread, programmed by the designer of the server application; the thread waits in a loop for such rendezvous request messages. Whenever a remote procedure call is made by a client, the RPC package or library creates a new thread to execute the server procedure, transparently to the programmer.

6.2 Message Passing in Java

Several forms of message passing are implemented in this book's `Synchronization` package, using the synchronization tools provided by Java: synchronized methods, synchronized blocks, and monitor waiting and signaling. If all the threads are in the same JVM (and necessarily on the same physical machine), a reference or handle to

an object is passed from one thread to the other through a message passing object used as a port, mailbox, or channel. Both synchronous and asynchronous message passing are implemented, as shown in Library Classes 6.1

```
public final class SyncMessagePassing
      extends MessagePassingRoot {
   private Object theMessage = null;
   private final Object sending = new Object();
   private final Object receiving = new Object();
   private final BinarySemaphore senderIn =
      new BinarySemaphore(0);
   private final BinarySemaphore receiverIn =
      new BinarySemaphore(0);
   public final void send(Object m) {
      if (m == null) throw new NullPointerException();
      synchronized (sending) {
         theMessage = m;
         V(senderIn);
         P(receiverIn);
      }
   }
   public final Object receive() {
      Object receivedMessage = null;
      synchronized (receiving) {
         P(senderIn);
         receivedMessage = theMessage;
         V(receiverIn);
      }
      return receivedMessage;
   }
}
```

and 6.2.

```
public final class AsyncMessagePassing
      extends MessagePassingRoot {
   private int numMessages = 0; // negative if waiting receivers
   private final Vector messages = new Vector();
   public final synchronized void send(Object m) {
      if (m == null) throw new NullPointerException();
      numMessages++;
      messages.addElement(m); // at end
      if (numMessages <= 0) notify();  // unblock a receiver
   }
   public final synchronized Object receive() {
      Object receivedMessage = null;
```

```
            numMessages--;
            if (numMessages < 0)
               try { wait(); } catch (InterruptedException e) {}
            receivedMessage = messages.firstElement();
            messages.removeElementAt(0);
            return receivedMessage;
         }
      }
```

The synchronous message passing class uses semaphores to coordinate the transfer of the message from the sender to the receiver, blocking one until the other arrives, then releasing both. Thus, the two threads rendezvous. The asynchronous message passing class uses a `Vector` to retain or buffer sent but not yet received messages; if the counter `numMessages` is negative, then there are receivers waiting for a message.

These classes implement the `MessagePassing` interface and extend the `MessagePassingRoot` class, shown in Library Classes 6.3

```
public interface MessagePassing {
   public abstract void send(Object m)
      throws NotImplementedMethodException;
   public abstract void send(int m)
      throws NotImplementedMethodException ;
   public abstract void send(double m)
      throws NotImplementedMethodException;
   public abstract Object receive()
      throws NotImplementedMethodException;
   public abstract int receiveInt()
      throws ClassCastException, NotImplementedMethodException;
   public abstract double receiveDouble()
      throws ClassCastException, NotImplementedMethodException;
   public abstract void close()
      throws NotImplementedMethodException;
}
```

and 6.4.

```
public abstract class MessagePassingRoot
      implements MessagePassing {
   public abstract void send(Object m)
      throws NotImplementedMethodException;
   public void send(int m) {
      send(this, new Integer(m));
   }
   public void send(double m) {
      send(this, new Double(m));
   }
   public abstract Object receive()
```



```
      throws NotImplementedMethodException;
   public int receiveInt() throws ClassCastException {
      return ((Integer) receive(this)).intValue();
   }
   public double receiveDouble() throws ClassCastException {
      return ((Double) receive(this)).doubleValue();
   }
}
```

The `MessagePassing` interface includes abstract methods for sending and receiving objects and the `int` and `double` base data types. Class `MessagePassingRoot` provides an implementation for sending and receiving `int`s and `double`s as objects using the wrapper classes `Integer` and `Double`. Both the `AsyncMessagePassing` and `SyncMessagePassing` classes provide implementations for sending and receiving objects. Thus, these two classes may also be used to send and receive `int` and `double` messages since they inherit the `MessagePassingRoot` methods. All of the abstract methods in the `MessagePassing` interface throw `NotImplementedMethodException`. A class extending `MessagePassingRoot` should override some of the methods with an implementation that just throws the exception if the class designer chooses to restrict the kinds of messages that may be passed.

The capacity of the asynchronous channel is limited only by the JVM memory since a `Vector` is used. The constructor argument for Library Class 6.5 specifies the amount of buffering, restricting the capacity of the channel. Its implementation uses a circular array of object references to retain unreceived messages.

```
public final class BBMessagePassing
      extends MessagePassingRoot {...
   public final void send(Object value) {
      if (value == null) throw new NullPointerException();
      synchronized (mutexS) {
         P(spaces);
         buffer[putIn] = value;
         putIn = (putIn + 1) % numSlots;
         V(elements);
      }
   }
   public final Object receive() {
      Object value = null;
      synchronized (mutexR) {
         P(elements);
         value = buffer[takeOut];
         takeOut = (takeOut + 1) % numSlots;
         V(spaces);
      }
      return value;
   }
```

```
}
```

If the circular array fills up, senders block. Note the resemblance to the bounded buffer implemented with semaphores and monitors in earlier chapters. The synchronized blocks on objects `mutexS` and `mutexR` are required because this bounded buffer is accessed by multiple producer threads (message senders) and multiple consumer threads (message receivers); see Exercise 4.2.

The input-output streams `PipedInputStream` and `PipedOutputStream` are used to send data values of type `int` and `double` that represent messages through a pipe, as shown in Library Class 6.6. From a pipe created in the class constructor, `DataInputStream` and `DataOutputStream` objects are created whose `readInt` and `writeInt` methods are used to read and write integer messages through the pipe. Messages of type `double` are passed using similar code, not shown.

```
public final class PipedMessagePassing
      extends MessagePassingRoot {...
   public final void send(int m) {
      synchronized (sending) {
         try { outData.writeInt(m); }
         catch (IOException e)
            { throw new MessagePassingException(); }
      }
   }
   public final int receiveInt() {
      int value = 0;
      synchronized (receiving) {
         try { value = inData.readInt(); }
         catch (IOException e)
            { throw new MessagePassingException(); }
      }
      return value;
   }
}
```

Objects may be serialized, that is converted into a stream of bytes, through a pipe, as shown in Library Class 6.7. From a pipe created in the class constructor, `ObjectInputStream` and `ObjectOutputStream` objects are created whose `readObject` and `writeObject` methods are used to read and write serialized object messages through the pipe.

```
public final class ObjPipedMessagePassing
      extends MessagePassingRoot {...
   public final void send(Object m) {
      synchronized (sending) {
         try { outObj.writeObject(m); }
         catch (IOException e)
            { throw new MessagePassingException(); }
```



```
      }
   }
   public final Object receive() {
      Object o = null;
      synchronized (receiving) {
         try { o = inObj.readObject(); }
         catch (Exception e)
            { throw new MessagePassingException(); }
      }
      return o;
   }
}
```

These two classes correspond to sending a message by value rather than by reference. Sending messages is buffered and nonblocking until the pipe fills up. Receiving messages is the blocking kind in these classes.

If the threads are in different JVMs (and possibly on different physical machines), a socket is set up between two threads wishing to communicate. A socket is a communication end point connected to a socket on another machine for two threads to send data back and forth using the TCP/IP networking protocol. See ([39], Chapter 6) for more information. The socket works very much like a pipe. Both Library Classes 6.6 and 6.7 have constructors that use a socket passed as an argument instead of an internal pipe. We will see examples of this in Section 6.3.

Program 6.1 is a simple example of sending and receiving messages that are object references within the same JVM.

```
class Problem { public int x, y;
   public Problem(int x, int y) { this.x = x;  this.y = y; }
}
...
AsyncMessagePassing gs = new AsyncMessagePassing();
SyncMessagePassing ca = new SyncMessagePassing();
...
class GenerateProblem extends MyObject implements Runnable {...
   public void run () {
      nap(1000);
      send(gs, new Problem(1, 2));
      nap(4000);
      int z = ((Integer) receive(ca)).intValue();
   }
}
class ComputeAnswer extends MyObject implements Runnable {...
   public void run () {
      nap(2000);
      Problem p = (Problem) receive(gs);
      nap(1000);
```

```
            send(ca, new Integer(p.x + p.y));
         }
      }
```

If an error occurs, a `MessagePassingException` (Library Class 6.8) is thrown. The two filter classes `MessagePassingSendOnly` (Library Class 6.10) and `MessagePassingReceiveOnly` (Library Class 6.11) may be wrapped around a message passing channel to permit only sending or receiving on that channel. This is done by overriding the restricted method with one that just throws `NotImplementedMethodException` (Library Class 6.9).

6.2.1 Synchronization Using Message Passing

Two threads in different address spaces needing to synchronize or communicate data cannot use semaphores or monitors; they must use message passing. Two threads in the same address space may use message passing as an alternative to semaphores and monitors. Program 6.2, whose threads share an address space, sets up message passing channels used by a producer thread and a consumer thread for synchronization and communication. The consumer thread receives items built by the producer thread through one of the channels, an example of communication. The consumer extracts the item from the message and sends the empty message back to the producer. Since the number of messages is finite, they act like a bounded buffer. The consumer blocks when the buffer is empty and the producer when the buffer is full, an example of synchronization.

```
class Buffer {
   public String who; public double value; public long when;
   public Buffer() { who = null; value = 0.0; when = 0; }
   public String toString() {
      return
         " who="+ who + " value=" + value + " when=" + when;
   }
}
...
MessagePassing mpEmpty = new AsyncMessagePassing();
MessagePassing mpFull = new AsyncMessagePassing();
...
class Producer extends MyObject implements Runnable {...
   public void run() {
      int napping;  double value;
      while (true) {
         napping = 1 + (int) random(pNap);
         nap(napping);
         value = random();
         Buffer buffer = (Buffer) receive(mpEmpty);
         buffer.who = getName();
```



```
         buffer.value = value;
         buffer.when=age();
         send(mpFull, buffer);
      }
   }
}
class Consumer extends MyObject implements Runnable {...
   public void run() {
      int napping;
      while (true) {
         napping = 1 + (int) random(cNap);
         nap(napping);
         Buffer buffer = (Buffer) receive(mpFull);
         buffer.who = null;
         buffer.value = 0.0;
         buffer.when = 0;
         send(mpEmpty, buffer);
      }
   }
}
```

The driver sends a number of empty messages

```
for (int i = 0; i < numSlots; i++) send(mpEmpty, new Buffer());
```

to the producer that represent the empty buffer slots of the initial bounded buffer ([41], Figure 2-15), ([43], Figure 2-15). After filling a slot, the producer sends it to the consumer, who empties it and sends it back to the producer. The output of the first sample run shows the producer thread filling up the buffer and then waiting for a free-slot message before inserting any more items into the buffer. The second sample run shows the consumer emptying the buffer and waiting for new items from the producer. Thus message passing synchronizes the producer and the consumer threads, just as the semaphores `elements` and `spaces` do in Class 4.4.

The next example, Program 6.3, tests each of the message passing channel types we have looked at: synchronous, asynchronous, controlled capacity, piped, and object piped.

```
mp = new AsyncMessagePassing();        // or
mp = new BBMessagePassing(5);          // or
mp = new SyncMessagePassing();         // or
mp = new PipedMessagePassing();        // or
mp = new ObjPipedMessagePassing();
...
class Producer extends MyObject implements Runnable {...
   public void run() {
      double item;
      int napping;
```

```
            while (true) {
               napping = 1 + (int) random(pNap);
               nap(napping);
               item = random();
               send(mp, item);
            }
         }
      }
      class Consumer extends MyObject implements Runnable {
         public void run() {
            double item;
            int napping;
            while (true) {
               napping = 1 + (int) random(cNap);
               nap(napping);
               item = receiveDouble(mp);
            }
         }
      }
```

This program has two types of threads sharing one address space: those that produce work and those that perform or consume the produced work. A producer thread puts some work to be done into a message passing channel that is called a *bag of tasks*. Consumer threads reach into the bag to extract the next piece of work to do. Note that the threads are sharing a many-to-many channel in this example. Such worker threads are called a *worker crew*. If just a single coordinator or administrator thread produces work while many worker threads reach into the bag, this technique is called *master/worker*. The channel used for the bag of tasks is one-to-many. Sometimes the worker threads put partially completed work back into the bag for other workers to do later. We will see an example of a worker crew in Program 6.14 of Section 6.5.

6.2.2 Distributed Mutual Exclusion

The next message passing example is the distributed mutual exclusion algorithm, described in ([6], Chapter 11) and implemented in Program 6.4. See also ([34], Section 18.2.2), ([36], page 554), ([41], Section 11.2.2), ([42], Section 3.2.2). A collection of threads that share memory may use any of the following for mutual exclusion and condition synchronization: the bakery algorithm of Section 3.5, semaphores, monitors, and message passing (messages sent through the shared memory). Suppose, though, that the threads do not share memory but have private memories and are connected to a local area network. Suppose they need mutually exclusive access to some shared resource like a printer or tape drive. If all we have is message passing, can we implement some sort of *distributed mutual exclusion* algorithm? Furthermore, suppose we do not want to use a central server in order to avoid a bottleneck.

To recap, we want to solve the N-thread mutual exclusion problem with an

algorithm that works in a distributed environment and does not involve a central server. We have N nodes connected by a network or point-to-point communication channels, and we assume

- error-free communication between all nodes, that is, no lost or garbled messages;
- messages sometimes arrive in a different order than they were sent; and
- nodes do not fail or halt, either inside or outside their critical sections.

In other words, nodes eventually respond to all query messages sent to them. The basic idea of the algorithm is

```
do forever {
        noncritical section code
        choose a sequence number
        send it to all other nodes
        wait for a reply message from all other nodes
        enter critical section
        postprotocol
}
```

Each of the N nodes has three threads executing concurrently (the three threads are executing on the CPU and memory of the node). One thread executes the above "do forever" loop. Another thread handles requests from other nodes. And the third waits for replies from all other nodes. A node, say node i, sends a reply or acknowledgment message to another node, say node j, that has sent node i a request message. Node i sends the reply immediately if node j has a lower sequence number (higher priority) in its message or if node i is not trying to enter its critical section. Node i defers the reply (until node i gets into and then out of its critical section) if node j has a higher sequence number (lower priority) in its message. Ties are broken by node identifiers. A node chooses its sequence number by adding one to the highest sequence number it has seen so far in incoming messages from other nodes. The following activities occur in each node.

```
execute noncritical section code

preprotocol:
        choose sequence number as highest seen so far + 1
        send it to all other nodes as a request to enter critical section

wait for replies
while waiting,
        reply to other nodes if their sequence number is lower

enter critical section
```

postprotocol:
reply to others (the deferreds) if their sequence number was higher

This algorithm enforces mutual exclusion because a node does not enter its critical section until it receives replies to its request message from all other nodes. There is no deadlock since ties are broken by node identifiers, as shown in Program 6.4. There is no starvation in the absence contention: if none of the other nodes wants to enter its critical section, replies are immediate. There is no starvation in the presence contention: after a node exits its critical section, it chooses a new sequence number the next time it wants to enter its critical section; that number will be higher than those of other contending nodes.

```
private void chooseNumber() {
   P(s); requesting = true; number = highNumber + 1; V(s);
}
private void sendRequest() {
   replyCount = 0;
   for (int j = 0; j < numNodes; j++) if (j != id)
      send(requestChannel[j], new Message(number, id));
}
private void waitForReply() { P(wakeUp); }
private void replyToDeferredNodes() {
   P(s); requesting = false; V(s);
   for (int j = 0; j < numNodes; j++) {
      if (deferred[j]) {
         deferred[j] = false; send(replyChannel[j], id);
      }
   }
}
private void outsideCS() {
   int napping;
   napping = ((int) random(napOutsideCS)) + 1;
   nap(napping);
}
private void insideCS() {
   int napping;
   napping = ((int) random(napInsideCS)) + 1;
   nap(napping);
}
private void main() {                    // node main thread
   while (true) {
      outsideCS();
      chooseNumber();                    // PRE-PROTOCOL
      sendRequest();                     //      "
      waitForReply();                    //      "
      insideCS();
      replyToDeferredNodes();            // POST-PROTOCOL
   }
}
private void handleRequests() {  // thread to handle requests
   while (true) {
      Message m = (Message) receive(requestsToMe);
      int receivedNumber = m.number;
      int receivedID = m.id;
      highNumber = Math.max(highNumber, receivedNumber);
      P(s);
      boolean decideToDefer = requesting
         && (number < receivedNumber
         || (number == receivedNumber && id < receivedID));
      if (decideToDefer) deferred[receivedID] = true;
      else send(replyChannel[receivedID], id);
      V(s);
   }
}
private void handleReplies() {   // thread to handle requests
   while (true) {
      int receivedID = receiveInt(repliesToMe);
      replyCount++;
      if (replyCount == numNodes - 1) V(wakeUp);
   }
}
```


Three features of the algorithm necessary for its correctness are

1. Putting sequence numbers into outgoing request-to-enter messages,
2. Keeping track of the highest sequence number seen in incoming request-to-enter messages, and
3. Choosing a sequence number higher than any seen so far.

Putting sequence numbers into messages is equivalent to running a local clock and time-stamping outgoing messages with the clock's value. Choosing a sequence number for an outgoing message higher than any incoming sequence number seen so far is equivalent to using Lamport's method ([41], Section 11.1.1), ([42], Section 3.1.1) to correct clock skew.

If outgoing messages are not time-stamped, deadlock may result in the distributed mutual exclusion algorithm. If the arrival time of a message is used to determine when a remote node decided it wanted to enter its critical section, two nodes might send request-to-enter messages to each other at about the same time and both think they have priority. The two nodes then deadlock.

If time-stamped messages are used to determine priority but clock skew is not corrected with Lamport's method, then starvation may occur. A node with a faster

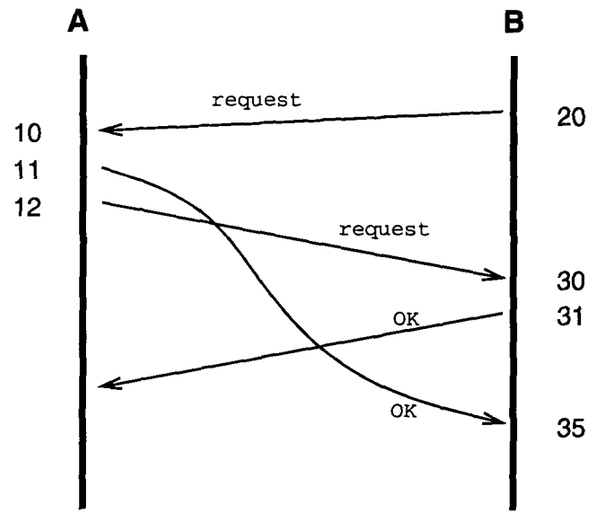


Figure 6.3: Clock Skew Allows Mutual Exclusion Violation.

running clock gets lower priority than one with a slower clock and starves. Worse, mutual exclusion can fail as the example in Figure 6.3 shows. At time 35, node B thinks node A has entered and exited its critical section because B gets an OK message with time stamp 11, causing B to think this is A's deferred reply to B's earlier request-to-enter message. So B enters its critical section. Since A is currently in its critical section, mutual exclusion is violated.

While interesting from a theoretical point of view, this algorithm has several serious practical problems. Many more messages are sent and received for a node to get permission to enter its critical section, compared to a central server algorithm in which one node arbitrates critical section entries for all other nodes. Instead of the single point of failure that a central server algorithm has, the distributed mutual exclusion algorithm described here depends on all of its nodes never crashing.

6.2.3 Conditional Message Passing

In *conditional* message passing, the message remains queued until some condition, specified by the receiver, becomes true. At that time, the message is passed to the receiver, unblocking it. The message may be sent synchronously or asynchronously. A simple conditional message passing example is Program 6.5, a variation of the producers and consumers with bounded buffer.

```
if (synchronous) cmp = new SyncConditionalMessagePassing(true);
else             cmp = new AsyncConditionalMessagePassing(false);
...
```



```
class ConsumerCondition extends MyObject implements Condition {
   private double value = 0;
   public ConsumerCondition(double value)
      { this.value = value; }
   public boolean checkCondition(Object message) {
      return ((Double) message).doubleValue() < value;
   }
}
class Producer extends MyObject implements Runnable {...
   public void run() {
      double item;
      int napping;
      while (true) {
         napping = 1 + (int) random(pNap);
         nap(napping);
         item = random();
         cmp.send(new Double(item));
      }
   }
}
class Consumer extends MyObject implements Runnable {...
   public void run() {
      double item;
      int napping;
      while (true) {
         napping = 1 + (int) random(cNap);
         nap(napping);
         double limit = random();
         item = ((Double) cmp.receive(
            new ConsumerCondition(limit))).doubleValue();
      }
   }
}
```

The consumers are "picky" and accept only producer messages whose data items are smaller than some limit specified as a condition in each consumer receive operation. Section 6.2.4 contains another conditional message passing example.

Program 6.5 tests both the asynchronous (Library Class 6.14) and synchronous (Library Class 6.15) conditional message passing classes. Both of these classes implement the interface in Library Class 6.12.

```
public interface ConditionalMessagePassing {
   public abstract void send(Object message);
   public abstract Object receive(Condition condition);
}
```

The condition object created by the receiver and passed as an argument in its receive request must implement the interface in Library Class 6.13. The condition object must contain a `checkCondition` method used by the channel to determine which messages sent are eligible to be received.

```
public interface Condition {
   public abstract boolean checkCondition(Object m);
}
```

The `AsyncConditionalMessagePassing` class uses three `Vector` queues to hold blocked receivers, their associated conditions, and messages that have been sent but not yet received. Each receive operation activates the private `matchedMessageWithReceiver` method, which scans the queue of sent messages for one meeting the receiver's condition. If one is found, it is returned to the receiver; otherwise the receiver is left blocked. Each send operation also activates the same method to see if the new message meets any blocked receiver's condition. If so, the receiver is unblocked and given the message; if not, the message is added to the queue of unreceived messages. The class starts an internal thread that calls the private method `matchedMessageWithReceiver` on every send and receive.

```
public class AsyncConditionalMessagePassing
      implements ConditionalMessagePassing, Runnable {...
   Vector blockedMessages = new Vector();
   Vector blockedConditions = new Vector();
   Vector blockedReceivers = new Vector();
   Thread me = new Thread(this);
   ...
   private boolean matchedMessageWithReceiver() {
      int numMessages = blockedMessages.size();
      int numReceivers = blockedReceivers.size();
      if (numMessages == 0 || numReceivers == 0) return false;
      for (int i = 0; i < numReceivers; i++) {
         for (int j = 0; j < numMessages; j++) {
            Object m = blockedMessages.elementAt(j);
            Condition c = (Condition)
               blockedConditions.elementAt(i);
            if (c.checkCondition(m)) {
               blockedMessages.removeElementAt(j);
               blockedConditions.removeElementAt(i);
               SyncMessagePassing mp = (SyncMessagePassing)
                  blockedReceivers.elementAt(i);
               blockedReceivers.removeElementAt(i);
               mp.send(m);
               return true;
            }
         }
```



```
      }
      return false;
   }
   public void run() {          // internal thread
      synchronized (me) {
         while (true) {
            while (matchedMessageWithReceiver()) ;
            try { me.wait(); } catch (InterruptedException e) {}
         }
      }
   }
   public void send(Object message) {
      if (message == null) throw new NullPointerException();
      synchronized (me) {
         blockedMessages.addElement(message);
         me.notify();
      }
   }
   public Object receive(Condition condition) {
      if (condition == null) throw new NullPointerException();
      SyncMessagePassing mp = new SyncMessagePassing();
      synchronized (me) {
         blockedConditions.addElement(condition);
         blockedReceivers.addElement(mp);
         me.notify();
      }
      return mp.receive();
   }
}
```

The `SyncConditionalMessagePassing` class is similar; it has an additional queue to hold senders since they must block until a conditional receive operation succeeds.

6.2.4 The Distributed Dining Philosophers

The next example, Program 6.6, uses conditional asynchronous message passing. It solves the distributed dining philosophers problem, implementing the dining philosophers without a central server thread. If there are a large number of philosophers, say millions, then a central server most likely becomes a bottleneck. In the distributed version, each philosopher communicates only with its two neighbors and negotiates the use of a fork with the neighbor sharing that fork.

```
class ServantCondition implements Condition {...
   private boolean hungry = false;
   private boolean dirtyL = false, dirtyR = false;
   ...
```

```
    public boolean checkCondition(Object m) {
       if (m instanceof Hungry) return true;
       else if (!hungry) return true;
       else if (m instanceof PassL || m instanceof PassR)
          return true;
       else if (m instanceof NeedL && dirtyL) return true;
       else if (m instanceof NeedR && dirtyR) return true;
       else return false;
    }
}
class Servant implements Runnable {...
    private BinarySemaphore eat = new BinarySemaphore(0);
    private BinarySemaphore releaseForks =
       new BinarySemaphore(0);
    ...
    public void takeForks(int id)
       { myChannel.send(new Hungry()); P(eat); }
    public void putForks(int id) { V(releaseForks); }
    public void run() {            // servant thread
       Object message = null;
       ServantCondition sc = null;
       boolean hungry = false;
       while (true) {
          sc = new ServantCondition(hungry, dirtyL, dirtyR);
          message = myChannel.receive(sc);
          if (message instanceof Hungry) {
             hungry = true;
             if (!haveR) rightServantChannel.send(new NeedL());
             if (!haveL) leftServantChannel.send(new NeedR());
             while (!(haveR && haveL)) {
                sc =
                   new ServantCondition(hungry, dirtyL, dirtyR);
                message = myChannel.receive(sc);
                if (message instanceof PassL) {
                   haveL = true; dirtyL = false;
                } else if (message instanceof PassR) {
                   // right servant sends fork
                   haveR = true; dirtyR = false;
                } else if (message instanceof NeedL) {
                   haveL = false; dirtyL = false;
                   leftServantChannel.send(new PassR());
                   leftServantChannel.send(new NeedR());
                } else if (message instanceof NeedR) {
                   haveR = false; dirtyR = false;
                   rightServantChannel.send(new PassL());
```



```
                   rightServantChannel.send(new NeedL());
                }
             }
             V(eat); dirtyR = true; dirtyL = true;
             P(releaseForks);
             hungry = false;
          } else if (message instanceof NeedR) {
             haveR = false; dirtyR = false;
             rightServantChannel.send(new PassL());
          } else if (message instanceof NeedL) {
             haveL = false; dirtyL = false;
             leftServantChannel.send(new PassR());
          }
       }
    }
}
```

Each philosopher has a servant thread that does the fork negotiation; the philosophers devote maximal time to philosophizing. In the central server version, the central server keeps track of the forks and hands them out to a hungry philosopher if both its forks are not currently in use. In the distributed version, the servants pass `needL`, `needR`, `passL`, and `passR` messages back and forth. Each fork is always in the possession of some philosopher (or its servant), one of the two on either side of the fork. When a philosopher finishes eating, it labels its two forks as used (soiled or dirty as seen by the other philosophers). As a courtesy to its neighbors, a philosopher (or its servant) wipes clean a soiled fork, one the philosopher was last to use, before passing the fork to a neighbor.

Starvation is prevented by requiring a philosopher to give up, when asked, a fork it holds that is dirty, i.e., that it was the last one to use. The algorithm allows a hungry philosopher to hold onto a clean fork while waiting for its other fork, even if a neighbor requests the fork. Deadlock is prevented by distributing the forks initially in an asymmetric pattern: philosopher 1 gets both forks in the dirty state, philosophers 2 through $n-1$ get one clean fork (the right one) each, and the last philosopher gets no forks. In general, if philosopher p is the most recent philosopher to eat, then it possesses both forks in the dirty state. If a circular chain starts to develop around the table, the chain breaks when it tries to pass through philosopher p. Either philosopher $p-1$ or $p+1$ eats and deadlock does not occur.

6.3 Rendezvous

Synchronous message passing is an example of the rendezvous: two threads reaching specific points in their code at the same time and communicating information. In synchronous message passing, the information flows in one direction only, from sender to receiver, called a simple rendezvous. In the full-fledged general or extended rendezvous, the information flows in both directions.

An extended rendezvous is sometimes called a *remote procedure call* from a client to a server (or a worker to the master) because it resembles a call to a procedure on a remote machine that is executed there. Interface definition language preprocessors make the syntax nearly identical. Typically the call represents a request for service, such as reading a file that resides on the remote machine. The server may handle the request in its main thread or the server may spawn a new thread to handle the request while the server's main thread handles additional requests for service from other clients. The latter gives greater throughput and efficiency since a lengthy request would otherwise delay the handling of requests from the other clients.

An addressing mechanism is needed for the client to contact an appropriate server. In the local case (all threads in the same JVM), an object (Library Class 6.17) is used as the place for the client and server to "meet" and establish a rendezvous.

```
public class EstablishRendezvous {...
   private MessagePassing in = new SyncMessagePassing();
   private ServerSocket serverSocket =
      new ServerSocket(portNum, 50);
   ...
   public Rendezvous serverToClient()
         throws MessagePassingException {
      if (portNum < 0)
         return (Rendezvous) receive(in);      // local case
      Socket socket = null;
      try { socket = serverSocket.accept(); } // remote case
      catch (IOException e)
         { throw new MessagePassingException(); }
      Rendezvous r = new ExtendedRendezvous(socket);
      return r;
   }
   public Rendezvous clientToServer()
         throws MessagePassingException {
      if (portNum < 0) {                       // local case
         Rendezvous r = new ExtendedRendezvous();
         send(in, r);
         return r;
      } else {                                 // remote case
         Socket socket = null;
         try { socket = new Socket(machineName, portNum); }
         catch (IOException e)
            { throw new MessagePassingException(); }
         Rendezvous r = new ExtendedRendezvous(socket);
         return r;
      }
   }
}
```


The server calls method `serverToClient` in this object and blocks until the client calls method `clientToServer`. At this point in time, both methods return a newly created `ExtendedRendezvous` object (Library Class 6.18) that the client and server subsequently use for bidirectional communication.

```
public class ExtendedRendezvous implements Rendezvous {...
   private MessagePassing in = new SyncMessagePassing();
   private MessagePassing out = new SyncMessagePassing();
                        // or
   out = in = new ObjPipedMessagePassing(socket);
   ...
   public Object clientMakeRequestAwaitReply(Object m) {
      send(in, m);  return receive(out);
   }
   public Object serverGetRequest() { return receive(in); }
   public void serverMakeReply(Object m) { send(out, m); }
```

Since the `clientMakeRequestAwaitReply` method is not synchronized, a particular `ExtendedRendezvous` object should be used by only one client thread. This object contains message passing channels shared by the client and server and implements the interface in Library Class 6.16.

```
public interface Rendezvous {
   public abstract Object clientMakeRequestAwaitReply(Object m);
   public abstract Object serverGetRequest();
   public abstract void serverMakeReply(Object m);
}
```

In the local case, one `EstablishRendezvous` object is shared by both client and server. In the remote case (between two threads in different JVMs that might be on different physical machines), a client creates an `EstablishRendezvous` object, passing the server's machine name and TCP/IP port number (see [39]) to the object's constructor; the server also creates an `EstablishRendezvous` object, passing only the port number to the constructor.

When the rendezvous occurs, an `ExtendedRendezvous` object is constructed by `EstablishRendezvous` and returned to both the client and server. In the local case (within the same JVM), the client and server share the `ExtendedRendezvous` object and use it to transact (synchronous message passing of object references). In the remote case (between JVMs), each gets its own `ExtendedRendezvous` object containing a socket (see [39]) to the other JVM (and machine). Objects are serialized through the socket. The case of sending raw data types through a pipe (same JVM) or a socket (different JVMs) is not implemented.

Program 6.7 is a local case example.

```
class Client implements Runnable {...
   private EstablishRendezvous er = // shared with server
   ...
```

```
      public void run() {
         int napping;
         RendezvousRequestReply rrr;
         while (true) {
            napping = 1 + (int)random(napTime);
            nap(napping);
            rrr = new RendezvousRequestReply(getName());
            rrr = (RendezvousRequestReply)
            er.clientToServer().clientMakeRequestAwaitReply(rrr);
         }
      }
   }
   class ServerThread implements Runnable {...
      private Rendezvous r = // from server
      ...
      public void run() {
         RendezvousRequestReply rrr =
            (RendezvousRequestReply) r.serverGetRequest();
         rrr.doRequest();
         r.serverMakeReply(rrr);
      }
   }
   class MultiThreadedServer implements Runnable {...
      private EstablishRendezvous er = // shared with client
      ...
      public void run() {
         while (true) {
            Rendezvous r = er.serverToClient();
            if (threadedServer) {
      // spawn a new thread to handle the request asynchronously
               new ServerThread(r);
            } else {
         // do it here and now before handling any more clients
               RendezvousRequestReply rrr =
                  (RendezvousRequestReply) r.serverGetRequest();
               rrr.doRequest();
               r.serverMakeReply(rrr);
            }
         }
      }
   }
```

The command line option `-t` controls whether or not the server spawns off a new thread to handle the request. The second run in the sample output shows the server can handle more requests per second if it spawns off a new thread to handle each incoming request. Depending on the characteristics of the incoming requests and

the architecture the server is running on, this is generally the case, for example IO-bound requests or multiple CPUs available to the server.

In this local case example, the clients and server share an `EstablishRendezvous` object for addressing. Each time a client wants to rendezvous with the server, it calls the `clientToServer` method to get an `ExtendedRendezvous` object whose `clientMakeRequestAwaitReply` method it uses to transact with the server. The client passes a reference to a `RendezvousRequestReply` object to the server. This object contains the data and a method, `doRequest`, for the server to call. The `ExtendedRendezvous` object is used only once by the client; however, it could be reused for multiple `clientMakeRequestAwaitReply` calls as is done in the next example, the remote dining philosophers.

Program 6.8, the multiple machine dining philosophers, is a remote case example.

```
   class Philosopher implements Runnable {...
      private static Rendezvous r = null;  // one per philosopher
      ...
      public void run() {        // thread for each philosopher
         while (true) {
            think();
            // takeForks(id);
            r.clientMakeRequestAwaitReply(new Integer(id));
            eat();
            // putForks(id);
            r.clientMakeRequestAwaitReply(new Integer(-id-1));
         }
      }
      public static void main(String[] args) {...
         EstablishRendezvous er =
            new EstablishRendezvous(diningServer, port);
         r = er.clientToServer();
         // tell the dining server that this philosopher is alive
         r.clientMakeRequestAwaitReply(new Integer(id));
      }
   }
   class DiningServer implements Runnable {...
      private Rendezvous r = null;  // one per philosopher
      ...
      public void run() {           // one of these threads for
         int message = 0;           // each philosopher
         while (true) {
            message = ((Integer) r.serverGetRequest()).intValue();
            if (message == -philosopherID-1) {
               putForks(philosopherID);
               // acknowledge the release of the forks
               r.serverMakeReply(new Integer(0));
            } else if (message == philosopherID) {
```

```
            takeForks(philosopherID);
            // release the philosopher when forks available
            r.serverMakeReply(new Integer(0));
         }
      }
   }
   public static void main(String[] args) {...
      // accept connections from the philosophers
      // and start a message passing thread for each philosopher
      EstablishRendezvous er = new EstablishRendezvous(port);
      Rendezvous[] r = new Rendezvous[numPhils];
      for (int i = 0; i < numPhils; i++) {
         r[i] = er.serverToClient();
         int id =
            ((Integer) r[i].serverGetRequest()).intValue();
         new DiningServer(id, r[i]);
      }
      // let the philosophers start eating
      for (int i = 0; i < numPhils; i++)
         r[i].serverMakeReply(new Integer(0));
   }
}
```

Suppose there are machines named `bander`, `cheshire`, `humpty`, `queen`, and `king` connected together on a LAN and used as clients in this example, along with a machine named `jubjub` used as the server. The example compile and run in the comment at the end of the program shows how (on UNIX) to run each philosopher in its own JVM on a different physical machine.

```
% java DiningServer -n5 -R20 &
% sleep 5; rsh bander "java Philosopher -s jubjub -i0" &
% sleep 5; rsh cheshire "java Philosopher -s jubjub -i1" &
% sleep 5; rsh humpty "java Philosopher -s jubjub -i2" &
% sleep 5; rsh queen "java Philosopher -s jubjub -i3" &
% sleep 5; rsh king "java Philosopher -s jubjub -i4"
```

Each philosopher sends an `Integer` object containing its *id* value to the server when it is hungry. Since this is a rendezvous, the philosopher is blocked until it gets a reply indicating that its forks are available. The server spawns a new thread for each philosopher to handle the transactions. The server uses semaphores internally to keep track of the forks. Each philosopher sends an `Integer` object containing its $-id-1$ value to indicate it is putting down its forks. Each philosopher has its own `ExtendedRendezvous` object whose `clientMakeRequestAwaitReply` method it calls over and over again (in contrast to the previous example, in which the clients obtained a new `ExtendedRendezvous` object for each transaction with the server).

How much does message passing cost? How much time does it take? Program 6.9 attempts to measure the amount of time the client and server take to transact

Number of Calls	Array Size	Average Time (ms) Per Call: intra-JVM (by reference)	inter-JVM `localhost`	inter-JVM LAN	inter-JVM WAN
10	1	3.0	138.8	2136	1813
10	10	4.9	143.8	1312	1115
10	100	3.8	183.8	2237	2697
10	1000	16.7	628.7	1372	9507

Table 6.1: Communication Times.

in a rendezvous. A client sends a message containing an array of length N to the server. The server adds one to each entry of the array and sends it back. The client does this M times and calculates the number of bytes sent per millisecond. The program is run in two ways. The local run passes the message as a reference from the client to the server within the same JVM. The remote run serializes the message containing the array through a socket over the network to the server running in a different JVM, possibly on another physical machine. Program 6.9 calculates the time it takes to call a method (indirectly via an extended rendezvous) with an object argument in several different situations:

intra-JVM The invoked method is in a class in the same JVM as the calling thread and the object is passed as a reference (JDK 1.1 on Solaris 2.5).

localhost The invoked method is in a class in a different JVM on the same machine as the calling thread; the object is serialized through a socket (JDK 1.1 on Solaris 2.5).

LAN The invoked method is in a class in a different JVM on a different machine connected to the machine of the calling thread by a departmental LAN (standard ethernet); the object is serialized through a socket (JDK 1.1 on Solaris 2.5).

WAN The invoked method is in a class in a different JVM on a different machine connected to the machine of the calling thread by a wide area network (WAN; modem, multiple Internet gateways and routers); the object is serialized through a socket (JDK 1.1 and Windows 95 calling Solaris 2.5).

The object passed as an argument to the invoked method contains an array of double-precision floating-point numbers (Java type `double`). The method adds one to each element of the array. The times in milliseconds are shown in Table 6.1 for four different sizes of the array. Each time is the average of ten trials for over and back serialization (except the intra-JVM case) of the object passed to the method. The times for the LAN and WAN cases vary considerably because of random network delays.

6.3.1 Conditional Rendezvous

Synchronous conditional message passing corresponds to a conditional simple rendezvous, that is, once the condition is met information in the form of a message object flows in one direction from the client to the server. Library Class 6.20 implements an extended *guarded* or *conditional* rendezvous, in which information flows in both directions once a server finds a client message meeting the condition. This class is a combination of `EstablishRendezvous` and `ExtendedRendezvous` with condition checking added; its implementation is very similar to `SyncConditionalMessagePassing`, Library Class 6.15. The client calls the `clientTransactServer` method of the `ConditionalRendezvous` object; the server first calls `serverGetClient` to obtain a client meeting the condition and then uses the `Rendezvous` object returned to interact with the client (`serverGetRequest` and `serverMakeReply` methods). The condition interface, Library Class 6.19, is enhanced with more information passed to the `checkCondition` method, compared to Library Class 6.13, so the condition check can be based on information gathered about all outstanding messages.

```
public interface RendezvousCondition {
/*
 * The information available to the checkCondition method is:
 *   the particular message being evaluated,
 *     blockedMessages.elementAt(messageNum);
 *   the queue of blocked messages itself, blockedMessages;
 *   and the number of blocked servers, numBlockedServers.
 * This is the state of the ConditionalRendezvous object.  The
 * particular message can be checked to see if it meets the
 * condition and this test may involving counting how many
 * blocked messages meet some other criterion and/or the number
 * of blocked servers.
 */
   public abstract boolean checkCondition(int messageNum,
         Vector blockedMessages, int numBlockedServers);
}
```

An example using a condition that depends only on the message passed and not on other queued messages is the dining philosophers classical problem, Class 6.10.

```
class EatCondition implements RendezvousCondition {...
   public boolean checkCondition(int messageNum,
         Vector blockedMessages, int numBlockedServers) {
      Object message = blockedMessages.elementAt(messageNum);
      int id = ((Integer) message).intValue();
      if (id < 0) return true;              // putForks()
      else if (state[left(id)] != EATING
           && state[right(id)] != EATING)
         return true;                       // takeForks()
      else return false;
   }
}
class DiningServer implements Runnable {...
   private ConditionalRendezvous cr =
      new ConditionalRendezvous();
   state = new int[numPhils];
   for (int i = 0; i < numPhils; i++) state[i] = THINKING;
   ...
   public void takeForks(int id) {
      cr.clientTransactServer(new Integer(id));
   }
   public void putForks(int id) {
      cr.clientTransactServer(new Integer(-id-1));
   }
   public void run() {  // makes atomic state changes
      while (true) {
         Rendezvous r = cr.serverGetClient(
            new EatCondition(state, EATING));
         int id = ((Integer) r.serverGetRequest()).intValue();
         if (id < 0) state[-id-1] = THINKING;
         else state[id] = EATING;
         r.serverMakeReply(new Integer(0));
      }
   }
}
```


A particular `ConditionalRendezvous` object can be shared by several servers; multiple calls by servers to the `serverGetClient` method can be outstanding, that is, blocked, waiting for a client message meeting the condition. Furthermore, after calling `serverGetClient` and before completing the rendezvous with that client, a server might call `serverGetClient` again; such a server is handling several rendezvous with different clients simultaneously. A banking example, Program 6.11, illustrates this. A bank with a single account is accessed by a number of threads making deposit and withdrawal requests to the account. The bank is created with an initial balance. The deposit and withdrawal threads make requests for random amounts at random times using the `clientTransactServer` method of a `ConditionalRendezvous` object they share with the bank.

```
class WithdrawCondition implements RendezvousCondition {...
   public boolean checkCondition(int messageNum,
         Vector blockedMessages, int numBlockedServers) {
      Object message = blockedMessages.elementAt(messageNum);
      int amount = ((Integer) message).intValue();
      int size = blockedMessages.size();
      // count number waiting deposits
```

```
      if (allDepositsBeforeAnyWithdrawals) {
         if (amount > 0) return true; // deposit is okay
         int numBlockedDeposits = 0;
         // count number waiting deposits
         for (int i = 0; i < size; i++) {
            Object m = blockedMessages.elementAt(i);
            int a = ((Integer) m).intValue();
            if (a > 0) numBlockedDeposits++;
         }
         if (numBlockedDeposits > 0 /* && amount < 0 */)
            return false;
         else return -amount < balance;
      } else {
   // special case (-1): any deposit or withdrawal is okay
         if (balance < 0) return true;
   // any deposit is okay but withdrawals must pass next test
         else if ( /* balance >= 0 && */ amount > 0)
            return true;
   // if we get here then balance >= 0 and amount <= 0,
   // so a |withdrawal| < balance is okay;
   // but if balance==0 (special flag value) then
   // no withdrawals are allowed
         else return -amount < balance;
      }
   }
}
class Depositor implements Runnable {...
   private ConditionalRendezvous cr = // from driver
   ...
   public void run() {                // thread
      int deposit, napping, put;
      while (true) {
         napping = 1 + (int) random(dNap);
         nap(napping);
         deposit = 1 + (int) random(dNap);
         put = ((Integer) cr.clientTransactServer(
               new Integer(deposit))).intValue();
      }
   }
}
class Withdrawer implements Runnable {...
   private ConditionalRendezvous cr = // from driver
   ...
   public void run() {                // thread
      int withdraw, napping, got;
```



```
      while (true) {
         napping = 1 + (int) random(wNap);
         nap(napping);
         withdraw = 1 + (int) random(wNap);
         got = ((Integer) cr.clientTransactServer(
               new Integer(-withdraw))).intValue();
      }
   }
}
class Bank implements Runnable {...
   private ConditionalRendezvous cr = // from driver
   ...
   public void run() {                // thread
      int balance = initialBalance;
      while (true) {
         if (starvationFree) {
            // take anything
            // (-1 is special flag value for this)
            RendezvousCondition c =
               new WithdrawCondition(-1, false);
            Rendezvous r = cr.serverGetClient(c);
            int amount =
               ((Integer) r.serverGetRequest()).intValue();
            if (amount > 0) {              // deposit
               balance += amount;
               nap(1 + (int) random(1000));
               r.serverMakeReply(new Integer(amount));
            } else if (-amount < balance) {
               // allowed withdrawal
               balance += amount;
               nap(1 + (int) random(1000));
               r.serverMakeReply(new Integer(-amount));
            } else {
               // withdrawal too big; take deposits until okay
               while (-amount >= balance) {
                  // deposits only
                  // (0 is special flag value for this)
                  RendezvousCondition nc =
                     new WithdrawCondition(0, false);
                  Rendezvous nr = cr.serverGetClient(nc);
                  int deposit = ((Integer)
                     nr.serverGetRequest()).intValue();
                  balance += deposit;
                  nap(1 + (int) random(1000));
                  nr.serverMakeReply(new Integer(deposit));
```

```
                }
                balance += amount;
                nap(1 + (int) random(1000));
                r.serverMakeReply(new Integer(-amount));
             }
          } else {
             // take only deposits and allowed withdrawals
             // but a large withdrawal may "starve" in the queue
             RendezvousCondition c =
                new WithdrawCondition(balance,
                   allDepositsBeforeAnyWithdrawals);
             Rendezvous r = cr.serverGetClient(c);
             int amount =
                ((Integer) r.serverGetRequest()).intValue();
             balance += amount;
             nap(1 + (int) random(1000));
             r.serverMakeReply(new Integer(Math.abs(amount)));
          }
       }
    }
}
```

If `starvationFree` is false, a large withdrawal might sit in the queue and starve while smaller withdrawals continue to be processed. In contrast, if `starvationFree` is true (set with the `-s` option), the server (bank thread) accepts only deposits and no withdrawals until the bank balance is large enough to process the waiting large withdrawal. The `-a` option sets the requirement that all queued deposits are processed unconditionally by the bank before any queued withdrawals. In this case, the condition checking depends not only on the message passed but also on the characteristics of the other messages in the queue. Each time the bank calls the `serverGetClient` method, it passes a `WithdrawCondition` object whose `checkCondition` method counts the number of queued deposits. A withdrawal is processed only if this count is zero. The `-B` option adds additional banks to test simultaneously executing servers, making the simulation less realistic in the sense that deposits and withdrawals are handled by the first available bank. How the request is processed depends on the handling bank's balance and not on the balances in the other banks.

Table 6.2 summarizes the message passing and rendezvous techniques we have considered, showing the types of messages passed.

6.4 Remote Method Invocation

Sun Microsystems has added a *remote method invocation* (RMI) package to Java, providing the ability to make remote procedure calls. We used the phrase "remote procedure call" in Section 6.3 to describe an extended rendezvous between two threads in different JVMs, perhaps on different physical machines. Sun's RMI is

Class Constructor	Blocking	Message Type	Medium
`SyncMessagePassing()`	synchronous	object	reference
`BBMessagePassing(int)`	bounded buffer	object	reference
`AsyncMessagePassing()`	asynchronous	object	reference
`PipedMessagePassing()`	asynchronous	`int`, `double`	pipe
`PipedMessagePassing(Socket)`	asynchronous	`int`, `double`	socket
`ObjPipedMessagePassing()`	asynchronous	serialized object	pipe
`ObjPipedMessagePassing(Socket)`	asynchronous	serialized object	socket
`SyncConditionalMessagePassing()`	conditional synchronous	object	reference
`AsyncConditionalMessagePassing()`	conditional asynchronous	object	reference
`ExtendedRendezvous()`	client/server	object	reference
`ExtendedRendezvous(Socket)`	client/server	serialized object	socket
`ConditionalRendezvous()`	conditional client/server	object	reference

Table 6.2: Message Passing Classes.

similar: a collection of classes, the `java.rmi` package, that allows a thread in one JVM to invoke (call) a method in an object in another JVM, perhaps on a different physical machine. A new thread is created in the other (remote) JVM to execute the called method. Parameters to the remote method and the method's return result, if any, are passed from one JVM to the other using object serialization (object streams) over the network.

Using RMI, a Java programmer writes *distributed computing* applications. Program 6.12 is an example.

```
public interface Compute extends Remote {
   public abstract Work compute(Work w) throws RemoteException;
}
class Work implements Serializable {...
   private final int N = 5;
   private double[]
      a = new double[N], b = new double[N], c = new double[N];
   for (int i = 0; i < N; i++)
      { a[i] = random(-N, N); b[i] = random(-N, N); }
   ...
   public void doWork() {
      // simulate some computation time
      nap(1+(int)random(1000*N));
      for (int i = 0; i < N; i++) c[i] = a[i] + b[i];
   }
```

```
}
class ComputeServer extends UnicastRemoteObject
      implements Compute {...
   public Work compute(Work w) throws RemoteException {
      w.doWork();
      return w;
   }
   public static void main(String args[])  {
      System.setSecurityManager(new RMISecurityManager());
      try {    // register this server
         ComputeServer server = new ComputeServer(serverName);
         Naming.bind("rmi://" + serverMachine + "/"
            + serverName, server);
      } catch (Exception e) {}
   }
}
class Client extends MyObject implements Runnable {
   private Compute server = // from main()
   ...
   public void run() {
      Work w = null;
      while (true) {
         nap(1 + (int) random(napTime));
         w = new Work(getName());
         try { w = server.compute(w); } catch (Exception e) {}
      }
   }
   public static void main(String[] args) {
      Compute server = null;
      try { server = (Compute)
         Naming.lookup("rmi://" + serverMachine + "/"
            + serverName);
      } catch (Exception e) {}
      for (int i = 0; i < numClients; i++)
         new Client(i, server, 1000*napTime);
   }
}
```

Suppose a client on one machine wants to send two vectors to be added to a server on another machine. Presumably the server executes on a computer architecture that performs the operation more quickly and efficiently. The client is thus using the remote server to have work performed on its behalf (adding vectors). The two vectors are placed in a `Work` object by the client; the server invokes the object's `doWork` method to perform the vector addition. The server is accessed through a `Compute` interface known to the client and implemented by the server. Such interfaces must extend `Remote` from the `java.rmi` package. In our example, the `Compute` interface contains a single method, `compute`, that takes a `Work` object parameter and returns a `Work` object result. All such interface methods must have a `throws RemoteException` clause. Note that the `Work` class implements the `Serializable` interface, required of objects that are serialized through the network as part of a remote method invocation. Library Class 3.1, `MyObject`, must also implement `Serializable` since the `Work` class extends it.

The `ComputeServer` remote server object implements the `Compute` interface and extends the `UnicastRemoteObject` class from the `java.rmi.server` package. The server registers itself with the `rmiregistry` program by calling the `Naming.bind` method from the `java.rmi` package, passing a string that resembles a URL and shows where the server is. (A URL is a World Wide Web uniform resource locator, for example `http://...`, `ftp://...`, and `mailto:....`) This URL optionally specifies an IP port number to be used by the client and server. The client accesses the server by calling `Naming.lookup`, passing the server's URL; this call returns a reference to an object on the client side that implements the `Compute` interface. Each time the client calls this object's `compute` method, the `Work` object passed as the method's parameter is serialized through the network to the server side. A server thread is created to call `compute`, which then calls the `Work` object's `doWork` method. The object, now containing the sum of the two vectors, is serialized back to the client.

The sample run attached as a comment to the end of Program 6.12 was produced by executing the client and server on two Sun workstations (named `cheshire` and `jubjub`, respectively) running Solaris 2.5 and connected by a LAN, with port 7777 specified by the user. In a directory accessible to both the client and server, we compile the program, create the "stubs" with the `rmic` command, and start the RMI registry program. For security reasons, Sun requires that the server and registry program be on the same machine.

```
% javac Compute.java
% rmic ComputeServer
% rmiregistry 7777 &
```

Then we start the server and the client.

```
% java ComputeServer -M jubjub:7777 &
% rsh cheshire "java Client -M jubjub:7777 -R10"
```

6.5 More Animation with Java

We conclude this chapter with animations of two message passing programs. Program 6.13 is an enhanced version of the distributed dining philosophers, Program 6.6; statements are added to drive the animation interpreter. The philosophers are represented by circles equally spaced around a large circle representing the table. A bowl of spaghetti rests in the center of the table, represented by a half-tone orange circle. Forks are represented with thick lines. The state of each philosopher is indicated by a different color: outline black for thinking, green for hungry, and

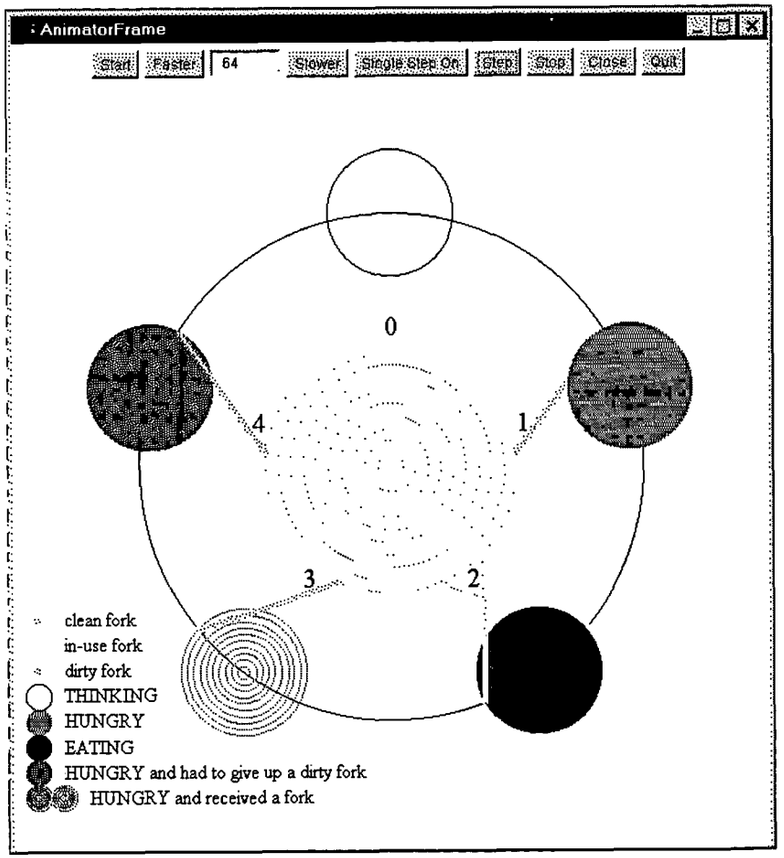


Figure 6.4: Animation Snapshot of the Distributed Dining Philosophers.

blue for eating. Forks used by a philosopher to eat are yellow; when the philosopher finishes eating, the forks are dirty and become orange. Each fork moves back and forth between the two philosophers who use it to eat. See the comments in the program for an explanation of the animation statements. Figure 6.4 is a snapshot of the window during the animation.

The quicksort algorithm is parallelized for a shared memory multiple CPU machine by dedicating to a worker thread each CPU allocated to the program and using a message passing channel as a bag of tasks.

```
AsyncMessagePassing task = new AsyncMessagePassing();
```

The `main` method of Program 6.14 puts the whole array to be sorted into the bag. A worker extracts a task (a segment of the array) from the bag,

```
while (true) {
```



```
    m = (Task) receive(task);
    quickSort(id, m.left, m.right);
}
```

chooses a pivot point, and partitions the array segment. Each of the two resulting partitions is put back into the bag for one of the workers to extract later.

```
if (right-(l+1) > 0) send(task, new Task(l+1, right));
if ((l-1)-left > 0) send(task, new Task(left, l-1));
```

This is an example of the worker crew paradigm, first described in Section 6.2.1. Even though message passing is used for the bag of tasks, this program requires shared memory because the array is sorted "in place" and the work requests put into the bag are array index pairs and not copies of parts of the array. During the animation, each array segment extracted by a worker thread is enclosed in a box and the circles representing the array values in that segment are colored to match the worker thread partitioning the segment. Figure 6.5 is a snapshot of an animation of 100 random numbers between 1 and 10,000 sorted by six threads. After watching several of these animations, we see the effects of a poor pivot value and the reason the algorithm has limited speedup.

Summary

Threads in different address spaces that want to synchronize and communicate cannot use semaphores and monitors. Instead they use message passing and two tools based on message passing, the rendezvous and the remote procedure call. We looked at two design parameters for the send and receive message passing primitives: blocking versus nonblocking and buffered versus nonbuffered. We said two threads participate in a simple rendezvous when one executes a blocking send, the other a blocking receive, and a message is sent.

The `Synchronization` package includes a variety of message passing classes for Java programs, in particular, synchronous, asynchronous, and their conditional versions. In conditional message passing, a message must satisfy a condition specified by the receiver to be acceptable. Messages may be integers, floating-point numbers, or objects. The latter are passed by reference within one JVM and serialized (passed by value) between JVMs, using a pair of sockets. Two major examples illustrated message passing in Java, distributed mutual exclusion and the distributed dining philosophers.

An extended rendezvous is a pair of messages between two threads, a request for service by a client and a reply by a server, performed with send/receive and receive/send in the two threads. The client and server thus carry out a transaction. The rendezvous is implemented in the `Synchronization` package with two classes, one for addressing and one for communicating. If the client and server execute in different JVMs on different machines, the addressing mechanism uses an IP address/port number to set up a socket pair through which serialized object messages are passed. The remote dining philosophers example showed how to start each philosopher on

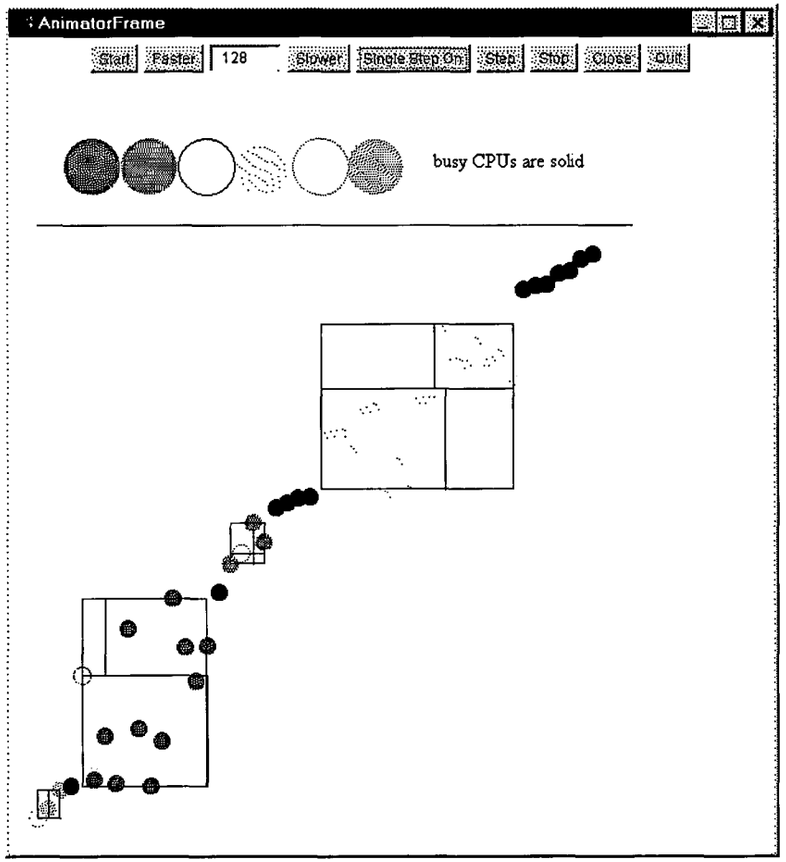


Figure 6.5: Animation Snapshot of Quicksort.

its own workstation. The philosophers communicate with the dining server over the LAN connecting the machines. A conditional version of the rendezvous allows servers to specify the kinds of transactions they are willing to accept.

A remote procedure call is a transaction or extended rendezvous that appears to the programmer as a local procedure call. Sun Microsystems has added a remote procedure call package, called RMI, to the JDK that allows a thread in one JVM to invoke a method in an object in a different JVM, perhaps on a different machine. Transparently to the programmer, a new thread is spawned in the remote JVM to execute the method invoked by the local thread.

The last examples in the chapter are two animations, the distributed dining philosophers and parallel quicksort. In the former, forks move back and forth between the philosophers in response to their messages negotiating the use of shared forks. Two or more forks can move concurrently, a feature of the animation. The

quicksort algorithm is parallelized for a shared-memory multiprocessor with a worker crew, one worker per CPU allocated to the program. The bag of tasks is implemented with a message passing channel.

6.6 Exercises

1. **Send/Receive Design Choices.** Section 6.1 gives three design choices for send and three for receive. Explain why combinations 2/3, 3/2, and 3/3 are not feasible.

2. **Message Passing Bounded Buffer.** Will Program 6.2 work correctly if there are multiple producer threads and multiple consumer threads?

3. **Distributed Bag of Tasks.** Modify Program 6.3 so that all producer threads execute on one machine and all consumer threads execute on another machine. Recall that classes `PipedMessagePassing` (Library Class 6.6) and `ObjPipedMessagePassing` (Library Class 6.7) have constructors with a socket argument. You will need to borrow some code from the second and third constructors of Library Class 6.17, `EstablishRendezvous`.

Do you think it is possible to modify Program 6.3 so that each thread, producer or consumer, executes on its own machine? Why, or why not?

4. **Distributed Mutual Exclusion.** Answer the following questions (based on [6], page 123).

   a. Program 6.4 has a binary semaphore `s`, local to each node. Why is it necessary?

   b. Suppose the binary semaphore `s` is deleted from method `replyToDeferredNodes` and the `V(s)` in thread `handleRequests` is moved up to just before the `if` statement. Show that deadlock is possible.

   c. Does the `V(s)` in `replyToDeferredNodes` need to be moved to the end of the method? In other words, why have a `P(s)...V(s)` bracket the single assignment statement `requesting = false`? Where is the race condition?

   d. Suppose node m decides to enter its critical section and sends a request message to node n. Node n sends a reply message to node m indicating that node n does not want to enter its critical section. Suppose further that node n later decides to enter its critical section and sends a request message to node m (and to all the other nodes). What happens if node n's request message is received by node m before the earlier-sent reply message of node n to node m?

5. **Distributed Dining Philosophers.** After reading Section 6.2.4, answer the following questions.

   a. Does the solution to the dining philosophers shown in Program 6.6 have the desirable "maximal parallelism" property?

   b. Show that deadlock is possible if philosopher 1 is initially given both forks in the clean state (that is, all forks are handed out initially clean).

c. Does the program work correctly if synchronous conditional message passing is used instead of asynchronous?

6. **The Classical Problems Using Rendezvous.** Implement the bounded buffer multiple producers and consumers with a (possibly conditional) rendezvous.

In the exercises for Chapters 4 and 5, you implemented starvation-free versions of the dining philosophers and database readers and writers. Add code to the dining philosophers conditional rendezvous, Class 6.10, to prevent starvation. Implement the database readers and writers with a (possibly conditional) rendezvous so it is fair to both readers and writers; that is, neither readers nor writers starve. Pick either strict serialization or platooning as your strategy.

First, take the version that allows starvation, seed the random number generator with a constant, and determine parameters for the command line that show starvation occurring in the simulation. With the random number generator seeded and the random numbers reproducible, you will have an easier time debugging. Then modify the code to prevent starvation.

7. **Fair Baboons.** Write a (possibly conditional) rendezvous version of the fair baboons program done in Exercise 4.7. Pick either strict serialization or platooning as your strategy.

8. **Sleeping Barbers.** Write a (possibly conditional) rendezvous version of the sleeping barbers program done in Exercise 4.4.

9. **Fraternity Party.** Write a (possibly conditional) rendezvous version of the fraternity party program done in Exercise 4.8.

10. **Bakery.** Write a (possibly conditional) rendezvous version of the bakery program done in Exercise 4.3.

11. **Banker's Algorithm.** Write a Java program that simulates a computing system in which there are n resource types and $E = (e_1, e_2, \ldots, e_n)$ existing resources of each type. There are m threads in the system. To apply the Banker's Algorithm ([15], Section 6.8), ([34], Section 7.5), ([36], Section 5.4), ([41], Section 6.5), ([43], Section 3.3.6), we must know the total demands of the threads in advance, and this information is in the m by n matrix T. Your program simulates the threads requesting and returning resources from a central server thread. In order to avoid deadlock, your program needs to compute (using the Banker's Algorithm) if granting the request keeps the system in a safe state. Use message passing and/or the rendezvous.

When a thread submits a request for resources, $R = (r_1, r_2, \ldots, r_n)$, i.e., it wants r_i units of type i, the central server satisfies the request atomically, that is, the central server allocates either all or none of the requested resources to the thread, depending on whether or not the resulting state is safe. If the state would be unsafe, the thread blocks and the central server remembers the request. Also, a request by a thread for more resources than are currently available must be queued by the central server

(the thread must block). Requests by a thread for more resources than exist or for more resources than its maximum demand aborts the thread.

Threads also release resources. If threads are blocked with outstanding requests, the freed resources may be used to satisfy outstanding requests, but only if the resulting state is safe.

Note that starvation is possible in this program. It could happen like this. You are keeping a FCFS queue of threads whose requests cannot be granted because the resulting state would be unsafe or the resources are currently not available. As other threads release resources, the central server scans the queue of blocked threads and grants resources to those for which it is safe. The problem is that threads possibly starve if they are repeatedly passed over as the central server scans the queue.

If, while trying to decide if a state is safe or not, you finish off as many threads as possible one by one, and there are some left, you do not have to backtrack and try a different order. In other words, one order of finishing off threads is good enough: either they all finish and the state is safe, or you end up with some potentially deadlocked ones, and the state is unsafe.

In order to test your program thoroughly, you need to drive the threads with input scripts. Use reasonable values for m, n, E, and T. Each thread has its own input file. The input data for each thread consist of lines of the following form.

napTime requestReleaseFlag $r_1\, r_2 \ldots r_n$

A thread reads lines from its input file and first naps for the napping time input, then sends a request or release message to the central server (Banker) for resources $(r_1, r_2, \ldots, r_n)$. The thread then blocks until its request is granted (no need to block for a release of resources).

12. **Message Passing Empty Method.** Add a

```
public boolean empty() {...}
```

method to each of the message passing classes. It returns true if no messages are queued, waiting to be received, and false otherwise. Do you think this is a good idea? Why, or why not? Add

```
protected static final boolean empty(MessagePassing mp) {
   return mp.empty();
}
```

to Library Class 3.1, `MyObject`. Design and implement a test program.

13. **Conditional Message Passing.** Enhance the `ConditionalMessagePassing` classes (Library Classes 6.14 and 6.15) by adding constructors with a socket parameter so that two threads in different JVMs, perhaps on different physical machines, can use a socket to perform conditional message passing.

When either the `send` or `receive` method is invoked, the thread executing the `run` method awakes and checks all queued message objects from senders with all queued condition objects from blocked receivers, looking for a match. A more efficient approach when `receive` is called is to check just the receiver's condition with all queued messages and when `send` is called just the sender's message with all queued conditions. Implement this performance improvement.

14. **Sharing `EstablishRendezvous` Objects.** Modify Program 6.7 so that there is a fixed-size pool of server threads operating independently instead of a single server thread (that optionally spawns off a new thread to handle each client request). The servers share a single `EstablishRendezvous` object. Then modify the program so that the client threads execute on one machine and the server threads execute on another machine.

15. **Conditional Rendezvous.** Enhance the `ConditionalRendezvous` class (Library Class 6.20) by adding a constructor with a socket parameter so that two threads in different JVMs, perhaps on different physical machines, can use a socket to perform conditional rendezvous.

16. **Remote Method Invocation.** Convert one of the monitor or semaphore classical problem solutions (bounded buffer producer and consumer, dining philosophers, database readers and writers, sleeping barbers) from local method calls (all in one JVM) to inter-JVM method calls (RMI).

Chapter 7

Parallel Computing

So far, most of our Java examples deal with race conditions, critical sections, mutual exclusion, thread synchronization, and interthread communication in shared and distributed memory platforms. In this book, the term "concurrent programming" is used to describe these types of programs. Although there are no standard definitions for concurrent programming and parallel processing, we use the latter to describe programs that perform numerically intensive computations using a large number of processors. Specialized parallel architectures, other than uniform memory access (UMA) multiple-instruction multiple-data (MIMD) shared-memory multiprocessors and clusters of workstations on a LAN, may be involved, such as nonuniform memory access (NUMA) multiprocessors, single-instruction multiple-data (SIMD) CM-2 Connection Machines, Cray vector supercomputers, and systolic arrays. The amount of synchronization required is closely related to the problem being solved and the hardware involved. Synchronization is automatic and fine-grained when done by the hardware, as on an SIMD machine. It is infrequent and done in software in coarse-grained MIMD algorithms on shared-memory multiprocessors. Concurrent programming is more concerned with resource allocation and avoiding deadlock and starvation. Parallel processing is more concerned with distributing the work to be done across the available processing units.

This chapter is a brief introduction to the parallel computing capabilities of Java. There are many books on parallel computer architectures and parallel programming. See, for example, [10, 30, 32]. We have seen one example of parallelizing an algorithm to use multiple CPUs: the animated quicksort in Program 6.14. That algorithm requires shared memory since the array is sorted "in place" and needs to be accessed by all the worker threads. A bag of tasks implemented with a message passing channel is used to distribute the work. In this chapter, we look at additional examples of parallel processing programs. Some require shared memory; others use message passing in a distributed memory architecture. Some parcel out the computational work to a fixed number of threads, one per CPU allocated to the program; others create a varying number of threads based on the input data. We will see a variety of message passing channel connection patterns among the threads.