Operating systems must provide a mechanism for parent processes to create new child processes. The parent may wait for its children to terminate before proceeding, or the parent and children may execute concurrently. There are several reasons for allowing concurrent execution: information sharing, computation speedup, modularity, and convenience.

The processes executing in the operating system may be either independent processes or cooperating processes. Cooperating processes require an interprocess communication mechanism to communicate with each other. Principally, communication is achieved through two schemes: shared memory and message passing. The shared-memory method requires communicating processes to share some variables. The processes are expected to exchange information through the use of these shared variables. In a shared-memory system, the responsibility for providing communication rests with the application programmers; the operating system needs to provide only the shared memory. The message-passing method allows the processes to exchange messages. The responsibility for providing communication may rest with the operating system itself. These two schemes are not mutually exclusive and can be used simultaneously within a single operating system.

Communication in client–server systems may use (1) sockets, (2) remote procedure calls (RPCs), or (3) Java's remote method invocation (RMI). A socket is defined as an endpoint for communication. A connection between a pair of applications consists of a pair of sockets, one at each end of the communication channel. RPCs are another form of distributed communication. An RPC occurs when a process (or thread) calls a procedure on a remote application. RMI is the Java version of RPCs. RMI allows a thread to invoke a method on a remote object just as it would invoke a method on a local object. The primary distinction between RPCs and RMI is that in RPCs data are passed to a remote procedure in the form of an ordinary data structure, whereas RMI allows objects to be passed in remote method calls.

Exercises

- 3.1 Describe the differences among short-term, medium-term, and long-term scheduling.
- 3.2 Describe the actions taken by a kernel to context-switch between processes.
- 3.3 Consider the RPC mechanism. Describe the undesirable consequences that could arise from not enforcing either the "at most once" or "exactly once" semantic. Describe possible uses for a mechanism that has neither of these guarantees.
- 3.4 Using the program shown in Figure 3.34, explain what will be output at `Line A`.
- 3.5 What are the benefits and the disadvantages of each of the following? Consider both the system level and the programmer level.
  a. Synchronous and asynchronous communication
  b. Automatic and explicit buffering

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
pid_t pid;

  pid = fork();

  if (pid == 0) { /* child process */
    value += 15;
  }
  else if (pid > 0) { /* parent process */
    wait(NULL);
    printf("PARENT: value = %d",value); /* LINE A */
    exit(0);
  }
}
```

Figure 3.34 C program.

c. Send by copy and send by reference

d. Fixed-sized and variable-sized messages

3.6 The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, ... Formally, it can be expressed as:

$$fib_0 = 0$$
$$fib_1 = 1$$
$$fib_n = fib_{n-1} + fib_{n-2}$$

Write a C program using the `fork()` system call that generates the Fibonacci sequence in the child process. The number of the sequence will be provided in the command line. For example, if 5 is provided, the first five numbers in the Fibonacci sequence will be output by the child process. Because the parent and child processes have their own copies of the data, it will be necessary for the child to output the sequence. Have the parent invoke the `wait()` call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a non-negative number is passed on the command line.

3.7 Repeat the preceding exercise, this time using the `CreateProcess()` function in the Win32 API. In this instance, you will need to specify a separate program to be invoked from `CreateProcess()`. It is this separate program that will run as a child process and will output the Fibonacci sequence. Perform necessary error checking to ensure that a non-negative number is passed on the command line.

3.8 Modify the date server shown in Figure 3.26 so that it delivers random fortunes rather than the current date. Allow the fortunes to contain multiple lines. The date client shown in Figure 3.27 can be used to read the multi-line fortunes returned by the fortune server.

3.9 An **echo server** is a server that echoes back whatever it receives from a client. For example, if a client sends the server the string *Hello there!* the server will respond with the exact data it received from the client—that is, *Hello there!*

Write an echo server using the Java networking API described in Section 3.6.1. This server will wait for a client connection using the `accept()` method. When a client connection is received, the server will loop, performing the following steps:

- Read data from the socket into a buffer.
- Write the contents of the buffer back to the client.

The server will break out of the loop only when it has determined that the client has closed the connection.

The date server is shown in Figure 3.26 uses the class `java.io.BufferedReader`. `BufferedReader` extends the class `java.io.Reader`, which is used for reading character streams. However, the echo server cannot guarantee that it will read characters from clients; it may receive binary data as well. The class `java.io.InputStream` deals with data at the byte level rather than the character level. Thus, this echo server must use an object that extends `java.io.InputStream`. The `read()` method in the `java.io.InputStream` class returns `-1` when the client has closed its end of the socket connection.

3.10 Write an RMI application in which the server delivers random one-line fortunes. The interface for the remote object appears as

```
import java.rmi.*;

public interface RemoteFortune extends Remote
{
  public abstract String getFortune()
    throws RemoteException;
}
```

A client invoking the `getFortune()` method will receive a random one-line fortune from the remote object.

Project—Creating a Shell Interface

This project consists of modifying a Java program so that it serves as a shell interface that accepts user commands and then executes each command in a separate process external to the Java virtual machine. A shell interface provides the user with a prompt, after which the user enters the next command. The example below illustrates the prompt `jsh>` and the user's next command: `cat`