#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;
        return 0;
    }
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = %d", value); /* LINE A */
        return 0;
    }
}

Figure 3.30 What output will be at Line A?

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) pipes. 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. Pipes provide a relatively simple ways for processes to communicate with one another. Ordinary pipes allow communication between parent and child processes, while named pipes permit unrelated processes to communicate.

Practice Exercises

3.1 Using the program shown in Figure 3.30, explain what the output will be at LINE A.

3.2 Including the initial parent process, how many processes are created by the program shown in Figure 3.31?
Chapter 3 Processes

#include <stdio.h>
#include <unistd.h>

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}

Figure 3.31 How many processes are created?

3.3 Original versions of Apple’s mobile iOS operating system provided no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system.

3.4 The Sun UltraSPARC processor has multiple register sets. Describe what happens when a context switch occurs if the new context is already loaded into one of the register sets. What happens if the new context is in memory rather than in a register set and all the register sets are in use?

3.5 When a process creates a new process using the fork() operation, which of the following states is shared between the parent process and the child process?
   a. Stack
   b. Heap
   c. Shared memory segments

3.6 Consider the “exactly once” semantic with respect to the RPC mechanism. Does the algorithm for implementing this semantic execute correctly even if the ACK message sent back to the client is lost due to a network problem? Describe the sequence of messages, and discuss whether “exactly once” is still preserved.

3.7 Assume that a distributed system is susceptible to server failure. What mechanisms would be required to guarantee the “exactly once” semantic for execution of RPCs?

Exercises

3.8 Describe the differences among short-term, medium-term, and long-term scheduling.
#include <stdio.h>
#include <unistd.h>

int main()
{
    int i;
    
    for (i = 0; i < 4; i++)
        fork();

    return 0;
}

Figure 3.32 How many processes are created?

3.9 Describe the actions taken by a kernel to context-switch between processes.

3.10 Construct a process tree similar to Figure 3.8. To obtain process information for the UNIX or Linux system, use the command ps -ael.

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

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execvp("/bin/ls", "ls", NULL);
        printf("LINE J");
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}

Figure 3.33 When will LINE J be reached?
Chapter 3  Processes

Use the command man ps to get more information about the ps command. The task manager on Windows systems does not provide the parent process ID, but the process monitor tool, available from technet.microsoft.com, provides a process-tree tool.

3.11 Explain the role of the init process on UNIX and Linux systems in regard to process termination.

3.12 Including the initial parent process, how many processes are created by the program shown in Figure 3.32?

3.13 Explain the circumstances under which the line of code marked printf("LINE J") in Figure 3.33 will be reached.

3.14 Using the program in Figure 3.34, identify the values of pid at lines A, B, C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

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

int main()
{
    pid_t pid, pid1;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        pid1 = getpid();
        printf("child: pid = \%d", pid); /* A */
        printf("child: pid1 = \%d", pid1); /* B */
    }
    else { /* parent process */
        pid1 = getpid();
        printf("parent: pid = \%d", pid); /* C */
        printf("parent: pid1 = \%d", pid1); /* D */
        wait(NULL);
    }

    return 0;
}
```

Figure 3.34 What are the pid values?
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

#define SIZE 5

int nums[SIZE] = {0, 1, 2, 3, 4};

int main()
{
    int i;
    pid_t pid;

    pid = fork();

    if (pid == 0) {
        for (i = 0; i < SIZE; i++) {
            nums[i] = -i;
            printf("CHILD: %d ", nums[i]); /* LINE X */
        }
    } else if (pid > 0) {
        wait(NULL);
        for (i = 0; i < SIZE; i++)
            printf("PARENT: %d ", nums[i]); /* LINE Y */
    }

    return 0;
}

**Figure 3.35** What output will be at Line X and Line Y?

3.15 Give an example of a situation in which ordinary pipes are more suitable than named pipes and an example of a situation in which named pipes are more suitable than ordinary pipes.

3.16 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.

3.17 Using the program shown in Figure 3.35, explain what the output will be at lines X and Y.

3.18 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

c. Send by copy and send by reference

d. Fixed-sized and variable-sized messages