of open files, signal-handling information, and virtual memory. When `fork()` is invoked, a new task is created, along with a *copy* of all the associated data structures of the parent process. A new task is also created when the `clone()` system call is made. However, rather than copying all data structures, the new task *points* to the data structures of the parent task, depending on the set of flags passed to `clone()`.

4.7 Summary

A thread is a flow of control within a process. A multithreaded process contains several different flows of control within the same address space. The benefits of multithreading include increased responsiveness to the user, resource sharing within the process, economy, and the ability to take advantage of multiprocessor architectures.

User-level threads are threads that are visible to the programmer and are unknown to the kernel. The operating-system kernel supports and manages kernel-level threads. In general, user-level threads are faster to create and manage than are kernel threads, as no intervention from the kernel is required. Three different types of models relate user and kernel threads: The many-to-one model maps many user threads to a single kernel thread. The one-to-one model maps each user thread to a corresponding kernel thread. The many-to-many model multiplexes many user threads to a smaller or equal number of kernel threads.

Most modern operating systems provide kernel support for threads. Among these are Windows 98, NT, 2000, and XP, as well as Solaris and Linux.

Thread libraries provide the application programmer with an API for creating and managing threads. Three primary thread libraries are in common use: POSIX Pthreads, Win32 threads for Windows systems, and Java threads.

Multithreaded programs introduce many challenges for the programmer, including the semantics of the `fork()` and `exec()` system calls. Other issues include thread cancellation, signal handling, and thread-specific data. The Java API addresses many of these threading issues.

ercises

4.1 Provide two programming examples in which multithreading does *not* provide better performance than a single-threaded solution.

4.2 Describe the actions taken by a thread library to context-switch between user-level threads.

4.3 Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

4.4 Which of the following components of program state are shared across threads in a multithreaded process?

a. Register values

b. Heap memory

c. Global variables

d. Stack memory

4.5 Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system?

4.6 The Java API provides several different thread-pool architectures:

a. `newFixedThreadPool(int)`

b. `newCachedThreadPool()`

c. `newSingleThreadExecutor()`

Discuss the merits of each.

4.7 As described in Section 4.6.2, Linux does not distinguish between processes and threads. Instead, Linux treats both in the same way allowing a task to be more akin to a process or a thread depending on the set of flags passed to the `clone()` system call. In contrast, many operating systems—such as Windows XP and Solaris—treat processes and threads differently. Typically, such systems use a notation where the data structure for a process contains pointers to the separate threads belonging to the process. Contrast these two approaches for modeling processes and threads within the kernel.

4.8 The program shown in Figure 4.20 uses the Pthreads API. What would be output from the program at `LINE C` and `LINE P`?

4.9 Consider a multiprocessor system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processors in the system. Discuss the performance implications of the following scenarios.

a. The number of kernel threads allocated to the program is less than the number of processors.

b. The number of kernel threads allocated to the program is equal to the number of processors.

c. The number of kernel threads allocated to the program is greater than the number of processors but less than the number of user-level threads.

4.10 Write a multithreaded Java, Pthreads, or Win32 program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.

```
#include <pthread.h>
#include <stdio.h>

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
   int pid;
   pthread_t tid;
   pthread_attr_t attr;

   pid = fork();

   if (pid == 0) { /* child process */
      pthread_attr_init(&attr);
      pthread_create(&tid,&attr,runner,NULL);
      pthread_join(tid,NULL);
      printf("CHILD: value = %d",value); /* LINE C */
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE P */
   }
}

void *runner(void *param) {
   value = 5;
   pthread_exit(0);
}
```

Figure 4.20 C program for Exercise 4.8.

4.11 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 multithreaded program that generates the Fibonacci series using either the Java, Pthreads, or Win32 thread library. This program should work as follows: The user will enter on the command line the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that are shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, this will require

having the parent thread wait for the child thread to finish, using th techniques described in Section 4.3.

4.12 Modify the socket-based date server (Figure 3.26) in Chapter 3 so tha the server services each client request in a separate thread.

4.13 Exercise 3.9 in Chapter 3 involves designing an echo server using th Java threading API. However, this server is single-threaded, meaning th server cannot respond to concurrent echo clients until the current clien exits. Modify the solution to Exercise 3.9 so that the echo server service each client in a separate thread.

4.14 In Exercise 4.13, you are asked to service each client in a separate thread Modify your solution so that each client is serviced using a thread poo as discussed in Section 4.5.4. Experiment with the three different model of thread-pool architecture presented in that section.

Project—Matrix Multiplication

This project consists of designing a multithreaded solution that perform matrix multiplication. You will write your solution using the three thread libraries discussed in this chapter: Java, Pthreads, and Win32.

Given two matrices A and B, where matrix A contains M rows and columns and matrix B contains K rows and N columns, the **matrix product** o A and B is matrix C, where C contains M rows and N columns. The entry i matrix C for row i column j ($C_{i,j}$) is the sum of the products of the element for row i in matrix A and column j in matrix B. That is,

$$C_{i,j} = \sum_{n=1}^{K} A_{i,n} \times B_{n,j}$$

For example, if A were a 3-by-2 matrix and B were a 2-by-3 matrix, elemen $C_{3,1}$ would be the sum of $A_{3,1} \times B_{1,1}$ and $A_{3,2} \times B_{2,1}$.

For this project, calculate each element $C_{i,j}$ in a separate *worker* thread. Th will involve creating $M \times N$ worker threads. The main—or parent—thread wi initialize matrices A and B and allocate sufficient memory for matrix C, whic will hold the product of matrices A and B. These matrices will be declared global data so that each worker thread has access to A, B, and C.

Matrices A and B can be initialized statically, as shown below:

```
#define M 3
#define K 2
#define N 3

int A [M][K] = { {1,4}, {2,5}, {3,6} };
int B [K][N] = { {8,7,6}, {5,4,3} };
int C [M] [N];
```

Alternatively, they can be populated by reading in values from a file.

Passing Parameters to Each Thread

The parent thread will create $M \times N$ worker threads, passing each worker the values of row i and column j that it is to use in calculating the matrix product. This requires passing two parameters to each thread. The easiest approach with Pthreads and Win32 is to create a data structure using a `struct`. The members of this structure are i and j, and the structure appears as follows:

```
/* structure for passing data to threads */
struct v
{
   int i; /* row */
   int j; /* column */
};
```

Both the Pthreads and Win32 programs will create the worker threads using a strategy similar to that shown below:

```
/* We have to create M * N worker threads */
for (i = 0; i < M, i++)
   for (j = 0; j < N; j++ ) {
      struct v *data = (struct v *) malloc(sizeof(struct v));
      data->i = i;
      data->j = j;
      /* Now create the thread passing it data as a parameter */
   }
}
```

The `data` pointer will be passed to either the `pthread_create()` (Pthreads) function or the `CreateThread()` (Win32) function, which in turn will pass it as a parameter to the function that is to run as a separate thread.

Sharing of data between Java threads is different from sharing between threads in Pthreads or Win32. One approach is for the main thread to create and initialize matrices A, B, and C. This main thread will then create the worker threads, passing the three matrices—along with row i and column j—to the constructor for each worker. Thus, the outline of a worker thread appears in Figure 4.21.

Waiting for Threads to Complete

Once all worker threads have completed, the main thread will output the product contained in matrix C. This requires the main thread to wait for all worker threads to finish before it can output the value of the matrix product. Several different strategies can be used to enable a thread to wait for other threads to finish. One technique, known as a **barrier**, will be covered in a programming exercise in Chapter 6. Section 4.3 describes how to wait for a child thread to complete using the Win32, Pthreads, and Java thread libraries. Win32 provides the `WaitForSingleObject()` function, whereas Pthreads and Java use `pthread_join()` and `join()`, respectively. However, in these

```
public class WorkerThread implements Runnable
{
    private int row;
    private int col;
    private int[][] A;
    private int[][] B;
    private int[][] C;

    public WorkerThread(int row, int col, int[][] A,
     int[][] B, int[][] C) {
      this.row = row;
      this.col = col;
      this.A = A;
      this.B = B;
      this.C = C;
    }
    public void run() {
      /* calculate the matrix product in C[row] [col] *
    }
}
```

Figure 4.21 Worker thread in Java.

programming examples, the parent thread waits for a single child thread finish; completing this exercise will require waiting for multiple threads.

In Section 4.3.2, we describe the `WaitForSingleObject()` function, which is used to wait for a single thread to finish. The Win32 API also provides the `WaitForMultipleObjects()` function, which is used when waiting for multiple threads to complete. `WaitForMultipleObjects()` is passed four parameters:

1. The number of objects to wait for
2. A pointer to the array of objects
3. A flag indicating if all objects have been signaled
4. A timeout duration (or `INFINITE`)

For example, if `THandles` is an array of thread `HANDLE` objects of size then the parent thread can wait for all its child threads to complete with

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Figure 4.22 Pthread code for joining multiple threads.

```
final static int NUM_THREADS = 10;

/* an array of threads to be joined upon */
Thread[] workers = new Thread[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++) {
   try {
      workers[i].join();
   } catch (InterruptedException ie) { }
}
```

Figure 4.23 Java code for joining multiple threads.

statement:

```
WaitForMultipleObjects(N, THandles, TRUE, INFINITE);
```

A simple strategy for waiting on several threads using the Pthreads `pthread_join()` or Java's `join()` is to enclose the join operation within a simple `for` loop. For example, you could join on ten threads using the Pthread code depicted in Figure 4.22. The equivalent code using Java threads is shown in Figure 4.23.

Bibliographical Notes

Thread performance issues were discussed by Anderson et al. [1989], who continued their work in Anderson et al. [1991] by evaluating the performance of user-level threads with kernel support. Bershad et al. [1990] describe combining threads with RPC. Engelschall [2000] discusses a technique for supporting user-level threads. An analysis of an optimal thread-pool size can be found in Ling et al. [2000]. Scheduler activations were first presented in Anderson et al. [1991], and Williams [2002] discusses scheduler activations in the NetBSD system. Other mechanisms by which the user-level thread library and the kernel cooperate with each other are discussed in Marsh et al. [1991], Govindan and Anderson [1991], Draves et al. [1991], and Black [1990]. Zabatta and Young [1998] compare Windows NT and Solaris threads on a symmetric multiprocessor. Pinilla and Gill [2003] compare Java thread performance on Linux, Windows, and Solaris.

Vahalia [1996] covers threading in several versions of UNIX. Mauro and McDougall [2001] describe recent developments in threading the Solaris kernel. Solomon and Russinovich [2000] discuss threading in Windows 2000. Bovet and Cesati [2002] and Love [2004] explain how Linux handles threading.

Information on Pthreads programming is given in Lewis and Berg [1998] and Butenhof [1997]. Beveridge and Wiener [1997] and Cohen and Woodring [1997] describe multithreading using Win32. Lewis and Berg [2000], Holub [2000], and Oaks and Wong discuss multithreading in Java. Goetz et al. [2006] present a detailed discussion of concurrent programming in Java, as well as the `java.util.concurrent` package.