threads have reached this point as well. When the last thread reaches the barrier point, all threads are released and can resume concurrent execution.

Assume that the barrier is initialized to N—the number of threads that must wait at the barrier point:

```
init(N);
```

Each thread then performs some work until it reaches the barrier point:

```
/* do some work for awhile */

barrier_point();

/* do some work for awhile */
```

Using either the POSIX or Java synchronization tools described in this chapter, construct a barrier that implements the following API:

- `int init(int n)`—Initializes the barrier to the specified size.
- `int barrier_point(void)`—Identifies the barrier point. All threads are released from the barrier when the last thread reaches this point.

The return value of each function is used to identify error conditions. Each function will return 0 under normal operation and will return −1 if an error occurs. A testing harness is provided in the source-code download to test your implementation of the barrier.

Programming Projects

Project 1—Designing a Thread Pool

Thread pools were introduced in Section 4.5.1. When thread pools are used, a task is submitted to the pool and executed by a thread from the pool. Work is submitted to the pool using a queue, and an available thread removes work from the queue. If there are no available threads, the work remains queued until one becomes available. If there is no work, threads await notification until a task becomes available.

This project involves creating and managing a thread pool, and it may be completed using either Pthreds and POSIX synchronization or Java. Below we provide the details relevant to each specific technology.

I. POSIX

The POSIX version of this project will involve creating a number of threads using the Pthreads API as well as using POSIX mutex locks and semaphores for synchronization.

The Client

Users of the thread pool will utilize the following API:

- `void pool_init()`—Initializes the thread pool.
- `int pool_submit(void (*somefunction)(void *p), void *p)`—where `somefunction` is a pointer to the function that will be executed by a thread from the pool and `p` is a parameter passed to the function.
- `void pool_shutdown(void)`—Shuts down the thread pool once all tasks have completed.

We provide an example program `client.c` in the source code download that illustrates how to use the thread pool using these functions.

Implementation of the Thread Pool

In the source code download we provide the C source file `threadpool.c` as a partial implementation of the thread pool. You will need to implement the functions that are called by client users, as well as several additional functions that support the internals of the thread pool. Implementation will involve the following activities:

1. The `pool_init()` function will create the threads at startup as well as initialize mutual-exclusion locks and semaphores.
2. The `pool_submit()` function is partially implemented and currently places the function to be executed—as well as its data— into a `task` struct. The `task` struct represents work that will be completed by a thread in the pool. `pool_submit()` will add these tasks to the queue by invoking the `enqueue()` function, and worker threads will call `dequeue()` to retrieve work from the queue. The queue may be implemented statically (using arrays) or dynamically (using a linked list).

   The `pool_init()` function has an `int` return value that is used to indicate if the task was successfully submitted to the pool (0 indicates success, 1 indicates failure). If the queue is implemented using arrays, `pool_init()` will return 1 if there is an attempt to submit work and the queue is full. If the queue is implemented as a linked list, `pool_init()` should always return 0 unless a memory allocation error occurs.
3. The `worker()` function is executed by each thread in the pool, where each thread will wait for available work. Once work becomes available, the thread will remove it from the queue and invoke `execute()` to run the specified function.

   A semaphore can be used for notifying a waiting thread when work is submitted to the thread pool. Either named or unnamed semaphores may be used. Refer to Section 7.3.2 for further details on using POSIX semaphores.

4. A mutex lock is necessary to avoid race conditions when accessing or modifying the queue. (Section 7.3.1 provides details on Pthreads mutex locks.)
5. The `pool_shutdown()` function will cancel each worker thread and then wait for each thread to terminate by calling `pthread_join()`. Refer to Section 4.6.3 for details on POSIX thread cancellation. (The semaphore operation `sem_wait()` is a cancellation point that allows a thread waiting on a semaphore to be cancelled.)

Refer to the source-code download for additional details on this project. In particular, the `README` file describes the source and header files, as well as the `Makefile` for building the project.

II. Java

The Java version of this project may be completed using Java synchronization tools as described in Section 7.4. Synchronization may depend on either (a) monitors using `synchronized/wait()/notify()` (Section 7.4.1) or (b) semaphores and reentrant locks (Section 7.4.2 and Section 7.4.3). Java threads are described in Section 4.4.3.

Implementation of the Thread Pool

Your thread pool will implement the following API:

- `ThreadPool()`—Create a default-sized thread pool.
- `ThreadPool(int size)`—Create a thread pool of size `size`.
- `void add(Runnable task)`—Add a task to be performed by a thread in the pool.
- `void shutdown()`—Stop all threads in the pool.

We provide the Java source file `ThreadPool.java` as a partial implementation of the thread pool in the source code download. You will need to implement the methods that are called by client users, as well as several additional methods that support the internals of the thread pool. Implementation will involve the following activities:

1. The constructor will first create a number of idle threads that await work.
2. Work will be submitted to the pool via the `add()` method, which adds a task implementing the `Runnable` interface. The `add()` method will place the `Runnable` task into a queue (you may use an available structure from the Java API such as `java.util.List`).
3. Once a thread in the pool becomes available for work, it will check the queue for any `Runnable` tasks. If there is such a task, the idle thread will remove the task from the queue and invoke its `run()` method. If the queue is empty, the idle thread will wait to be notified when work

becomes available. (The `add()` method may implement notification using either `notify()` or semaphore operations when it places a `Runnable` task into the queue to possibly awaken an idle thread awaiting work.)

4. The `shutdown()` method will stop all threads in the pool by invoking their `interrupt()` method. This, of course, requires that `Runnable` tasks being executed by the thread pool check their interruption status (Section 4.6.3).

Refer to the source-code download for additional details on this project. In particular, the README file describes the Java source files, as well as further details on Java thread interruption.

Project 2—The Sleeping Teaching Assistant

A university computer science department has a teaching assistant (TA) who helps undergraduate students with their programming assignments during regular office hours. The TA's office is rather small and has room for only one desk with a chair and computer. There are three chairs in the hallway outside the office where students can sit and wait if the TA is currently helping another student. When there are no students who need help during office hours, the TA sits at the desk and takes a nap. If a student arrives during office hours and finds the TA sleeping, the student must awaken the TA to ask for help. If a student arrives and finds the TA currently helping another student, the student sits on one of the chairs in the hallway and waits. If no chairs are available, the student will come back at a later time.

Using POSIX threads, mutex locks, and semaphores, implement a solution that coordinates the activities of the TA and the students. Details for this assignment are provided below.

The Students and the TA

Using Pthreads (Section 4.4.1), begin by creating n students where each student will run as a separate thread. The TA will run as a separate thread as well. Student threads will alternate between programming for a period of time and seeking help from the TA. If the TA is available, they will obtain help. Otherwise, they will either sit in a chair in the hallway or, if no chairs are available, will resume programming and will seek help at a later time. If a student arrives and notices that the TA is sleeping, the student must notify the TA using a semaphore. When the TA finishes helping a student, the TA must check to see if there are students waiting for help in the hallway. If so, the TA must help each of these students in turn. If no students are present, the TA may return to napping.

Perhaps the best option for simulating students programming—as well as the TA providing help to a student—is to have the appropriate threads sleep for a random period of time.

Coverage of POSIX mutex locks and semaphores is provided in Section 7.3. Consult that section for details.

Project 3—The Dining-Philosophers Problem

In Section 7.1.3, we provide an outline of a solution to the dining-philosophers problem using monitors. This project involves implementing a solution to this problem using either POSIX mutex locks and condition variables or Java condition variables. Solutions will be based on the algorithm illustrated in Figure 7.7.

Both implementations will require creating five philosophers, each identified by a number 0 . . 4. Each philosopher will run as a separate thread. Philosophers alternate between thinking and eating. To simulate both activities, have each thread sleep for a random period between one and three seconds.

I. POSIX

Thread creation using Pthreads is covered in Section 4.4.1. When a philosopher wishes to eat, she invokes the function

```
pickup_forks(int philosopher_number)
```

where `philosopher_number` identifies the number of the philosopher wishing to eat. When a philosopher finishes eating, she invokes

```
return_forks(int philosopher_number)
```

Your implementation will require the use of POSIX condition variables, which are covered in Section 7.3.

II. Java

When a philosopher wishes to eat, she invokes the method `takeForks(philosopherNumber)`, where `philosopherNumber` identifies the number of the philosopher wishing to eat. When a philosopher finishes eating, she invokes `returnForks(philosopherNumber)`.

Your solution will implement the following interface:

```
public interface DiningServer
{
  /* Called by a philosopher when it wishes to eat */
  public void takeForks(int philosopherNumber);

  /* Called by a philosopher when it is finished eating */
  public void returnForks(int philosopherNumber);
}
```

It will require the use of Java condition variables, which are covered in Section 7.4.4.

Project 4—The Producer–Consumer Problem

In Section 7.1.1, we presented a semaphore-based solution to the producer–consumer problem using a bounded buffer. In this project, you will design a programming solution to the bounded-buffer problem using the producer and consumer processes shown in Figures 5.9 and 5.10. The solution presented in Section 7.1.1 uses three semaphores: `empty` and `full`, which count the number of empty and full slots in the buffer, and `mutex`, which is a binary (or mutual-exclusion) semaphore that protects the actual insertion or removal of items in the buffer. For this project, you will use standard counting semaphores for `empty` and `full` and a mutex lock, rather than a binary semaphore, to represent `mutex`. The producer and consumer—running as separate threads—will move items to and from a buffer that is synchronized with the `empty`, `full`, and `mutex` structures. You can solve this problem using either Pthreads or the Windows API.

The Buffer

Internally, the buffer will consist of a fixed-size array of type `buffer_item` (which will be defined using a `typedef`). The array of `buffer_item` objects will be manipulated as a circular queue. The definition of `buffer_item`, along with the size of the buffer, can be stored in a header file such as the following:

```
/* buffer.h */
typedef int buffer_item;
#define BUFFER_SIZE 5
```

The buffer will be manipulated with two functions, `insert_item()` and `remove_item()`, which are called by the producer and consumer threads, respectively. A skeleton outlining these functions appears in Figure 7.14.

The `insert_item()` and `remove_item()` functions will synchronize the producer and consumer using the algorithms outlined in Figure 7.1 and Figure 7.2. The buffer will also require an initialization function that initializes the mutual-exclusion object `mutex` along with the `empty` and `full` semaphores.

The `main()` function will initialize the buffer and create the separate producer and consumer threads. Once it has created the producer and consumer threads, the `main()` function will sleep for a period of time and, upon awakening, will terminate the application. The `main()` function will be passed three parameters on the command line:

1. How long to sleep before terminating
2. The number of producer threads
3. The number of consumer threads

A skeleton for this function appears in Figure 7.15.

```
#include "buffer.h"

/* the buffer */
buffer_item buffer[BUFFER_SIZE];

int insert_item(buffer_item item) {
  /* insert item into buffer
   return 0 if successful, otherwise
   return -1 indicating an error condition */
}

int remove_item(buffer_item *item) {
  /* remove an object from buffer
   placing it in item
   return 0 if successful, otherwise
   return -1 indicating an error condition */
}
```

Figure 7.14 Outline of buffer operations.

The Producer and Consumer Threads

The producer thread will alternate between sleeping for a random period of time and inserting a random integer into the buffer. Random numbers will be produced using the `rand()` function, which produces random integers between 0 and RAND_MAX. The consumer will also sleep for a random period of time and, upon awakening, will attempt to remove an item from the buffer. An outline of the producer and consumer threads appears in Figure 7.16.

```
#include "buffer.h"

int main(int argc, char *argv[]) {
 /* 1. Get command line arguments argv[1],argv[2],argv[3] */
 /* 2. Initialize buffer */
 /* 3. Create producer thread(s) */
 /* 4. Create consumer thread(s) */
 /* 5. Sleep */
 /* 6. Exit */
}
```

Figure 7.15 Outline of skeleton program.

```
#include <stdlib.h> /* required for rand() */
#include "buffer.h"

void *producer(void *param) {
  buffer_item item;

  while (true) {
    /* sleep for a random period of time */
    sleep(...);
    /* generate a random number */
    item = rand();
    if (insert_item(item))
      fprintf("report error condition");
    else
      printf("producer produced %d\n",item);
}

void *consumer(void *param) {
  buffer_item item;

  while (true) {
    /* sleep for a random period of time */
    sleep(...);
    if (remove_item(&item))
      fprintf("report error condition");
    else
      printf("consumer consumed %d\n",item);
}
```

Figure 7.16 An outline of the producer and consumer threads.

As noted earlier, you can solve this problem using either Pthreads or the Windows API. In the following sections, we supply more information on each of these choices.

Pthreads Thread Creation and Synchronization

Creating threads using the Pthreads API is discussed in Section 4.4.1. Coverage of mutex locks and semaphores using Pthreads is provided in Section 7.3. Refer to those sections for specific instructions on Pthreads thread creation and synchronization.

Windows Threads

Section 4.4.2 discusses thread creation using the Windows API. Refer to that section for specific instructions on creating threads.

Windows Mutex Locks

Mutex locks are a type of dispatcher object, as described in Section 7.2.1. The following illustrates how to create a mutex lock using the `CreateMutex()` function:

```
#include <windows.h>

HANDLE Mutex;
Mutex = CreateMutex(NULL, FALSE, NULL);
```

The first parameter refers to a security attribute for the mutex lock. By setting this attribute to `NULL`, we prevent any children of the process from creating this mutex lock to inherit the handle of the lock. The second parameter indicates whether the creator of the mutex lock is the lock's initial owner. Passing a value of `FALSE` indicates that the thread creating the mutex is not the initial owner. (We shall soon see how mutex locks are acquired.) The third parameter allows us to name the mutex. However, because we provide a value of `NULL`, we do not name the mutex. If successful, `CreateMutex()` returns a `HANDLE` to the mutex lock; otherwise, it returns `NULL`.

In Section 7.2.1, we identified dispatcher objects as being either ***signaled*** or ***nonsignaled***. A signaled dispatcher object (such as a mutex lock) is available for ownership. Once it is acquired, it moves to the nonsignaled state. When it is released, it returns to signaled.

Mutex locks are acquired by invoking the `WaitForSingleObject()` function. The function is passed the `HANDLE` to the lock along with a flag indicating how long to wait. The following code demonstrates how the mutex lock created above can be acquired:

```
WaitForSingleObject(Mutex, INFINITE);
```

The parameter value `INFINITE` indicates that we will wait an infinite amount of time for the lock to become available. Other values could be used that would allow the calling thread to time out if the lock did not become available within a specified time. If the lock is in a signaled state, `WaitForSingleObject()` returns immediately, and the lock becomes nonsignaled. A lock is released (moves to the signaled state) by invoking `ReleaseMutex()`—for example, as follows:

```
ReleaseMutex(Mutex);
```

Windows Semaphores

Semaphores in the Windows API are dispatcher objects and thus use the same signaling mechanism as mutex locks. Semaphores are created as follows:

```
#include <windows.h>

HANDLE Sem;
Sem = CreateSemaphore(NULL, 1, 5, NULL);
```

The first and last parameters identify a security attribute and a name for the semaphore, similar to what we described for mutex locks. The second and third parameters indicate the initial value and maximum value of the semaphore. In this instance, the initial value of the semaphore is 1, and its maximum value is 5. If successful, `CreateSemaphore()` returns a `HANDLE` to the mutex lock; otherwise, it returns `NULL`.

Semaphores are acquired with the same `WaitForSingleObject()` function as mutex locks. We acquire the semaphore `Sem` created in this example by using the following statement:

```
WaitForSingleObject(Sem, INFINITE);
```

If the value of the semaphore is > 0, the semaphore is in the signaled state and thus is acquired by the calling thread. Otherwise, the calling thread blocks indefinitely—as we are specifying `INFINITE`—until the semaphore returns to the signaled state.

The equivalent of the `signal()` operation for Windows semaphores is the `ReleaseSemaphore()` function. This function is passed three parameters:

1. The HANDLE of the semaphore
2. How much to increase the value of the semaphore
3. A pointer to the previous value of the semaphore

We can use the following statement to increase `Sem` by 1:

```
ReleaseSemaphore(Sem, 1, NULL);
```

Both `ReleaseSemaphore()` and `ReleaseMutex()` return a nonzero value if successful and 0 otherwise.