The pthread_cond_wait() function is used for waiting on a condition variable. The following code illustrates how a thread can wait for the condition a == b to become true using a Pthread condition variable:

```
pthread_mutex_lock(&mutex);
while (a != b)
    pthread_cond_wait(&mutex, &cond_var);

pthread_mutex_unlock(&mutex);
```

The mutex lock associated with the condition variable must be locked before the pthread_cond_wait() function is called, since it is used to protect the data in the conditional clause from a possible race condition. Once this lock is acquired, the thread can check the condition. If the condition is not true, the thread then invokes pthread_cond_wait(), passing the mutex lock and the condition variable as parameters. Calling pthread_cond_wait() releases the mutex lock, thereby allowing another thread to access the shared data and possibly update its value so that the condition clause evaluates to true. (To protect against program errors, it is important to place the conditional clause within a loop so that the condition is rechecked after being signaled.)

A thread that modifies the shared data can invoke the pthread_cond_signal() function, thereby signaling one thread waiting on the condition variable. This is illustrated below:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

It is important to note that the call to pthread_cond_signal() does not release the mutex lock. It is the subsequent call to pthread_mutex_unlock() that releases the mutex. Once the mutex lock is released, the signaled thread becomes the owner of the mutex lock and returns control from the call to pthread_cond_wait().

Project 3—Producer–Consumer Problem

In Section 5.7.1, 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 5.7.1 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.

```
#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 5.24 Outline of buffer operations.

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 5.24.

The `insert_item()` and `remove_item()` functions will synchronize the producer and consumer using the algorithms outlined in Figures 5.9 and 5.10. 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

```
#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 5.25 Outline of skeleton program.

A skeleton for this function appears in Figure 5.25.

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 5.26.

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 5.9.4. Refer to those sections for specific instructions on Pthreads thread creation and synchronization.

Windows

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 5.9.1. The following illustrates how to create a mutex lock using the `CreateMutex()` function:

```
#include <windows.h>

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

```
#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 5.26 An outline of the producer and consumer threads.

The first parameter refers to a security attribute for the mutex lock. By setting this attribute to NULL, we disallow any children of the process 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 5.9.1, 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);
```