**6.28** The `decrease_count()` function in the previous exercise currently returns 0 if sufficient resources are available and -1 otherwise. This leads to awkward programming for a process that wishes obtain a number of resources:

```
while (decrease_count(count) == -1)
   ;
```

Rewrite the resource-manager code segment using a monitor and condition variables so that the `decrease_count()` function suspends the process until sufficient resources are available. This will allow a process to invoke `decrease_count()` by simply calling

```
decrease_count(count);
```

The process will only return from this function call when sufficient resources are available.

## Project: Producer–Consumer Problem

In Section 6.6.1, we present a semaphore-based solution to the producer–consumer problem using a bounded buffer. In this project, we will design a programming solution to the bounded-buffer problem using the producer and consumer processes shown in Figures 6.10 and 6.11. The solution presented in Section 6.6.1 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, standard counting semaphores will be used for `empty` and `full`, and, rather than a binary semaphore, a mutex lock will be used to represent `mutex`. The producer and consumer—running as separate threads—will move items to and from a buffer that is synchronized with these `empty`, `full`, and `mutex` structures. You can solve this problem using either Pthreads or the Win32 API.

### The Buffer

Internally, the buffer will consist of a fixed-size array of type `buffer_item` (which will be defined using a `typefdef`). 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 as:

```
#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 */
}
```

The `insert_item()` and `remove_item()` functions will synchronize the producer and consumer using the algorithms outlined in Figures 6.10 and 6.11. 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 as:

```
#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 */
}
```

### 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 as:

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

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

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

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

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

In the following sections, we first cover details specific to Pthreads and then describe details of the Win32 API.

### Pthreads Thread Creation

Creating threads using the Pthreads API is discussed in Chapter 4. Please refer to that chapter for specific instructions regarding creation of the producer and consumer using Pthreads.

### Pthreads Mutex Locks

The following code sample illustrates how mutex locks available in the Pthread API can be used to protect a critical section:

```
#include <pthread.h>
pthread_mutex_t mutex;

/* create the mutex lock */
pthread_mutex_init(&mutex,NULL);

/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/*** critical section ***/

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

Pthreads uses the `pthread_mutex_t` data type for mutex locks. A mutex is created with the `pthread_mutex_init(&mutex,NULL)` function, with the first parameter being a pointer to the mutex. By passing `NULL` as a second parameter, we initialize the mutex to its default attributes. The mutex is acquired and released with the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. If the mutex lock is unavailable when `pthread_mutex_lock()` is invoked, the calling thread is blocked until the owner invokes `pthread_mutex_unlock()`. All mutex functions return a value of 0 with correct operation; if an error occurs, these functions return a nonzero error code.

### Pthreads Semaphores

Pthreads provides two types of semaphores—named and unnamed. For this project, we use unnamed semaphores. The code below illustrates how a semaphore is created:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 5 */
sem_init(&sem, 0, 5);
```

The `sem_init()` creates and initializes a semaphore. This function is passed three parameters:

1. A pointer to the semaphore
2. A flag indicating the level of sharing
3. The semaphore's initial value

In this example, by passing the flag 0, we are indicating that this semaphore can only be shared by threads belonging to the same process that created the semaphore. A nonzero value would allow other processes to access the semaphore as well. In this example, we initialize the semaphore to the value 5.

In Section 6.5, we described the classical `wait()` and `signal()` semaphore operations. Pthreads names the `wait()` and `signal()` operations `sem_wait()` and `sem_post()`, respectively. The code example below creates a binary semaphore `mutex` with an initial value of 1 and illustrates its use in protecting a critical section:

```
#include <semaphore.h>
sem_t sem mutex;

/* create the semaphore */
sem_init(&mutex, 0, 1);

/* acquire the semaphore */
sem_wait(&mutex);

/*** critical section ***/

/* release the semaphore */
sem_post(&mutex);
```

### Win32

Details concerning thread creation using the Win32 API are available in Chapter 4. Please refer to that chapter for specific instructions.

### Win32 Mutex Locks

Mutex locks are a type of dispatcher object, as described in Section 6.8.2. 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 are disallowing any children of the process creating this mutex lock to inherit the handle of the mutex. The second parameter indicates whether the creator of the mutex is the initial owner of the mutex lock. 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 naming of 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 6.8.2, we identified dispatcher objects as being either *signaled* or *nonsignaled*. A signaled object is available for ownership; once a dispatcher object (such as a mutex lock) is acquired, it moves to the nonsignaled state. When the object is released, it returns to signaled.

Mutex locks are acquired by invoking the `WaitForSingleObject()` function, passing the function the `HANDLE` to the lock and 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 nonsignaled state) by invoking `ReleaseMutex()`, such as:

```
ReleaseMutex(Mutex);
```

### Win32 Semaphores

Semaphores in the Win32 API are also 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 was 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 statement:

```
WaitForSingleObject(Semaphore, 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 becomes signaled.

The equivalent of the `signal()` operation on Win32 semaphores is the `ReleaseSemaphore()` function. This function is passed three parameters: (1) the `HANDLE` of the semaphore, (2) the amount by which to increase the value of the semaphore, and (3) a pointer to the previous value of the semaphore. We can increase `Sem` by 1 using the following statement:

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

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