CHAPTER 3

Thread Synchronization

In the last chapter, we described several synchronization objects for multithreaded programming. For correct concurrent programs, multiple threads of execution must be synchronized to protect the integrity of shared data. In this chapter, we illustrate basic synchronization techniques using some classic concurrency problems of *producer-consumer, bounded-buffer,* and *readers-writers.*

3.1 The Producer-Consumer Problem

In the last chapter, we saw the simplest form of mutual exclusion: before accessing shared data, each thread acquires ownership of a synchronization object. Once the thread has finished accessing the data, it relinquishes the ownership so that other threads can acquire the synchronization object and access the same data. Therefore, when accessing shared data, each thread excludes all others from accessing the same data.

This simple form of mutual exclusion, however, is not enough for certain classes of applications where designated threads are *producers* and *consumers* of data. The producer threads write new values, while consumer threads read them. An analogy of the producer-

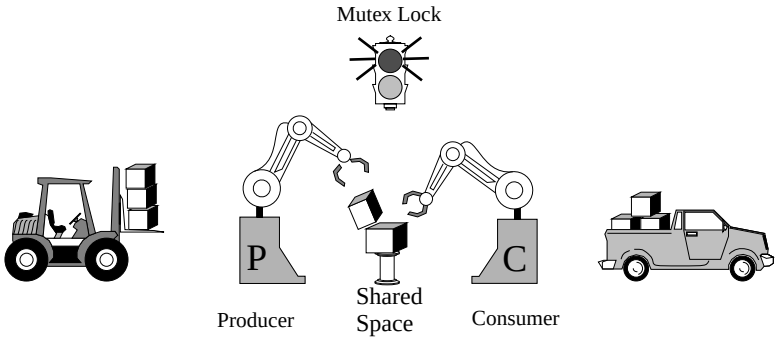

Figure 3-1. An Illustrative Analogy for the Producer-Consumer Problem.

consumer situation is illustrated in Figure 3-1, where each thread, *producer* and *consumer*, is represented by a robot arm. The producer picks up a box and puts it on a pedestal (shared buffer) from which the consumer can pick it up. The consumer robot picks up boxes from the pedestal for delivery. If we just use the simple synchronization technique of acquiring and relinquishing a synchronization object, we may get incorrect behavior. For example, the producer may try to put a block on the pedestal when there is already a block there. In such a case, the newly produced block will fall off the pedestal.

To illustrate this situation in a multithreaded program, we present an implementation of a producer-consumer situation with only mutual exclusion among threads.

```
1   #include <iostream.h>
2   #include <windows.h>
3
4   int SharedBuffer;
5   HANDLE hMutex;
6
7   void Producer()
8   {
9     int i;
10
11    for (i=20; i>=0; i--) {
12      if (WaitForSingleObject(hMutex,INFINITE) == WAIT_FAILED){
13        cerr << "ERROR: Producer()" << endl;
14        ExitThread(0);
15      }
16      // got Mutex, begin critical section
17      cout << "Produce: " << i << endl;
18      SharedBuffer = i;
19      ReleaseMutex(hMutex);  // end critical section
20    }
21  }
22
23
24  void Consumer()
25  {
26    int result;
```

```
27
28    while (1) {
29      if (WaitForSingleObject(hMutex,INFINITE) == WAIT_FAILED){
30        cerr << "ERROR: Producer" << endl;
31        ExitThread(0);
32      }
33      if (SharedBuffer == 0) {
34        cout << "Consumed " << SharedBuffer << ": end of data" << endl;
35        ReleaseMutex(hMutex); // end critical section
36        ExitThread(0);
37      }
38
39      // got Mutex, data in buffer, start consuming
40      if (SharedBuffer > 0){  // ignore negative values
41        result = SharedBuffer;
42        cout << "Consumed: " << result << endl;
43        ReleaseMutex(hMutex); // end critical section
44      }
45    }
46  }
47
48  void main()
49  {
50    HANDLE hThreadVector[2];
51    DWORD ThreadID;
52
53    SharedBuffer = -1;
54    hMutex = CreateMutex(NULL,FALSE,NULL);
55
56     hThreadVector[0]= CreateThread(NULL,0,
57        (LPTHREAD_START_ROUTINE)Producer,
58        NULL, 0, (LPDWORD)&ThreadID);
59     hThreadVector[1]=CreateThread(NULL,0,
60        (LPTHREAD_START_ROUTINE)Consumer,
61        NULL, 0, (LPDWORD)&ThreadID);
62    WaitForMultipleObjects(2,hThreadVector,TRUE,INFINITE);
63    // process ends here
64  }
65
```

This program creates two threads, *Producer* and *Consumer*, which exchange data using a shared variable, named `SharedBuffer`. All access to `SharedBuffer` must be from within a critical section. The program serializes access to the `SharedBuffer`, guaranteeing that concurrent accesses by producer and consumer threads will not corrupt the data in it. A sample output from a random run of the program is:

1. Produce: 20
2. Consumed: 20
3. Produce: 19
4. Consumed: 19
5. Produce: 18
6. Produce: 17
7. Produce: 16
16. Produce: 9
17. Produce: 8
18. Produce: 7
19. Produce: 6
20. Produce: 5
21. Produce: 4
22. Produce: 3

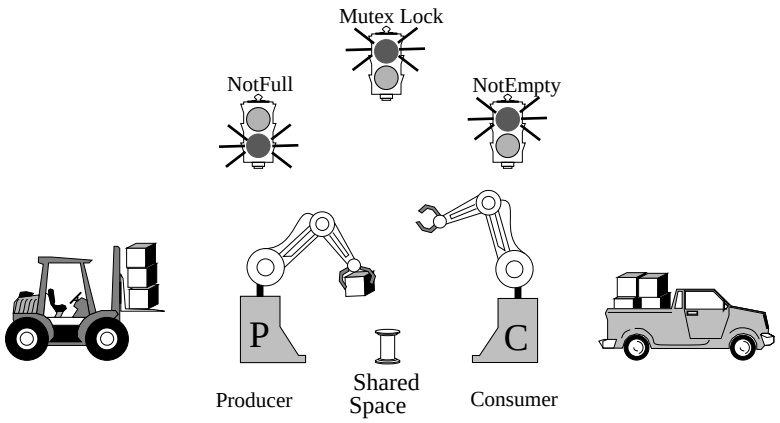

Figure 3-2. The Producer and Consumer Robots with Synchronization.

```
8. Produce: 15
9. Produce: 14
10. Produce: 13
11. Produce: 12
12. Produce: 11
13. Consumed: 11
14. Consumed: 11
15. Produce: 10
23. Produce: 2
24. Consumed: 2
25. Consumed: 2
26. Consumed: 2
27. Consumed: 2
28. Produce: 1
29. Consumed: 1
30. Consumed 0: end of data
```

As we can see, something is wrong: not every value produced by the producer is consumed, and sometimes the same value is consumed many times. The intended behavior is that the producer and consumer threads alternate in their access to the shared variable. We do not want the producer to overwrite the variable before its value is consumed; nor do we want the consumer thread to use the same value more than once.

The behavior occurs because mutual exclusion alone is not sufficient to solve the producer-consumer problem—we need both mutual exclusion and synchronization among the producer and consumer threads.

In order to achieve synchronization, we need a way for each thread to communicate with the others. When a producer produces a new value for the shared variable, it must inform the consumer threads of this event. Similarly, when a consumer has read a data value, it must trigger an event to notify possible producer threads about the empty buffer. Threads receiving such event signals can then gain access to the shared variable in order to produce or consume more data. Figure 3-2 shows our producer and consumer robots with added event signals.

The next program uses two event objects, named `hNotEmptyEvent` and `hNotFullEvent`, to synchronize the producer and the consumer threads.

```
1   #include <iostream.h>
2   #include <windows.h>
3
4   #define FULL 1
5   #define EMPTY 0
```

```
6
7   int SharedBuffer;
8   int BufferState;
9   HANDLE hMutex;
10  HANDLE hNotFullEvent, hNotEmptyEvent;
11
12  void Producer()
13  {
14    int i;
15
16    for (i=20; i>=0; i--) {
17      while(1) {
18        if (WaitForSingleObject(hMutex,INFINITE) == WAIT_FAILED){
19          cerr << "ERROR: Producer()" << endl;
20          ExitThread(0);
21        }
22        if (BufferState == FULL) {
23          ReleaseMutex(hMutex);
24          // wait until buffer is not full
25          WaitForSingleObject(hNotFullEvent,INFINITE);
26          continue; // back to loop to test BufferState again
27        }
28        // got mutex and buffer is not FULL, break out of while loop
29        break;
30      }
31
32      // got Mutex, buffer is not full, producing data
33      cout << "Produce: " << i << endl;
34      SharedBuffer = i;
35      BufferState = FULL;
36      ReleaseMutex(hMutex);  // end critical section
37      PulseEvent(hNotEmptyEvent); // wake up consumer thread
38    }
39  }
40
41  void Consumer()
42  {
43    int result;
44
45    while (1) {
46      if (WaitForSingleObject(hMutex,INFINITE) == WAIT_FAILED){
47        cerr << "ERROR: Producer()" << endl;
48        ExitThread(0);
49      }
50      if (BufferState == EMPTY) { // nothing to consume
51        ReleaseMutex(hMutex); // release lock to wait
52        // wait until buffer is not empty
53        WaitForSingleObject(hNotEmptyEvent,INFINITE);
54        continue; // return to while loop to contend for Mutex again
55      }
56
57      if (SharedBuffer == 0) { // test for end of data token
58        cout << "Consumed " << SharedBuffer << ": end of data" << endl;
59        ReleaseMutex(hMutex); // end critical section
```

```
60        ExitThread(0);
61      }
62       else {  // got Mutex, data in buffer, start consuming
63         result = SharedBuffer;
64         cout << "Consumed: " << result << endl;
65         BufferState = EMPTY;
66         ReleaseMutex(hMutex); // end critical section
67         PulseEvent(hNotFullEvent); // wake up producer thread
68      }
69    }
70  }
71
72  void main()
73  {
74    HANDLE hThreadVector[2];
75    DWORD ThreadID;
76
77    BufferState = EMPTY;
78    hMutex = CreateMutex(NULL,FALSE,NULL);
79
80     // create manual event objects
81    hNotFullEvent  = CreateEvent(NULL,TRUE,FALSE,NULL);
82    hNotEmptyEvent = CreateEvent(NULL,TRUE,FALSE,NULL);
83
84     hThreadVector[0]=CreateThread(NULL,0,
85       (LPTHREAD_START_ROUTINE)Producer,
86       NULL, 0, (LPDWORD)&ThreadID);
87     hThreadVector[1]=CreateThread(NULL,0,
88       (LPTHREAD_START_ROUTINE)Consumer,
89       NULL, 0, (LPDWORD)&ThreadID);
90    WaitForMultipleObjects(2,hThreadVector,TRUE,INFINITE);
91    // process ends here
92  }
```

We add a state variable `BufferState` to indicate the state of the buffer (`FULL` or `EMPTY`) and initialize it to `EMPTY`. Like the `SharedBuffer` variable itself, the `BufferState` variable must be protected in a critical section to serialize access. Before producing a new value for the shared buffer, a producer thread must test `BufferState` (line 22). If the state of the buffer is `FULL`, the producer thread cannot produce data; it must release the mutex lock, and wait for the buffer to be empty on the event object `hNotFullEvent`. Similarly, before attempting to read data from the shared buffer, a consumer thread must check its state. If the state is `EMPTY`, there is no data to read. In such a situation, a consumer thread must release the mutex lock, and wait for a nonempty buffer using the event object `hNotEmptyEvent`. When a thread produces a new value, it triggers the event `hNotEmptyEvent` to wake up any consumer threads waiting for data. Similarly, when a thread consumes a value, it triggers the event `hNotFullEvent` to wake up any producer threads waiting for the buffer to empty.

Note that when a producer or consumer thread wakes up after waiting for an event object, it must retest the value of `BufferState` (lines 22 and 50) in order to handle the situation when there is more than one producer or consumer thread, since another thread may change the value of `BufferState` by the time a producer or consumer thread successfully

regains access to the critical section. For example, suppose three consumer threads are awakened by an `hNotEmptyEvent`, and each of them immediately tries to get a mutex lock. The thread that gets the mutex lock will find data in the `SharedBuffer`, consume it, and set `BufferState` to `EMPTY`. Assuming that no new data is produced when the other two consumer threads get their respective mutex locks some time later, they will find that `BufferState` `EMPTY`; so they must wait on the `hNotEmptyEvent` again.

It is important that the producer and consumer threads each wait for the event objects outside their critical section; otherwise, the program can deadlock where each thread is waiting for the other to make progress. For example, suppose we do not release the mutex before waiting for the event object:

```
1   void Producer()
2   {
3      int i;
4
5      for (i=20; i>=0; i--) {
6         WaitForSingleObject(hMutex, INFINITE) ;
7         while(1) {
8                if (BufferState == FULL) {
9                   WaitForSingleObject(hNotFullEvent,INFINITE);
10                  continue; // back to loop to test BufferState again
11             }
12          break;
13       }
14       printf("Produce: %d\n", i);
15       SharedBuffer = i;
16       BufferState = FULL;
17       ReleaseMutex(hMutex);
18       PulseEvent(hNotEmptyEvent); // wake up consumer thread
19    }
20  }
```

In this program, a producer thread might enter its critical section and wait for the consumer to signal the event `hNotFullEvent`. The consumer thread, on the other hand, is waiting for the producer thread to leave its critical section before it can enter and make progress. Each thread is waiting for the other to make progress, so we have deadlock. Chapter 6 contains a more detailed discussion of deadlocks.

3.1.1 A Producer-Consumer Example—File Copy

To exemplify producers and consumers, we present a simple multithreaded file copy program (see Figure 3-3). The program spawns producer and consumer threads to perform the file copy; the threads share a common data buffer of `SIZE` bytes. The producer thread reads data from the original file up to `SIZE` bytes at a time into the shared buffer.

The consumer thread gets data from the shared buffer, and writes to a new file on disk.

```
1   #include <stdio.h>
2   #include <windows.h>
3
4   #define FULL 1
5   #define EMPTY 0
```

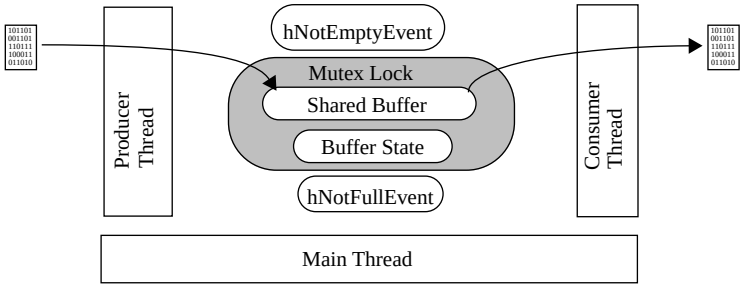

Figure 3-3. A File Copy Program Using Producer and Consumer Threads.

```
6   #define SIZE 10000
7
8   void *SharedBuffer[SIZE];
9   int BufferState;
10  HANDLE hMutex;
11  HANDLE hFullEvent, hEmptyEvent;
12  size_t nbyte = -1;
13
14  void Producer(FILE *infile)
15  {
16     size_t count;
17
18     do {
19        while(1){
20           WaitForSingleObject(hMutex,INFINITE);
21           if (BufferState == FULL) {
22              ReleaseMutex(hMutex);
23              // wait until buffer is not full
24              WaitForSingleObject(hEmptyEvent,INFINITE);
25              continue; // back to loop to test BufferState again
26           }
27           // got mutex and buffer is not FULL, break out of while loop
28           break;
29        }
30
31        // got Mutex, buffer is not full, producing data
32        nbyte = fread(SharedBuffer,1,SIZE,infile);
33        count = nbyte; // for use outside of critical section
34        printf("Produce %d bytes\n", nbyte);
35        BufferState = FULL;
36        ReleaseMutex(hMutex);  // end critical section
37        PulseEvent(hFullEvent); // wake up consumer thread
38     } while(count > 0);
39     printf("exit producer thread\n");
40  }
41
42  void Consumer(FILE *outfile)
43  {
44     while (1) {
```

```
45       WaitForSingleObject(hMutex,INFINITE);
46       if (nbyte == 0) {
47          printf("End of data, exit consumer thread\n");
48          ReleaseMutex(hMutex); // end critical section
49          ExitThread(0);
50       }
51
52       if (BufferState == EMPTY) { // nothing to consume
53          ReleaseMutex(hMutex); // release lock to wait
54          // wait until buffer is not empty
55          WaitForSingleObject(hFullEvent,INFINITE);
56       }
57       else {  // got Mutex, data in buffer, start consuming
58          fwrite(SharedBuffer,nbyte,1,outfile);
59          printf("Consumed: wrote %d bytes\n", nbyte);
60          BufferState = EMPTY;
61          ReleaseMutex(hMutex); // end critical section
62          PulseEvent(hEmptyEvent); // wake up producer thread
63       }
64    }
65 }
66
67 void main(int argc, char **argv)
68 {
69    HANDLE hThreadVector[2];
70    DWORD ThreadID;
71    FILE *infile, *outfile;
72
73    infile = fopen(argv[1],"rb");
74    outfile = fopen(argv[2],"wb");
75
76    BufferState = EMPTY;
77    hMutex = CreateMutex(NULL,FALSE,NULL);
78    hFullEvent  = CreateEvent(NULL,TRUE,FALSE,NULL);
79    hEmptyEvent = CreateEvent(NULL,TRUE,FALSE,NULL);
80
81    hThreadVector[0] = _beginthreadex (NULL, 0,
82         (LPTHREAD_START_ROUTINE)Producer,infile, 0,
83         (LPDWORD)&ThreadID);
84    hThreadVector[1] = _beginthreadex (NULL, 0,
85         (LPTHREAD_START_ROUTINE)Consumer, outfile, 0,
86         (LPDWORD)&ThreadID);
87    WaitForMultipleObjects(2,hThreadVector,TRUE,INFINITE);
88 }
```

3.2 The Bounded-Buffer Problem

The *bounded-buffer* problem is a natural extension of the producer-consumer problem, where producers and consumers share a set of buffers instead of just one. With multiple buffers, a producer does not necessarily have to wait for the last value to be consumed before producing another value. Similarly, a consumer is not forced to consume a single value each

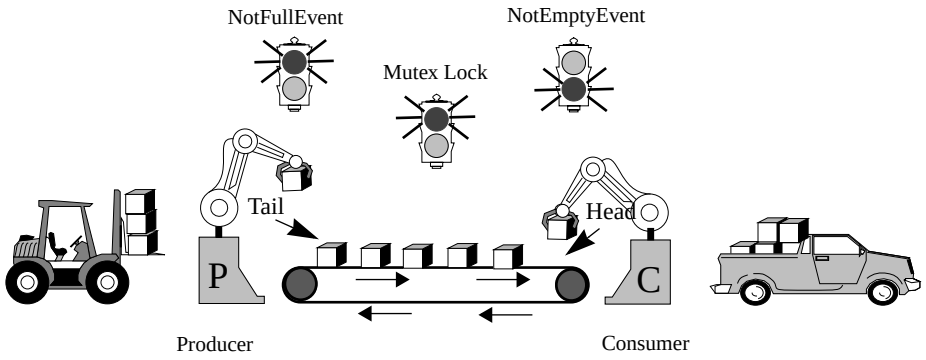

Figure 3-4. Producer and Consumer Robots with a Shared "Bounded Buffer."

time. This generalization enables producer and consumer threads to work much more efficiently by not having to wait for each other in lockstep.

Extending our analogy of producer and consumer robot arms, in the case of the bounded-buffer problem the shared pedestal is replaced by a conveyer belt that can carry more than one block at a time (Figure 3-4). The producer adds values at the **head** of the buffer, while the consumer consumes values from the **tail**. Each production or consumption moves the conveyor belt one step. The belt will lock when the buffer is full, and the conveyer belt stops and waits for a consumer to pick-up a block from it. Each time a block is picked up or a new block is produced, the belt moves forward one step. Therefore, consumers read values in the order in which they were produced.

A counter, named `count`, can be used to keep track of the number of buffers in use. As in the producer-consumer situation, we use two events, `hNotEmptyEvent` and `hNotFullEvent`, to synchronize the producer and consumer threads. Whenever a producer finds a full buffer space (`count == BUFSIZE`), it waits on the event `hNotFullEvent`. Similarly, when a consumer finds an empty buffer, it waits on the event `hNotEmptyEvent`. Whenever a producer writes a new value, it signals the event `hNotEmptyEvent` to awaken any waiting consumers. Likewise, when a consumer reads a value, it signals the event `hNotFullEvent` to wake any waiting producers. The following code illustrates this synchronization:

```
1   #include <iostream.h>
2   #include <windows.h>
3
4   #define BUFSIZE 5
5
6   int SharedBuffer[BUFSIZE];
7   int head,tail;
8   int count;
9
10  HANDLE hMutex;
11  HANDLE hNotFullEvent, hNotEmptyEvent;
12
13  void BB_Producer()
```

```
14  {
15    int i;
16
17    for (i=20; i>=0; i--) {
18      while(1) {
19        WaitForSingleObject(hMutex,INFINITE);
20        if (count == BUFSIZE) {  // buffer is full
21          ReleaseMutex(hMutex);
22          // wait until buffer is not full
23          WaitForSingleObject(hNotFullEvent,INFINITE);
24          continue; // back to loop to test buffer state again
25        }
26        // got mutex and buffer is not FULL, break out of while loop
27        break;
28      }
29
30      // got Mutex, buffer is not full, producing data
31      cout << "Produce: " << i << endl;
32      SharedBuffer[tail] = i;
33      tail = (tail+1) % BUFSIZE;
34      count++;
35      ReleaseMutex(hMutex);  // end critical section
36      PulseEvent(hNotEmptyEvent); // wake up consumer thread
37    }
38  }
39
40  void BB_Consumer()
41  {
42    int result;
43
44    while (1) {
45      WaitForSingleObject(hMutex,INFINITE);
46      if (count == 0) { // nothing to consume
47        ReleaseMutex(hMutex); // release lock to wait
48        // wait until buffer is not empty
49        WaitForSingleObject(hNotEmptyEvent,INFINITE);
50      }
51      else if (SharedBuffer[head] == 0) {// test for end of data token
52        cout << "Consumed 0: end of data" << endl;
53        ReleaseMutex(hMutex); // end critical section
54        ExitThread(0);
55      }
56      else {  // got Mutex, data in buffer, start consuming
57        result = SharedBuffer[head];
58        cout << "Consumed: " << result << endl;
59        head = (head+1) % BUFSIZE;
60        count--;
61        ReleaseMutex(hMutex); // end critical section
62        PulseEvent(hNotFullEvent); // wake up producer thread
63      }
64    }
65  }
66
67  void main()
```

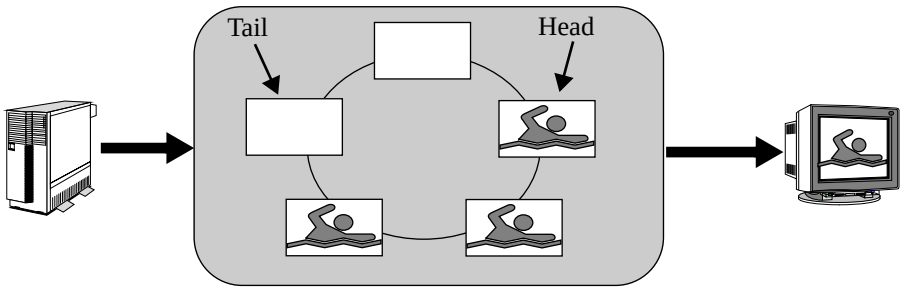

Figure 3-5. A Bounded-Buffer of Five Video Frames.

```
68  {
69     HANDLE hThreadVector[2];
70     DWORD ThreadID;
71
72     count = 0;
73     head = 0;
74     tail = 0;
75
76     hMutex = CreateMutex(NULL,FALSE,NULL);
77     hNotFullEvent  = CreateEvent(NULL,TRUE,FALSE,NULL);
78     hNotEmptyEvent = CreateEvent(NULL,TRUE,FALSE,NULL);
79
80     hThreadVector[0] = CreateThread (NULL, 0,
81            (LPTHREAD_START_ROUTINE) BB_Producer,
82            NULL, 0, (LPDWORD)&ThreadID);
83     hThreadVector[1] = CreateThread (NULL, 0,
84            (LPTHREAD_START_ROUTINE) BB_Consumer,
85            NULL, 0, (LPDWORD)&ThreadID);
86     WaitForMultipleObjects(2,hThreadVector,TRUE,INFINITE);
87  }
```

In general, this program resembles the producer-consumer example on page 56. The bounded-buffer program is, however, a more efficient way of sharing data when multiple values can be produced and consumed at the same time. In such cases, the producer-consumer program will force a thread context switch each time a new value is produced or consumed. In the bounded-buffer program, however, a producer thread can produce multiple values before it relinquishes processing to the consumer thread. Likewise, the consumer thread can consume multiple values during its time slice, when there is available data, before it is forced to context-switch.

An example of the bounded-buffer problem is a multimedia application that uncompresses and plays back video. Such an application might have two threads that manipulate a ring-buffer data structure, shown in Figure 3-5. One thread reads data from a disk, uncompresses, and fills each display buffer entry in memory, while another thread reads data from each buffer entry to display it. The two threads revolve around the ring supplying and getting data.

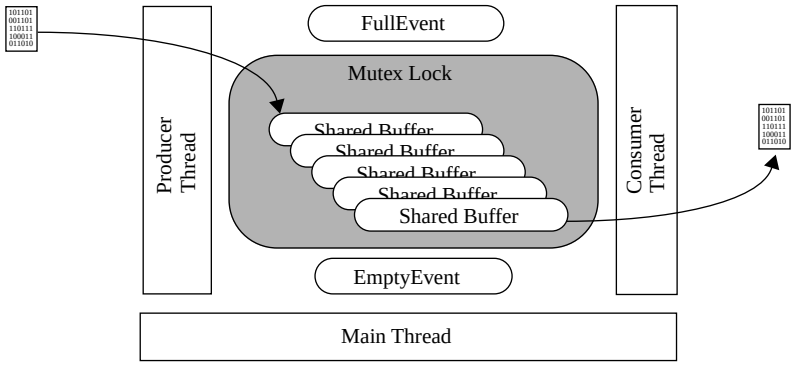

Figure 3-6. A File Copy Program Using Producer and Consumer Threads Sharing a Ring of Buffers.

3.2.1 Bounded-Buffer File Copy

With the bounded-buffer technique, we can now reimplement the multithreaded file copy program of Section 3.1.1 with several buffers instead of one (Figure 3-6).

```
1   #include <stdio.h>
2   #include <windows.h>
3   #include <stdlib.h>
4
5   #define SIZE 10000
6   #define BUFSIZE 5
7
8   typedef struct _Buffer {
9        void *Buf[SIZE];
10       size_t nbyte;
11  } Buffer;
12
13  Buffer *SharedBuffer[BUFSIZE];
14  int head, tail;
15  int count;
16
17  int TheEnd = FALSE;
18
19  HANDLE hMutex;
20  HANDLE hNotFullEvent, hNotEmptyEvent;
21  size_t nbyte;
22  void BB_Producer(FILE *infile)
23  {
24     size_t n;
25
26     do {
27        while(1){
28           WaitForSingleObject(hMutex,INFINITE);
29           if (count == BUFSIZE) {
30              ReleaseMutex(hMutex);
```

```
31            // wait until buffer is not full
32            WaitForSingleObject(hNotFullEvent,INFINITE);
33            continue; // back to loop to test buffer state again
34          }
35          // got mutex and buffer is not FULL, break out of while loop
36          break;
37        }
38
39        // got Mutex, buffer is not full, producing data
40        n = fread(SharedBuffer[tail]->Buf,1,SIZE,infile);
41        SharedBuffer[tail]->nbyte = n;
42        printf("Produce %d bytes\n",n);
43        tail = (tail+1) % BUFSIZE;
44        count++;
45        ReleaseMutex(hMutex);  // end critical section
46        PulseEvent(hNotEmptyEvent); // wake up consumer thread
47     } while(n > 0);
48     TheEnd = TRUE; // not thread safe here!
49     printf("exit producer thread\n");
50  }
51
52  void BB_Consumer(FILE *outfile)
53  {
54     While (1) {
55        WaitForSingleObject(hMutex,INFINITE);
56        if ((count == 0) && (TheEnd == TRUE)) {
57          printf("End of data, exit consumer thread\n");
58          ReleaseMutex(hMutex); // end critical section
59          ExitThread(0);
60        }
61
62        if (count == 0){ // nothing to consume
63          ReleaseMutex(hMutex); // release lock to wait
64          // wait until buffer is not empty
65          WaitForSingleObject(hNotEmptyEvent,INFINITE);
66          continue; // go back to while loop to try for mutex again
67        }
68
69        // got Mutex, data in buffer, start consuming
70        fwrite(SharedBuffer[head]->Buf,SharedBuffer[head]->nbyte,1,
71               outfile);
72        printf("Consumed: wrote %d bytes\n",SharedBuffer[head]->nbyte);
73        head = (head+1) % BUFSIZE;
74        count--;
75        ReleaseMutex(hMutex); // end critical section
76        PulseEvent(hNotFullEvent); // wake up producer thread
77     }
78  }
79
80  void main(int argc, char **argv)
81  {
82     HANDLE hThreadVector[2];
83     DWORD ThreadID;
84     FILE *infile, *outfile;
```

```
85     Buffer Buf0,Buf1,Buf2,Buf3,Buf4;
86
87     infile = fopen(argv[1],"rb");
88     outfile = fopen(argv[2],"wb");
89     count = 0;
90     head = 0;
91     tail = 0;
92     // create the bounded buffer
93     SharedBuffer[0] = &Buf0;
94     SharedBuffer[1] = &Buf1;
95     SharedBuffer[2] = &Buf2;
96     SharedBuffer[3] = &Buf3;
97     SharedBuffer[4] = &Buf4;
98     hMutex = CreateMutex(NULL,FALSE,NULL);
99     // create manual event objects
100    hNotFullEvent  = CreateEvent(NULL,TRUE,FALSE,NULL);
101    hNotEmptyEvent = CreateEvent(NULL,TRUE,FALSE,NULL);
102
103    hThreadVector[0] = _beginthreadex (NULL, 0,
104         (LPTHREAD_START_ROUTINE)BB_Producer, infile, 0,
105         (LPDWORD)&ThreadID);
106    hThreadVector[1] = _beginthreadex (NULL, 0,
107         (LPTHREAD_START_ROUTINE)BB_Consumer, outfile, 0,
108         (LPDWORD)&ThreadID);
109    WaitForMultipleObjects(2,hThreadVector,TRUE,INFINITE);
110 }
```

This example is more efficient than the producer-consumer file copy program with only one buffer, since producer and consumer threads don't necessarily proceed in lockstep. In this case, the granularity of locking is the entire set of buffers that is being shared among producer and consumer threads.

3.2.2 Bounded-Buffer with Finer Locking Granularity

To achieve even more concurrency, it is possible to implement a finer level of locking (Figure 3-7) where each buffer in the ring is locked individually. In this case, a producer can write data in one buffer while a consumer is reading from another. We get more concurrency at the expense of a little more complexity in managing the buffer locks. For this, we need two levels of mutex locks with one to protect the buffer state (i.e., a counter). In addition, there must be a mutex lock for each buffer in the ring. A producer or consumer thread must lock at the first level, then at the second level, increase the counter, and release the first-level lock so that other threads can proceed. Each thread produces or consumes data in the buffer it has locked, releases its lock, and then signals.

```
1   #include <stdio.h>
2   #include <windows.h>
3   #include <stdlib.h>
4
5   #define SIZE 10000
6   #define BUFSIZE 5
7
8   typedef struct _Buffer {
```

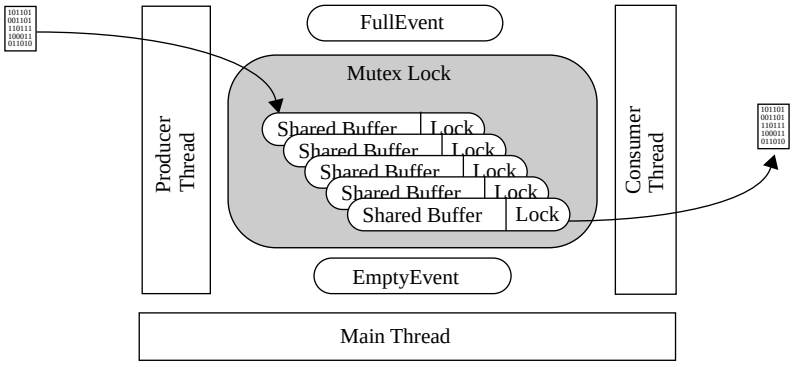

Figure 3-7. A File Copy Program with Individual Locks for Each Buffer in the Ring of Buffers.

```
9         void *Buf[SIZE];
10        HANDLE hBufferLock;
11        size_t nbyte;
12    } Buffer;
13
14    Buffer *SharedBuffer[BUFSIZE];
15    int head, tail;
16    int count;
17
18    int TheEnd;
19
20    HANDLE hMutex;
21    HANDLE hNotFullEvent, hNotEmptyEvent;
22    size_t nbyte;
23
24    void BB_Producer(FILE *infile)
25    {
26       size_t n;
27       int index;
28
29       do {
30          while(1){
31             // get lock for ring buffer
32             WaitForSingleObject(hMutex,INFINITE);
33             if (count == BUFSIZE) {
34                ReleaseMutex(hMutex);
35                // wait until buffer is not full
36                WaitForSingleObject(hNotFullEvent,INFINITE);
37                continue; // back to loop to test buffer state again
38             }
39             // got mutex and buffer is not FULL, break out of while loop
40             break;
41          }
42          // get lock for individual buffer, this is 2nd level lock
43          WaitForSingleObject(SharedBuffer[tail]->hBufferLock,INFINITE);
44          // got first and second level locks, update buffer state
```

```
45       //and release first level lock so other threads can access other
46       //parts of ring buffer
47       index = tail;
48       tail = (tail+1) % BUFSIZE;
49       count++;
50       ReleaseMutex(hMutex);  // release first level lock
51       // still have lock for this buffer, start producing data
52       n = fread(SharedBuffer[index]->Buf,1,SIZE,infile);
53       SharedBuffer[index]->nbyte = n;
54       printf("Produce %d bytes\n",n);
55       // release second level lock
56       ReleaseMutex(SharedBuffer[index]->hBufferLock);
57       PulseEvent(hNotEmptyEvent); // wake up consumer thread
58    } while(n > 0);
59    TheEnd = TRUE; // not thread safe here!
60    printf("exit producer thread\n");
61 }
62
63 void BB_Consumer(FILE *outfile)
64 {
65    int index;
66    while (1) {
67       WaitForSingleObject(hMutex,INFINITE);
68       if ((count == 0) && (TheEnd == TRUE)) {
69          printf("End of data, exit consumer thread\n");
70          ReleaseMutex(hMutex); // end critical section
71          ExitThread(0);
72       }
73       if (count == 0){ // nothing to consume
74          ReleaseMutex(hMutex); // release lock to wait
75          // wait until buffer is not empty
76          WaitForSingleObject(hNotEmptyEvent,INFINITE);
77          continue; // back to while loop and try for mutex again
78       }
79       // get lock for individual buffer, this is 2nd level lock
80       WaitForSingleObject(SharedBuffer[head]->hBufferLock,INFINITE);
81       // got first and second level locks, update buffer state
82       // and release first level lock so other threads can access
83       // other parts of ring buffer
84       index = head;
85       head = (head+1) % BUFSIZE;
86       count--;
87       ReleaseMutex(hMutex);  // release first level lock
88       // still have lock for this buffer, start consuming data
89       fwrite(SharedBuffer[index]->Buf, SharedBuffer[index]->nbyte,1,
90              outfile);
91       printf("Consumed:wrote %d bytes\n",SharedBuffer[index]->nbyte);
92       // release second level lock
93       ReleaseMutex(SharedBuffer[index]->hBufferLock);
94       PulseEvent(hNotFullEvent); // wake up producer thread
95    }
96 }
97
98 void main(int argc, char **argv)
```

```
99  {
100    HANDLE hThreadVector[2];
101    DWORD ThreadID;
102    FILE *infile, *outfile;
103    Buffer Buf0,Buf1,Buf2,Buf3,Buf4;
104    int i;
105
106    infile = fopen(argv[1],"rb");
107    outfile = fopen(argv[2],"wb");
108
109    count = 0;
110    head = 0;
111    tail = 0;
112    // create the bounded buffer
113    SharedBuffer[0] = &Buf0;
114    SharedBuffer[1] = &Buf1;
115    SharedBuffer[2] = &Buf2;
116    SharedBuffer[3] = &Buf3;
117    SharedBuffer[4] = &Buf4;
118
119    // create mutex lock for each buffer in buffer ring
120    for (i=0;i<BUFSIZE;i++)
121         SharedBuffer[i]->hBufferLock = CreateMutex(NULL,FALSE,NULL);
122
123    // create mutex lock for buffer ring
124    hMutex = CreateMutex(NULL,FALSE,NULL);
125
126    // create manual event object for signaling
127    hNotFullEvent  = CreateEvent(NULL,TRUE,FALSE,NULL);
128    hNotEmptyEvent = CreateEvent(NULL,TRUE,FALSE,NULL);
129
130    // create producer and consumer threads
131    hThreadVector[0] = _beginthreadex (NULL, 0,
132         (LPTHREAD_START_ROUTINE)BB_Producer, infile, 0,
133         (LPDWORD)&ThreadID);
134    hThreadVector[1] = _beginthreadex (NULL, 0,
135         (LPTHREAD_START_ROUTINE)BB_Consumer, outfile, 0,
136         (LPDWORD)&ThreadID);
137
138    // wait for producer and consumer thread to finish before
139    // terminating process
140    WaitForMultipleObjects(2,hThreadVector,TRUE,INFINITE);
141 }
```

This implementation is more efficient than the previous version because we spend little time locking the first-level lock. The lengthy update or read operations are synchronized only by the second-level lock, so many threads can read or write concurrently to different parts of the ring buffer. They will not deadlock because we have ordered the lock acquisitions (see Section 6.6.3).

3.3 The Readers-Writers Problem

When multiple threads share data, sometimes strict mutual exclusion is not necessary. This happens when there are several threads that only read a data value, and once in a while a thread writes into it. Such programming situations are examples of the *readers-writers* problem. For these, an exclusive single read-write protocol is too restrictive. Such applications can be made more efficient by allowing multiple simultaneous reads and an exclusive write. For example, consider a server program that maintains information on the latest prices for a variety of stocks. Suppose that this server program gets requests from several clients to either read or change the price of a stock. It is quite likely that the server will get many more requests to read stock prices than to change them. In this case, it is more efficient to let multiple readers obtain stock prices concurrently, without requiring mutual exclusion for each reader. When a thread wants to update a stock price, however, we have to make sure that no other thread is reading or writing it.

We now describe the multiple-readers/single-writer locking protocol in which there are two mutually exclusive phases of operation: the read phase and the write phase. During a read phase, all reader threads read the shared data in parallel. During a write phase, only one writer thread updates the shared data.

If a read phase is in progress, and we continue to allow newly arrived reader threads to join those in progress, it is possible that no writer thread can ever have a chance to run. Similarly, during a write phase, if newly arrived writer threads succeed other writer threads (one at a time) in updating the data, it is also possible that no reader thread ever gets a chance to run. This problem is known as starvation.

To prevent starvation of either reader or writer threads, the locking protocol alternates the read and write phases when necessary. For example, suppose there are both reader and writer threads present, at the start of a read phase all existing reader threads at that time are allowed to proceed to read the shared data concurrently. All reader threads that arrive after this time, while there are pending writer threads, are forced to wait until the next read phase. When all the current reader threads finish, a write phase begins in which one writer thread is allowed access to the shared data. Likewise, to prevent starvation of reader threads, other writer threads must wait until another write phase begins. And when the writer thread finishes, another read phase begins, which allows all the waiting reader threads at this time to proceed. The switching of read and write phases continues as long as there are pending reader and writer threads.

Things are simpler when we only have reader threads or writer threads. If we only have reader threads, there would be no write phase, and all of them are allowed to read the shared data in parallel. Likewise, if we only have writer threads, there will only be one write phase where one writer thread is allowed to follow another in updating the shared data, sequentially.

In an active system where there are many active reader and writer threads, one will see that the read and write phases alternate, allowing one batch of reader threads to proceed, followed by one writer thread, followed by another batch of (newly arrived) reader threads, followed by a writer thread, and so on.

To further illustrate this multiple-readers/single-writer synchronization protocol, Table 3–1 lists the arrival time of five reader threads and two writer threads. Assuming that

Arrival Time (ms)	Thread
t = 0	R1
t = 50	R2
t = 70	W1
t = 80	R3
t = 100	W2
t = 110	R4
t = 170	R5

Table 3–1. Arriving Reader and Writer Threads to Access Shared Data.

each read operation takes 40 milliseconds and each write operations takes 50 milliseconds, Figure 3-8 shows their behavior through time.

As you can see, the reader thread R1 arrives first. The protocol allows R1 to read the shared data immediately. When the reader thread R2 arrives, it immediately gets to read the shared data because R1 has finished and there are no waiting writer threads. There is no need to alternate the read phase and the write phase at this point because there are no pending writer threads.

The third thread that arrives is the writer thread W1. Since W1 arrives while the read phase is in progress (with thread R2), it waits until R2 completes. Now that there is a pending writer thread, when the read phase completes, a write phase will be started.

Shortly after that, at time t=80, a reader thread R3 arrives. Since there is a pending writer thread, the protocol does not allow this reader thread to join the read phase in progress. It must instead wait for the next read phase. This policy ensures that there is no starvation of writer threads.

When R2 completes at time t=90, a write phase begins with thread W1. While thread W1 is in progress, threads W2 and R4 arrive at times t=100 and t=110, respectively. Both W2 and R4 must wait. Thread W2 cannot execute because we allow only one writer thread in a write phase. Thread R4 cannot proceed because it has to wait for the next read phase.

When thread W1 completes at time t=140, a read phase begins and we allow both reader threads R3 and R4 to proceed concurrently. It is important to note that although thread W2 arrived before thread R4, thread R4 is allowed to access the shared data before W2 because both R3 and R4 are present at the start of the read phase.

When threads R3 and R4 are in progress, another reader thread R5 arrives. Since R5 misses the start of the current read phase, it must now wait for the next read phase.

When threads R3 and R4 complete, a write phase begins with thread W2. Finally when W2 finishes, a read phase begins and we allow thread R5 to run.

We now demonstrate the use of mutexes and semaphores to implement this protocol. Before reading data, each reader thread will call a function `start_reading` to make sure that it is allowed to do so. After reading, it will call the function `stop_reading` to

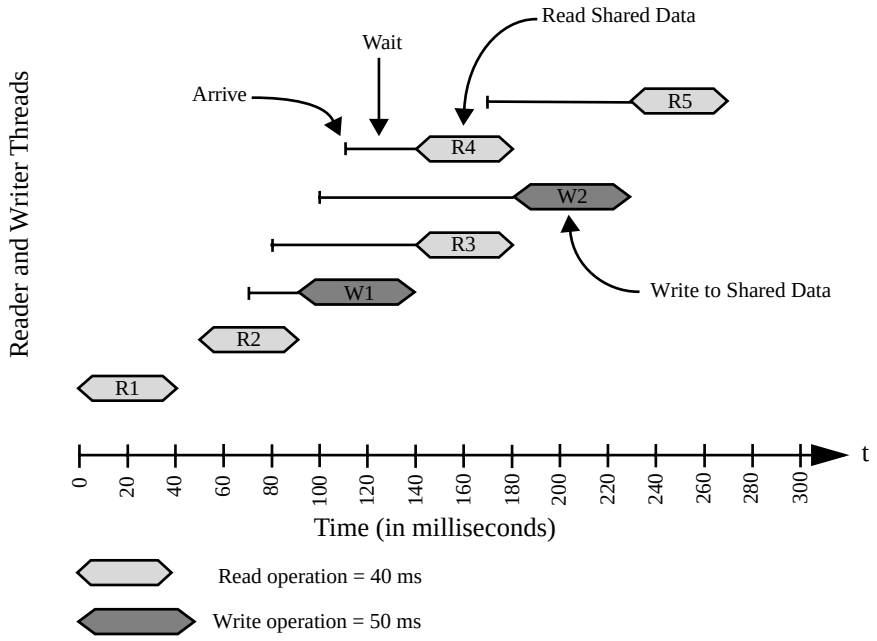

Figure 3-8. The Behavior of Reader and Writer Threads Operating Under the Multiple-Readers/ Single-Writer Synchronization Protocol.

announce that it has finished so that any waiting writer thread can be activated. Similarly, each writer thread will call functions `start_writing` and `stop_writing` before and after doing so. The following code implements this protocol:

```
1   #include <stdio.h>
2   #include <windows.h>
3   #include <stdlib.h>
4
5   #define MAX 9999
6   HANDLE mutex;          // init to count = 1, FALSE
7   HANDLE blockedReaders;  // init to count = 0, max = MAX
8   HANDLE blockedWriters;  // init to count = 0, max = MAX
9   int activeReaders = 0, waitingReader = 0;
10  int activeWriters = 0, waitingWriter = 0;
11  int sharedBuffer = 0;
12
13  VOID StartReading()
14  {
15       WaitForSingleObject(mutex, INFINITE);
16       // if there is active reader or waiting writer, wait for next
17       // read batch
18       if (activeWriters > 0 || waitingWriter > 0) {
19          waitingReader++;
20          ReleaseMutex(mutex);
```

```
21          WaitForSingleObject(blockedReaders, INFINITE);
22       }
23       else {
24          activeReaders++;
25          ReleaseMutex(mutex);
26       }
27  }
28
29  VOID StopReading()
30  {
31     WaitForSingleObject(mutex, INFINITE);
32     activeReaders--;
33     // last reader thread to finish reading needs to activate a
34     // waiting writer
35     if (activeReaders == 0 && waitingWriter > 0) {
36        activeWriters = 1;
37        waitingWriter--;
38        ReleaseSemaphore(blockedWriters, 1, NULL);
39     }
40     ReleaseMutex(mutex);
41  }
42
43  VOID StartWriting()
44  {
45     WaitForSingleObject(mutex, INFINITE);
46     if (activeReaders == 0 && activeWriters == 0) {
47        // there is no active reader or writer, OK to start writing
48        activeWriters = 1;
49        ReleaseMutex(mutex);
50     }
51     else {
52        // there is active readers or writer, put thread on write queue
53        waitingWriter++;
54        ReleaseMutex(mutex);
55        WaitForSingleObject(blockedWriters, INFINITE);
56     }
57  }
58
59  VOID StopWriting()
60  {
61     WaitForSingleObject(mutex, INFINITE);
62     activeWriters = 0;
63     if (waitingReader > 0) {
64     // if there are waiting readers, release them all from read queue
65        while (waitingReader > 0) {
66           waitingReader--;
67           activeReaders++;
68           ReleaseSemaphore(blockedReaders, 1, NULL);
69        }
70     }
71     else if (waitingWriter > 0) {
72        // no waiting reader and we have waiting writer,
73        // release 1 writer from write queue
74           waitingWriter--;
```

```
75          ReleaseSemaphore(blockedWriters, 1, NULL);
76       }
77       ReleaseMutex(mutex);
78    }
79
80    VOID Reader(LPVOID num)
81    {
82       int localVar, i;
83       for (i = 0; i<10;i++){
84          StartReading();
85          localVar = sharedBuffer;
86          cout << "Reader Thread " << num << " reads " << localVar << endl;
87          Sleep (1000);
88          StopReading();
89       }
90    }
91
92    VOID Writer(LPVOID num)
93    {
94       int i, j;
95
96       j = (int) num * 10;
97
98       for (i=j; i< j+5; i++){
99          StartWriting();
100         sharedBuffer = i;
101         cout << "\nWriter Thread " << num << " writes "
102            << sharedBuffer << endl;
103         Sleep (1000);
104         StopWriting();
105      }
106   }
107
108   main()
109   {
110      HANDLE hThreadVector[4];
111      DWORD ThreadID;
112      mutex = CreateMutex(NULL, FALSE, NULL);
113      blockedReaders = CreateSemaphore(NULL, 0, MAX, NULL);
114      blockedWriters = CreateSemaphore(NULL, 0, MAX, NULL);
115      hThreadVector[0] = CreateThread (NULL, 0,
116           (LPTHREAD_START_ROUTINE)Writer,(LPVOID) 1, 0,
117           (LPDWORD)&ThreadID);
118
119      hThreadVector[1] = CreateThread (NULL, 0,
120           (LPTHREAD_START_ROUTINE)Writer,(LPVOID) 2, 0,
121           (LPDWORD)&ThreadID);
122
123      hThreadVector[2] = CreateThread (NULL, 0,
124           (LPTHREAD_START_ROUTINE)Reader,(LPVOID) 3, 0,
125           (LPDWORD)&ThreadID);
126
127      hThreadVector[3] = CreateThread (NULL, 0,
128           (LPTHREAD_START_ROUTINE)Reader,(LPVOID) 4, 0,
```

```
129         (LPDWORD)&ThreadID);
130    WaitForMultipleObjects(4,hThreadVector,TRUE,INFINITE);
131 }
```

We maintain a count of the active readers and writers in the variables `activeReaders` and `activeWriters`, and a count of readers and writers that are currently blocked, waiting for a chance to read or write, in the variables `waitingReaders` and `waitingWriters`. For synchronizing concurrent access to these variables, we use the mutex `mutex`. Finally, we use two semaphore objects, `blockedReaders` and `blockedWriters`, to synchronize among reader and writer threads.

When a new reader wants access to start reading, it calls the function `start_reading`. If currently there is a writer active or blocked (line 18), then the reader will block on the semaphore `blockedReaders` after incrementing the count `blockedReaders` by 1. If there is no writer active or blocked, the reader increments the count for `activeReaders` by 1 and proceeds to read. When a reader has finished, it calls the function `stop_reading`. Here it checks to see if all readers have finished and if there is any waiting writer (line 35); if so, it wakes up the writer.

When a thread wants to write the shared variable, it calls the function `start_writing`. The writer proceeds to write if there are no active readers and writers (line 46). Otherwise, it increments the `waitingWriters` count by 1 and blocks on the semaphore `blockedWriters`. Finally, when a writer thread finishes, it calls the function `stop_writing`. Here it unblocks any readers waiting for the writer to finish. If there are none, the writer checks to see if there are any waiting writers; if so, it unblocks one of them.

Using the above solution, we can improve certain programs' responsiveness by having multiple concurrent readers, while preserving data integrity with exclusive writes.

3.4 Summary

In this chapter, we have demonstrated the use of synchronization objects to synchronize the execution of multiple threads. Synchronization protects data that is shared among threads. As we saw in the sample programs, failure to synchronize means we can get incorrect results from one thread accidently overwriting the work of another. We illustrated the use of simple mutual exclusion with critical sections, where only one thread is allowed to enter at a time. We then extended this to a producer-consumer situation where, in addition to effecting mutually exclusive access to shared data, threads inform each other about the state of the data using event objects. Finally, we introduced the situation where multiple readers can read from data concurrently, but only one thread is allowed to write to them. In the next chapter, we will see a more elegant solution to the synchronization problems using a concept called monitors.

3.5 Bibliography Notes

The mutual exclusion, producer-consumer, and bounded-buffer problems were all first introduced and solved in [Dijkstra, 1968].

3.6 Exercises

1. The event objects in the file copy program are created as manual-reset objects (lines 82–83). What are the implications if these were auto-reset objects instead?
2. The sleeping barber problem is another classic synchronization problem [Dijkstra, 1968]: There are a barber, a barber chair, and *n* waiting chairs. The barber works on one customer at a time, and he sleeps when he has no customers. When a potential customer arrives and there are no other customers, the customer wakes up the barber. If the barber is working on someone else, the customer waits in one of the waiting chairs, or leaves the shop if all chairs are occupied. When the barber finishes giving a haircut to one customer, he selects the first waiting customer. Write a program that simulates the barber and his customers.

   Refer to Figure 3-8 to answer the following questions.

3. From looking at the picture, which are the read phases? Which are the write phases? How can you tell (from the picture) that the read and write phases are mutually exclusive?
4. How long does the writer thread W2 have to wait before it is allowed to update the shared data?
5. At time t=130, what phase is in progress (read or write)? How many threads are waiting? Which ones?
6. At time t=90, both reader thread R3 and writer thread W1 are waiting. Why was W1 chosen to run and not R3?
7. Thread W2 arrived before thread R4, but why does R4 access shared data before W2?
8. Thread R3 arrives during a read phase. Why isn't it allowed to join the readers in progress? What will happen if we change the protocol to allow R3 to join this read phase immediately?

3.7 References

1. Dijkstra, E. W. (1968). Cooperating sequential processes. In Genuys, F., ed., *Programming Languages*. Academic Press, Reading, MA.