23.8 (Advanced) Producer/Consumer Relationship: Bounded Buffers

The program in Section 23.7 uses thread synchronization to guarantee that two threads manipulate data in a shared buffer correctly. However, the application may not perform optimally. If the two threads operate at different speeds, one of them will spend more (or most) of its time waiting. For example, in the program in Section 23.7 we shared a single integer variable between the two threads. If the `Producer` thread produces values *faster* than the `Consumer` can consume them, then the `Producer` thread *waits* for the `Consumer`, because there are no other locations in the buffer in which to place the next value. Similarly, if the `Consumer` consumes values *faster* than the `Producer` produces them, the `Consumer` *waits* until the `Producer` places the next value in the shared buffer. Even when we have threads that operate at the *same* relative speeds, those threads may occasionally become “out of sync” over a period of time, causing one of them to *wait* for the other.

Performance Tip 23.3

We cannot make assumptions about the relative speeds of concurrent threads—*interactions that occur with the operating system, the network, the user and other components can cause the threads to operate at different and ever-changing speeds. When this happens, threads wait. When threads wait excessively, programs become less efficient, interactive programs become less responsive and applications suffer longer delays.*

Bounded Buffers

To minimize the amount of waiting time for threads that share resources and operate at the same average speeds, we can implement a **bounded buffer** that provides a fixed number of buffer cells into which the `Producer` can place values, and from which the `Consumer` can retrieve those values. (In fact, the `ArrayBlockingQueue` class in Section 23.6 is a bounded buffer.) If the `Producer` temporarily produces values faster than the `Consumer` can consume them, the `Producer` can write additional values into the extra buffer cells, if any are available. This capability enables the `Producer` to perform its task even though the `Consumer` is not ready to retrieve the current value being produced. Similarly, if the `Consumer` temporarily consumes faster than the `Producer` produces new values, the `Consumer` can read additional values (if there are any) from the buffer. This

enables the `Consumer` to keep busy even though the `Producer` is not ready to produce additional values. An example of the producer/consumer relationship that uses a bounded buffer is video streaming, which we discussed in Section 23.1.

Even a *bounded buffer* is inappropriate if the `Producer` and the `Consumer` operate consistently at different speeds. If the `Consumer` always executes faster than the `Producer`, then a buffer containing one location is enough. If the `Producer` always executes faster, only a buffer with an "infinite" number of locations would be able to absorb the extra production. However, if the `Producer` and `Consumer` execute at about the same average speed, a bounded buffer helps to smooth the effects of any occasional speeding up or slowing down in either thread's execution.

The key to using a *bounded buffer* with a `Producer` and `Consumer` that operate at about the same speed is to provide the buffer with enough locations to handle the anticipated "extra" production. If, over a period of time, we determine that the `Producer` often produces as many as three more values than the `Consumer` can consume, we can provide a buffer of at least three cells to handle the extra production. Making the buffer too small would cause threads to wait longer.

[*Note:* As we mention in Fig. 23.22, `ArrayBlockingQueue` can work with multiple producers and multiple consumers. For example, a factory that produces its product very fast will need to have many more delivery trucks (i.e., consumers) to remove those products quickly from

the warehousing area (i.e., the bounded buffer) so that the factory can continue to produce products at full capacity.]

Performance Tip 23.4

Even when using a bounded buffer*, it's possible that a producer thread could fill the buffer, which would force the producer to wait until a consumer consumed a value to free an element in the buffer. Similarly, if the buffer is empty at any given time, a consumer thread must wait until the producer produces another value. The key to using a bounded buffer is to optimize the buffer size to minimize the amount of thread wait time, while not wasting space.*

Bounded Buffers Using `ArrayBlockingQueue`

The simplest way to implement a bounded buffer is to use an `ArrayBlockingQueue` for the buffer so that *all of the synchronization details are handled for you*. This can be done by modifying the example from Section 23.6 to pass the desired size for the bounded buffer into the `ArrayBlockingQueue` constructor. Rather than repeat our previous `ArrayBlockingQueue` example with a different size, we instead present an example that illustrates how you can build a bounded buffer yourself. Again, using an `ArrayBlockingQueue` will result in more-maintainable

and better-performing code. In Exercise 23.13, we ask you to reimplement this section's example, using the Java Concurrency API techniques presented in Section 23.9.

Implementing Your Own Bounded Buffer as a Circular Buffer

The program in Figs. 23.18 and 23.19 demonstrates a `Producer` and a `Consumer` accessing a *bounded buffer with synchronization*. Again, we reuse interface `Buffer` and classes `Producer` and `Consumer` from the example in Section 23.5, except that line 24 is removed from class `Producer` and class `Consumer`. We implement the bounded buffer (Fig. 23.18) as a **circular buffer** that uses a shared array of three elements. A circular buffer writes into and reads from the array elements in order, beginning at the first cell and moving toward the last. When a `Producer` or `Consumer` reaches the last element, it returns to the first and begins writing or reading, respectively, from there. In this version of the producer/consumer relationship, the `Consumer` consumes a value only when the array is not empty and the `Producer` produces a value only when the array is not full. Once again, the output statements used in this class's `synchronized` methods are for *demonstration purposes only*.

```
1   // Fig. 23.18: CircularBuffer.java
2   // Synchronizing access to a shared three-elemen
```

```
3  public class CircularBuffer implements Buffer {
4     private final int[] buffer = {-1, -1, -1}; //
5
6     private int occupiedCells = 0; // count numbe
7     private int writeIndex = 0; // index of next
8     private int readIndex = 0; // index of next e
9
10    // place value into buffer
11    @Override
12    public synchronized void blockingPut(int valu
13       throws InterruptedException {
14
15       // wait until buffer has space available,
16       // while no empty locations, place thread
17       while (occupiedCells == buffer.length) {
18          System.out.printf("Buffer is full. Prod
19          wait(); // wait until a buffer cell is
20       }
21
22       buffer[writeIndex] = value; // set new buf
23
24       // update circular write index
25       writeIndex = (writeIndex + 1) % buffer.len
26
27       ++occupiedCells; // one more buffer cell i
28       displayState("Producer writes " + value);
29       notifyAll(); // notify threads waiting to
30    }
31
32    // return value from buffer
33    @Override
34    public synchronized int blockingGet() throws
35       // wait until buffer has data, then read v
36       // while no data to read, place thread in
37       while (occupiedCells == 0) {
38          System.out.printf("Buffer is empty. Con
39          wait(); // wait until a buffer cell is
40       }
41
42       int readValue = buffer[readIndex]; // read
```

```
                    43
     44         // update circular read index
45         readIndex = (readIndex + 1) % buffer.lengt
                    46
47         --occupiedCells; // one fewer buffer cells
48         displayState("Consumer reads " + readValue
49         notifyAll(); // notify threads waiting to
                    50
          51         return readValue;
                   52      }
                    53
54      // display current operation and buffer state
55      public synchronized void displayState(String
56         // output operation and number of occupied
57         System.out.printf("%s%s%d)%n%s", operation
58            " (buffer cells occupied: ", occupiedCe
                    59
      60         for (int value : buffer) {
61            System.out.printf(" %2d  ", value); //
                   62         }
                    63
64         System.out.printf("%n               ");
                    65
66         for (int i = 0; i < buffer.length; i++) {
     67            System.out.print("---- ");
                   68         }
                    69
70         System.out.printf("%n               ");
                    71
72         for (int i = 0; i < buffer.length; i++) {
73            if (i == writeIndex && i == readIndex)
74               System.out.print(" WR"); // both wri
                  75            }
    76             else if (i == writeIndex) {
77               System.out.print(" W  "); // just wr
                  78            }
     79             else if (i == readIndex) {
80               System.out.print("  R  "); // just r
                  81            }
                82            else {
```

```
83                System.out.print("  "); // neither i
                  84             }
                    85          }
                          86
          87          System.out.printf("%n%n");
                       88       }
                         89    }
```

Fig. 23.18

Synchronizing access to a shared three-element bounded buffer.

Line 4 initializes array `buffer` as a three-element `int` array that represents the circular buffer. Variable `occupiedCells` (line 6) counts the number of elements in `buffer` that contain data to be read. When `occupiedCells` is `0`, the circular buffer is *empty* and the `Consumer` must *wait*—when `occupiedCells` is `3` (the size of the circular buffer), the circular buffer is *full* and the `Producer` must *wait*. Variable `writeIndex` (line 7) indicates the next location in which a value can be placed by a `Producer`. Variable `readIndex` (line 8) indicates the position from which the next value can be read by a `Consumer`. `CircularBuffer`'s instance variables are *all* part of the class's shared mutable data, thus access to all of these variables must be synchronized to ensure that a `CircularBuffer` is thread safe.

`CircularBuffer` Method `blockingPut`

`CircularBuffer` method `blockingPut` (lines 11–30) performs the same tasks as in Fig. 23.16, with a few modifications. The loop at lines 17–20 of Figs. 23.18 determines whether the `Producer` must *wait* (i.e., all buffer cells are *full*). If so, line 18 indicates that the `Producer` is *waiting* to perform its task. Then line 19 invokes method `wait`, causing the `Producer` thread to *release* the `CircularBuffer`'s *lock* and *wait* until there's space for a new value to be written into the buffer. When execution continues at line 22 after the `while` loop, the value written by the `Producer` is placed in the circular buffer at location `writeIndex`. Then line 25 updates `writeIndex` for the next call to `CircularBuffer` method `blockingPut`. This line is the key to the buffer's *circularity*. When `writeIndex` is incremented *past the end of the buffer*, the line sets it to `0`. Line 27 increments `occupiedCells`, because there's now one more value in the buffer that the `Consumer` can read. Next, line 28 invokes method `displayState` (lines 55–88) to update the output with the value produced, the number of occupied buffer cells, the contents of the buffer cells and the current `writeIndex` and `readIndex`. Line 29 invokes method `notifyAll` to transition *waiting* threads to the *runnable* state, so that a waiting `Consumer` thread (if there is one) can now try again to read a value from the buffer.

`CircularBuffer` Method `blockingGet`

`CircularBuffer` method `blockingGet` (lines 33–52) also performs the same tasks as it did in Fig. 23.16, with a few minor modifications. The loop at lines 37–40 (Fig. 23.18) determines whether the `Consumer` must wait (i.e., all buffer cells are *empty*). If the `Consumer` must *wait*, line 38 updates the output to indicate that the `Consumer` is *waiting* to perform its task. Then line 39 invokes method `wait`, causing the current thread to *release the lock* on the `CircularBuffer` and *wait* until data is available to read. When execution eventually continues at line 42 after a `notifyAll` call from the `Producer`, `readValue` is assigned the value at location `readIndex` in the circular buffer. Then line 45 updates `readIndex` for the next call to `CircularBuffer` method `blockingGet`. This line and line 25 implement the *circularity* of the buffer. Line 47 decrements `occupiedCells`, because there's now one more position in the buffer in which the `Producer` thread can place a value. Line 48 invokes method `displayState` to update the output with the consumed value, the number of occupied buffer cells, the contents of the buffer cells and the current `writeIndex` and `readIndex`. Line 49 invokes method `notifyAll` to allow any `Producer` threads *waiting to write* into the `CircularBuffer` object to attempt to write again. Then line 51 returns the consumed value to the caller.

`CircularBuffer` Method `displayState`

Method `displayState` (lines 55–88) outputs the application's state. Lines 60–62 output the values of the buffer cells, using a `"%2d"` format specifier to print the contents of each buffer with a leading space if it's a single digit. Lines 72–85 output the current `writeIndex` and `readIndex` with the letters `when` and `R`, respectively. Once again, `displayState` is a synchronized method because it accesses class `CircularBuffer`'s shared mutable data.

Testing Class `CircularBuffer`

Class `CircularBufferTest` (Fig. 23.19) contains the `main` method that launches the application. Line 10 creates the `ExecutorService`, and line 13 creates a `CircularBuffer` object and assigns its reference to `CircularBuffer` variable `sharedLocation`. Line 16 invokes the `CircularBuffer`'s `displayState` method to show the initial state of the buffer. Lines 19–20 execute the `Producer` and `Consumer` tasks. Line 22 calls method `shutdown` to end the application when the threads complete the `Producer` and `Consumer` tasks, and line 23 waits for the tasks to complete.

```
1   // Fig. 23.19: CircularBufferTest.java
2   // Producer and Consumer threads correctly manipu
3   import java.util.concurrent.ExecutorService;
4   import java.util.concurrent.Executors;
5   import java.util.concurrent.TimeUnit;
6
7   public class CircularBufferTest {
8      public static void main(String[] args) throws
9         // create new thread pool
10        ExecutorService executorService = Executor
11
12        // create CircularBuffer to store ints
13        CircularBuffer sharedLocation = new Circul
14
15        // display the initial state of the Circul
16        sharedLocation.displayState("Initial State
17
18        // execute the Producer and Consumer tasks
19        executorService.execute(new Producer(share
20        executorService.execute(new Consumer(share
21
22        executorService.shutdown();
23        executorService.awaitTermination(1, TimeUn
24     }
25  }
```

```
Initial State (buffer cells occupied: 0)
      buffer cells:   -1   -1   -1
                     ---- ---- ----
                           WR

Producer writes 1 (buffer cells occupied: 1)
      buffer cells:    1   -1   -1
                     ---- ---- ----
                          R   W

Consumer reads 1 (buffer cells occupied: 0)
```

```
        buffer cells:    1   -1   -1
                       ---- ---- ----
                              WR

      Buffer is empty. Consumer waits.
Producer writes 2 (buffer cells occupied: 1)
        buffer cells:    1   2   -1
                       ---- ---- ----
                              R   W

Consumer reads 2 (buffer cells occupied: 0)
        buffer cells:    1   2   -1
                       ---- ---- ----
                                  WR

Producer writes 3 (buffer cells occupied: 1)
        buffer cells:    1   2   3
                       ---- ---- ----
                         W          R

Consumer reads 3 (buffer cells occupied: 0)
        buffer cells:    1   2   3
                       ---- ---- ----
                             WR
```


```
Producer writes 4 (buffer cells occupied: 1)
        buffer cells:    4    2    3
                       ---- ---- ----
                            R   W

Producer writes 5 (buffer cells occupied: 2)
        buffer cells:    4    5    3
                       ---- ---- ----
                           R        W

Consumer reads 4 (buffer cells occupied: 1)
        buffer cells:    4    5    3
                       ---- ---- ----
```

```
                                R   W

Producer writes 6 (buffer cells occupied: 2)
        buffer cells:    4    5    6
                        ---- ---- ----
                            W     R

Producer writes 7 (buffer cells occupied: 3)
        buffer cells:    7    5    6
                        ---- ---- ----
                                 WR

 Consumer reads 5 (buffer cells occupied: 2)
        buffer cells:    7    5    6
                        ---- ---- ----
                              W     R

Producer writes 8 (buffer cells occupied: 3)
        buffer cells:    7    8    6
                        ---- ---- ----
                                   WR

 Consumer reads 6 (buffer cells occupied: 2)
        buffer cells:    7    8    6
                        ---- ---- ----
                           R        W

 Consumer reads 7 (buffer cells occupied: 1)
        buffer cells:    7    8    6
                        ---- ---- ----
                                 R   W

Producer writes 9 (buffer cells occupied: 2)
        buffer cells:    7    8    9
                        ---- ---- ----
                            W     R

 Consumer reads 8 (buffer cells occupied: 1)
        buffer cells:    7    8    9
                        ---- ---- ----
```

```
                              W          R
```

```
      Consumer reads 9 (buffer cells occupied: 0)
             buffer cells:    7    8    9
                            ---- ---- ----
                                 WR

     Producer writes 10 (buffer cells occupied: 1)
             buffer cells:    10    8    9
                            ---- ---- ----
                                 R   W

                Producer done producing
                  Terminating Producer
     Consumer reads 10 (buffer cells occupied: 0)
             buffer cells:    10    8    9
                            ---- ---- ----
                                   WR

           Consumer read values totaling: 55
                 Terminating Consumer
```

Fig. 23.19

`Producer` and `Consumer` threads correctly manipulating a circular buffer.

Each time the `Producer` writes a value or the `Consumer` reads a value, the program outputs a message indicating the action performed (a read or a write), the contents of `buffer`, and the location of `writeIndex` and `readIndex`. In the

output of Fig. 23.19, the `Producer` first writes the value `1`. The buffer then contains the value `1` in the first cell and the value `-1` (the default value that we use for output purposes) in the other two cells. The write index is updated to the second cell, while the read index stays at the first cell. Next, the `Consumer` reads `1`. The buffer contains the same values, but the read index has been updated to the second cell. The `Consumer` then tries to read again, but the buffer is empty and the `Consumer` is forced to wait. Only once in this execution of the program was it necessary for either thread to wait.