23.5 Producer/Consumer Relationship without Synchronization

In a **producer/consumer relationship**, the **producer** portion of an application generates data and *stores it in a shared object*, and the **consumer** portion of the application *reads data from the shared object*. The producer/consumer relationship separates the task of identifying work to be done from the tasks involved in actually carrying out the work.

Examples of Producer/Consumer Relationship

One example of a common producer/consumer relationship is **print spooling**. Although a printer might not be available when you want to print from an application (i.e., the producer), you can still “complete” the print task, as the data is temporarily placed on disk until the printer becomes available. Similarly, when the printer (i.e., a consumer) is available, it doesn’t have to wait until a current user wants to print. The spooled print jobs can be printed as soon as the printer becomes available. Another example of the producer/consumer relationship is an application that copies data onto DVDs by placing data in a fixed-size buffer, which is emptied as the DVD drive "burns" the data onto the DVD.

Synchronization and State Dependence

In a multithreaded producer/consumer relationship, a **producer thread** generates data and places it in a shared object called a **buffer**. A **consumer thread** reads data from the buffer. This relationship requires *synchronization* to ensure that values are produced and consumed properly. All operations on *mutable* data that's shared by multiple threads (e.g., the data in the buffer) must be guarded with a lock to prevent corruption, as discussed in Section 23.4. Operations on the buffer data shared by a producer and consumer thread are also **state dependent**—the operations should proceed only if the buffer is in the correct state. If the buffer is in a *not-full state*, the producer may produce; if the buffer is in a *not-empty state*, the consumer may consume. All operations that access the buffer must use synchronization to ensure that data is written to the buffer or read from the buffer only if the buffer is in the proper state. If the producer attempting to put the next data into the buffer determines that it's full, the producer thread must *wait* until there's space to write a new value. If a consumer thread finds the buffer empty or finds that the previous data has already been read, the consumer must also *wait* for new data to become available. Other examples of state dependence are that you can't drive your car if its gas tank is empty and you can’t put more gas into the tank if it’s already full.

Logic Errors from Lack of Synchronization

Consider how logic errors can arise if we do not synchronize access among multiple threads manipulating shared mutable data. Our next example (Figs. 23.9–23.13) implements a producer/consumer relationship *without the proper synchronization*. A producer thread writes the numbers 1 through 10 into a shared buffer—a single memory location shared between two threads (a single `int` variable called `buffer` in line 5 of Fig. 23.12 in this example). The consumer thread reads this data from the shared buffer and displays the data. The program’s output shows the values that the producer writes (produces) into the shared buffer and the values that the consumer reads (consumes) from the shared buffer.

Each value the producer thread writes to the shared buffer must be consumed *exactly once* by the consumer thread. However, the threads in this example are not synchronized. Therefore, *data can be lost or garbled if the producer places new data into the shared buffer before the consumer reads the previous data*. Also, data can be incorrectly *duplicated* if the consumer consumes data again before the producer produces the next value. To show these possibilities, the consumer thread in the following example keeps a total of all the values it reads. The producer thread produces values from 1 through 10. If the consumer reads each value produced once and only once, the total will be 55. However, if you execute this program several times, you'll see that the total is not always 55 (as shown in the outputs in Fig. 23.13). To emphasize the point, the producer and consumer threads in the example each sleep for random intervals of up to three seconds between performing their tasks. Thus, we do not know when the producer thread will attempt to write a new value, or when the consumer thread will attempt to read a value.

Interface `Buffer`

The program consists of interface `Buffer` (Fig. 23.9) and classes `Producer` (Fig. 23.10), `Consumer` (Fig. 23.11), `UnsynchronizedBuffer` (Fig. 23.12) and `SharedBufferTest` (Fig. 23.13). Interface `Buffer` (Fig. 23.9) declares methods `blockingPut` (line 5) and `blockingGet` (line 8) that a `Buffer` (such as `UnsynchronizedBuffer`) must implement to enable the `Producer` thread to place a value in the `Buffer` and the `Consumer` thread to retrieve a value from the `Buffer`, respectively. In subsequent examples, methods `blockingPut` and `blockingGet` will call methods that throw `InterruptedException`s—typically this indicates that a method temporarily could be blocked from performing a task. We declare each method with a `throws` clause here so that we don't have to modify this interface for the later examples.

```
// Fig. 23.9: Buffer.java
// Buffer interface specifies methods called by P
public interface Buffer {
   // place int value into Buffer
   public void blockingPut(int value) throws Inte

   // return int value from Buffer
   public int blockingGet() throws InterruptedExc
}
```

Fig. 23.9

Buffer interface specifies methods called by `Producer` and `Consumer`. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.)

Class `Producer`

Class `Producer` (Fig. 23.10) implements the `Runnable` interface, allowing it to be executed as a task in a separate thread. The constructor (lines 10–12) initializes the `Buffer` reference `sharedLocation` with an object created in `main` (line 13 of Fig. 23.13) and passed to the constructor. As we'll see, this is an `UnsynchronizedBuffer` object that implements interface `Buffer` *without synchronizing access to the shared object*.

```java
// Fig. 23.10: Producer.java
// Producer with a run method that inserts the v
import java.security.SecureRandom;

public class Producer implements Runnable{
   private static final SecureRandom generator =
   private final Buffer sharedLocation; // refer

   // constructor
   public Producer(Buffer sharedLocation) {
      this.sharedLocation = sharedLocation;
   }

   // store values from 1 to 10 in sharedLocatio
   @Override
   public void run() {
      int sum = 0;

      for (int count = 1; count <= 10; count++)
         try { // sleep 0 to 3 seconds, then pla
            Thread.sleep(generator.nextInt(3000)
            sharedLocation.blockingPut(count); /
            sum += count; // increment sum of va
            System.out.printf("\t%2d%n", sum);
         }
         catch (InterruptedException exception)
            Thread.currentThread().interrupt();
         }
      }

      System.out.printf(
         "Producer done producing%nTerminating P
   }
}
```

Fig. 23.10

`Producer` with a `run` method that inserts the values 1 to 10 in buffer. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.)

The `Producer` thread in this program executes the tasks specified in the method `run` (Fig. 23.10, lines 15–33). Each iteration of its loop invokes `Thread` method `sleep` (line 21) to place the `Producer` thread into the *timed waiting* state for a random time interval between 0 and 3 seconds. When the thread awakens, line 22 passes the value of control variable `count` to the `Buffer` object's `blockingPut` method to set the shared buffer's value. Lines 23–24 keep a total of all the values produced so far and output that value. When the loop completes, lines 31–32 display a message indicating that the `Producer` has finished producing data and is terminating. Next, method `run` terminates, which indicates that the `Producer` completed its task. Any method called from a `Runnable`'s `run` method (e.g., `Buffer` method `blockingPut`) executes as part of that task's thread of execution. This fact becomes important in Sections 23.6–23.8 when we add synchronization to the producer/consumer relationship.

Class `Consumer`

Class `Consumer` (Fig. 23.11) also implements interface

Runnable, allowing the Consumer to execute concurrently with the Producer. Lines 10–12 initialize Buffer reference sharedLocation with an object that implements the Buffer interface (created in main, Fig. 23.13) and passed to the constructor as the parameter sharedLocation. As we’ll see, this is the same UnsynchronizedBuffer object that’s used to initialize the Producer object—thus, the two threads share the same object. The Consumer thread in this program performs the tasks specified in method run. Lines 19–29 in Fig. 23.11 iterate 10 times. Each iteration invokes Thread method sleep (line 22) to put the Consumer thread into the *timed waiting* state for up to 3 seconds. Next, line 23 uses the Buffer’s blockingGet method to retrieve the value in the shared buffer, then adds the value to variable sum. Line 24 displays the total of all the values consumed so far. When the loop completes, lines 31–32 display the sum of the consumed values. Then method run terminates, which indicates that the Consumer completed its task. Once both threads enter the *terminated* state, the program ends.

```
1   // Fig. 23.11: Consumer.java
2   // Consumer with a run method that loops, reading
3   import java.security.SecureRandom;
4
5   public class Consumer implements Runnable {
6      private static final SecureRandom generator =
7      private final Buffer sharedLocation; // refere
8
9      // constructor
10     public Consumer(Buffer sharedLocation) {
11        this.sharedLocation = sharedLocation;
```

```
   }

   // read sharedLocation's value 10 times and s
   @Override
   public void run() {
      int sum = 0;

      for (int count = 1; count <= 10; count++)
         // sleep 0 to 3 seconds, read value fro
         try {
            Thread.sleep(generator.nextInt(3000)
            sum += sharedLocation.blockingGet();
            System.out.printf("\t\t\t%2d%n", sum
         }
         catch (InterruptedException exception)
            Thread.currentThread().interrupt();
         }
      }

      System.out.printf("%n%s %d%n%s%n",
         "Consumer read values totaling", sum, "
   }
}
```

Fig. 23.11

`Consumer` with a `run` method that loops, reading 10 values from buffer. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.)

We Call `Thread` Method

`sleep` Only for Demonstration Purposes

We call method `sleep` in method `run` of the `Producer` and `Consumer` classes to emphasize the fact that, *in multithreaded applications, it's unpredictable when each thread will perform its task and for how long it will perform the task when it has a processor*. Normally, these thread-scheduling issues are beyond the control of the Java developer. In this program, our thread's tasks are quite simple—the `Producer` writes the values `1` to `10` to the buffer, and the `Consumer` reads 10 values from the buffer and adds each value to variable `sum`. Without the `sleep` method call, and if the `Producer` executes first, given today's phenomenally fast processors, the `Producer` would likely complete its task before the `Consumer` got a chance to execute. If the `Consumer` executed first, it would likely consume garbage data ten times, then terminate before the `Producer` could produce the first real value.

Class `UnsynchronizedBuffer` Does Not Synchronize Access to the Buffer

Class `UnsynchronizedBuffer` (Fig. 23.12) implements interface `Buffer` (line 4), but does *not* synchronize access to the buffer’s state—we purposely do this to demonstrate the problems that occur when multiple threads access *shared* mutable data *without* synchronization. Line 5 declares instance variable `buffer` and initializes it to `–1`. This value is used to demonstrate the case in which the `Consumer` attempts to consume a value *before* the `Producer` ever places a value in `buffer`. Again, methods `blockingPut` (lines 8–12) and `blockingGet` (lines 15–19) do *not* synchronize access to the `buffer` instance variable. Method `blockingPut` simply assigns its argument to `buffer` (line 11), and method `blockingGet` simply returns the value of `buffer` (line 18). As you’ll see in Fig. 23.13, `UnsynchronizedBuffer` object is shared between the `Producer` and the `Consumer`.

```
 1  // Fig. 23.12: UnsynchronizedBuffer.java
 2  // UnsynchronizedBuffer maintains the shared int
 3  // a producer thread and a consumer thread.
 4  public class UnsynchronizedBuffer implements Buf
 5     private int buffer = -1; // shared by produce
 6
 7     // place value into buffer
 8     @Override
 9     public void blockingPut(int value) throws Int
10        System.out.printf("Producer writes\t%2d",
11        buffer = value;
12     }
13
14     // return value from buffer
15     @Override
16     public int blockingGet() throws InterruptedEx
17        System.out.printf("Consumer reads\t%2d", b
18        return buffer;
19     }
```

```
}
```

Fig. 23.12

`UnsynchronizedBuffer` maintains the shared integer that is accessed by a producer thread and a consumer thread. (*Caution:* The example of Fig. 23.9–Fig. 23.13 is *not* thread safe.)

Class `SharedBufferTest`

In class `SharedBufferTest` (Fig. 23.13), line 10 creates an `ExecutorService` to execute the `Producer` and `Consumer Runnable`s. Line 13 creates an `UnsynchronizedBuffer` and assigns it to `Buffer` variable `sharedLocation`. This object stores the data that the `Producer` and `Consumer` threads will share. Lines 22–23 create and execute the `Producer` and `Consumer`. The `Producer` and `Consumer` constructors are each passed the same `Buffer` object (`sharedLocation`), so each object refers to the same `Buffer`. These lines also implicitly launch the threads and call each `Runnable`’s `run` method. Finally, line 25 calls method `shutdown` so that the application can terminate when the threads executing the `Producer` and `Consumer` complete their tasks and line 26 waits for the scheduled tasks to complete. When `main` terminates, the main thread of execution enters the *terminated* state.

```
// Fig. 23.13: SharedBufferTest.java
// Application with two threads manipulating an u
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class SharedBufferTest {
   public static void main(String[] args) throws
      // create new thread pool
      ExecutorService executorService = Executor

      // create UnsynchronizedBuffer to store in
      Buffer sharedLocation = new Unsynchronized

      System.out.println(
         "Action\t\tValue\tSum of Produced\tSum
      System.out.printf(
         "------\t\t-----\t---------------\t----

      // execute the Producer and Consumer, givi
      // access to the sharedLocation
      executorService.execute(new Producer(share
      executorService.execute(new Consumer(share

      executorService.shutdown(); // terminate a
      executorService.awaitTermination(1, TimeUn
   }
}
```

Action	Value	Sum of Produced	Sum of Consumed
------	-----	---------------	---------------
Producer	1	1	

writes			
Producer writes	2	3	*—1 lost*
Producer writes	3	6	*—2 lost*
Consumer reads	3		3
Producer writes	4	10	
Consumer reads	4		7
Producer writes	5	15	
Producer writes	6	21	*—5 lost*
Producer writes	7	28	*—6 lost*
Consumer reads	7	14	
Consumer reads	7		21 *— 7 read again*
Producer writes	8	36	
Consumer reads	8		29
Consumer reads	8		37 *— 8 read again*
Producer writes	9	45	
Producer writes	10	55	*— 9 lost*

```
Producer done producing
Terminating Producer
```

Consumer reads	10	47
Consumer reads	10	57 — *10 read again*
Consumer reads	10	67 — *10 read again*
Consumer reads	10	77 — *10 read again*

```
Consumer read values totaling 77
Terminating Consumer
```

Action	Value	Sum of Produced	Sum of Consumed
------	-----	--------------	--------------
Consumer reads	-1		-1 — *reads -1 bad data*
Producer writes	1	1	
Consumer reads	1		0
Consumer reads	1		1 — *1 read again*
Consumer reads	1		2 — *1 read again*
Consumer reads	1		3 — *1 read again*
Consumer reads	1		4 — *1 read again*
Producer writes	2	3	
Consumer			

reads	2		6
Producer writes	3	6	
Consumer reads	3		9
Producer writes	4	10	
Consumer reads	4		13
Producer writes	5	15	
Producer writes	6	21	*— 5 lost*
Consumer reads	6		19

```
Consumer read values totaling 19
Terminating Consumer
```

Producer writes	7	28	*— 7 never read*
Producer writes	8	36	*— 8 never read*
Producer writes	9	45	*— 9 never read*
Producer writes	10	55	*— 9 never read*

```
Producer done producing
Terminating Producer
```

Fig. 23.13

Application with two threads manipulating an unsynchronized buffer—the italicized text in the output is our commentary, which is not part of the program's output. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.)

Recall that the `Producer` should execute first, and every value it produces should be consumed exactly once by the `Consumer`. However, in the first output of Fig. 23.13, notice that the `Producer` writes `1`, `2` and `3` before the `Consumer` reads its first value (`3`). Therefore, the values `1` and `2` are *lost*. Later, `5`, `6` and `9` are *lost*, while `7` and `8` are *read twice* and `10` is read four times. So the first output produces an incorrect total of 77, instead of the correct total of 55. (Lines in the output where the `Producer` or `Consumer` acted out of order are highlighted.) In the second output, the `Consumer` reads the value `-1` *before* the `Producer` ever writes a value. The `Consumer` reads the value `1` *five times* before the `Producer` writes the value `2`. Meanwhile, `5`, `7`, `8`, `9` and `10` are all *lost*—the last four because the `Consumer` terminates *before* the `Producer`. The result is an incorrect consumer total of 19.

Error-Prevention Tip 23.1

Access to a shared object by concurrent threads must be controlled carefully or a program may produce incorrect results.

To solve the problems of *lost* and *duplicated* data, Section 23.6 presents an example in which we use an `ArrayBlockingQueue` (from package `java.util.concurrent`) to synchronize access to the shared object, guaranteeing that each and every value will be processed once and only once.