Multithreading - II

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.
- 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.
- 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.

```
1  // Fig. 26.9: Buffer.java
2  // Buffer interface specifies methods called by Producer and Consumer.
3  public interface Buffer
4  {
5     // place int value into Buffer
6     public void set( int value ) throws InterruptedException;
7
8     // return int value from Buffer
9     public int get() throws InterruptedException;
10 } // end interface Buffer
```

```
1  // Fig. 26.10: Producer.java
2  // Producer with a run method that inserts the values 1 to 10 in buffer.
3  import java.util.Random;
4
5  public class Producer implements Runnable
6  {
7     private final static Random generator = new Random();
8     private final Buffer sharedLocation; // reference to shared object
9
10    // constructor
11    public Producer( Buffer shared )
12    {
13        sharedLocation = shared;
14    } // end Producer constructor
15
```

```
16      // store values from 1 to 10 in sharedLocation
17      public void run()
18      {
19         int sum = 0;
20
21         for ( int count = 1; count <= 10; count++ )
22         {
23            try // sleep 0 to 3 seconds, then place value in Buffer
24            {
25               Thread.sleep( generator.nextInt( 3000 ) ); // random sleep
26               sharedLocation.set( count ); // set value in buffer
27               sum += count; // increment sum of values
28               System.out.printf( "\t%2d\n", sum );
29            } // end try
30            // if lines 25 or 26 get interrupted, print stack trace
31            catch ( InterruptedException exception )
32            {
33               exception.printStackTrace();
34            } // end catch
35         } // end for
36
```

```
37          System.out.println(
38             "Producer done producing\nTerminating Producer" );
39       } // end method run
40    } // end class Producer
```

```
1  // Fig. 26.11: Consumer.java
2  // Consumer with a run method that loops, reading 10 values from buffer.
3  import java.util.Random;
4
5  public class Consumer implements Runnable
6  {
7     private final static Random generator = new Random();
8     private final Buffer sharedLocation; // reference to shared object
9
10    // constructor
11    public Consumer( Buffer shared )
12    {
13       sharedLocation = shared;
14    } // end Consumer constructor
15
```

```
16       // read sharedLocation's value 10 times and sum the values
17       public void run()
18       {
19          int sum = 0;
20
21          for ( int count = 1; count <= 10; count++ )
22          {
23             // sleep 0 to 3 seconds, read value from buffer and add to sum
24             try
25             {
26                Thread.sleep( generator.nextInt( 3000 ) );
27                sum += sharedLocation.get();
28                System.out.printf( "\t\t\t%2d\n", sum );
29             } // end try
30             // if lines 26 or 27 get interrupted, print stack trace
31             catch ( InterruptedException exception )
32             {
33                exception.printStackTrace();
34             } // end catch
35          } // end for
36
```

```
37            System.out.printf( "\n%s %d\n%s\n",
38               "Consumer read values totaling", sum, "Terminating Consumer" );
39         } // end method run
40      } // end class Consumer
```

```java
// Fig. 26.12: UnsynchronizedBuffer.java
// UnsynchronizedBuffer maintains the shared integer that is accessed by
// a producer thread and a consumer thread via methods set and get.
public class UnsynchronizedBuffer implements Buffer
{
   private int buffer = -1; // shared by producer and consumer threads

   // place value into buffer
   public void set( int value ) throws InterruptedException
   {
      System.out.printf( "Producer writes\t%2d", value );
      buffer = value;
   } // end method set

   // return value from buffer
   public int get() throws InterruptedException
   {
      System.out.printf( "Consumer reads\t%2d", buffer );
      return buffer;
   } // end method get
} // end class UnsynchronizedBuffer
```

```
1  // Fig. 26.13: SharedBufferTest.java
2  // Application with two threads manipulating an unsynchronized buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class SharedBufferTest
7  {
8     public static void main( String[] args )
9     {
10       // create new thread pool with two threads
11       ExecutorService application = Executors.newCachedThreadPool();
12
13       // create UnsynchronizedBuffer to store ints
14       Buffer sharedLocation = new UnsynchronizedBuffer();
15
16       System.out.println(
17          "Action\t\tValue\tSum of Produced\tSum of Consumed" );
18       System.out.println(
19          "------\t\t-----\t---------------\t---------------\n" );
20
```

```
21          // execute the Producer and Consumer, giving each of them access
22          // to sharedLocation
23          application.execute( new Producer( sharedLocation ) );
24          application.execute( new Consumer( sharedLocation ) );
25
26          application.shutdown(); // terminate application when tasks complete
27       } // end main
28    } // end class SharedBufferTest
```

Action	Value	Sum of Produced	Sum of Consumed	
Producer writes	1	1		
Producer writes	2	3		— 1 is lost
Producer writes	3	6		— 2 is lost
Consumer reads	3		3	
Producer writes	4	10		
Consumer reads	4		7	
Producer writes	5	15		
Producer writes	6	21		— 5 is lost
Producer writes	7	28		— 6 is 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 is 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 is 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		—— 10 never read

```
Producer done producing
Terminating Producer
```

Producer/Consumer Relationship with Synchronization

- A shared buffer using the `synchronized` keyword and methods of class `Object`.
- The first step in synchronizing access to the buffer is to implement methods `get` and `set` as `synchronized` methods.
- This requires that a thread obtain the monitor lock on the `Buffer` object before attempting to access the buffer data.

Producer/Consumer Relationship with Synchronization (cont.)

- `Object` methods `wait`, `notify` and `notifyAll` can be used with conditions to make threads wait when they cannot perform their tasks.
- Calling `Object` method wait on a `synchronized` object releases its monitor lock, and places the calling thread in the *waiting state*.
- Call `Object` method notify on a `synchronized` object allows a waiting thread to transition to the *runnable state* again.
- If a thread calls notifyAll on the `synchronized` object, then all the threads waiting for the monitor lock become eligible to reacquire the lock.

It's an error if a thread issues a `wait`*, a* `notify` *or a* `notifyAll` *on an object without having acquired a lock for it. This causes an* **`IllegalMonitorStateException`**.

It's a good practice to use `notifyAll` *to notify* waiting *threads to become* runnable. *Doing so avoids the possibility that your program would forget about waiting threads, which would otherwise starve.*

```
1  // Fig. 26.16: SynchronizedBuffer.java
2  // Synchronizing access to shared data using Object
3  // methods wait and notifyAll.
4  public class SynchronizedBuffer implements Buffer
5  {
6     private int buffer = -1; // shared by producer and consumer threads
7     private boolean occupied = false; // whether the buffer is occupied
8
9     // place value into buffer
10    public synchronized void set( int value ) throws InterruptedException
11    {
12       // while there are no empty locations, place thread in waiting state
13       while ( occupied )
14       {
15          // output thread information and buffer information, then wait
16          System.out.println( "Producer tries to write." );
17          displayState( "Buffer full. Producer waits." );
18          wait();
19       } // end while
20
```

```
21          buffer = value; // set new buffer value
22
23          // indicate producer cannot store another value
24          // until consumer retrieves current buffer value
25          occupied = true;
26
27          displayState( "Producer writes " + buffer );
28
29          notifyAll(); // tell waiting thread(s) to enter runnable state
30       } // end method set; releases lock on SynchronizedBuffer
31
```

```
32      // return value from buffer
33      public synchronized int get() throws InterruptedException
34      {
35         // while no data to read, place thread in waiting state
36         while ( !occupied )
37         {
38            // output thread information and buffer information, then wait
39            System.out.println( "Consumer tries to read." );
40            displayState( "Buffer empty. Consumer waits." );
41            wait();
42         } // end while
43
44         // indicate that producer can store another value
45         // because consumer just retrieved buffer value
46         occupied = false;
47
48         displayState( "Consumer reads " + buffer );
49
50         notifyAll(); // tell waiting thread(s) to enter runnable state
51
52         return buffer;
53      } // end method get; releases lock on SynchronizedBuffer
54
```

```
55      // display current operation and buffer state
56      public void displayState( String operation )
57      {
58         System.out.printf( "%-40s%d\t\t%b\n\n", operation, buffer,
59            occupied );
60      } // end method displayState
61   } // end class SynchronizedBuffer
```

```
1  // Fig. 26.17: SharedBufferTest2.java
2  // Two threads correctly manipulating a synchronized buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class SharedBufferTest2
7  {
8     public static void main( String[] args )
9     {
10       // create a newCachedThreadPool
11       ExecutorService application = Executors.newCachedThreadPool();
12
13       // create SynchronizedBuffer to store ints
14       Buffer sharedLocation = new SynchronizedBuffer();
15
16       System.out.printf( "%-40s%s\t\t%s\n%-40s%s\n\n", "Operation",
17          "Buffer", "Occupied", "---------", "------\t\t--------" );
18
```

```
19          // execute the Producer and Consumer tasks
20          application.execute( new Producer( sharedLocation ) );
21          application.execute( new Consumer( sharedLocation ) );
22
23          application.shutdown();
24       } // end main
25    } // end class SharedBufferTest2
```

Operation	Buffer	Occupied
Consumer tries to read. Buffer empty. Consumer waits.	-1	false
Producer writes 1	1	true
Consumer reads 1	1	false
Consumer tries to read. Buffer empty. Consumer waits.	1	false
Producer writes 2	2	true
Consumer reads 2	2	false
Producer writes 3	3	true
Consumer reads 3	3	false
Producer writes 4	4	true
Producer tries to write. Buffer full. Producer waits.	4	true

Consumer reads 4	4	false
Producer writes 5	5	true
Consumer reads 5	5	false
Producer writes 6	6	true
Producer tries to write. Buffer full. Producer waits.	6	true
Consumer reads 6	6	false
Producer writes 7	7	true
Producer tries to write. Buffer full. Producer waits.	7	true
Consumer reads 7	7	false
Producer writes 8	8	true
Consumer reads 8	8	false
Consumer tries to read. Buffer empty. Consumer waits.	8	false

```
Producer writes 9                       9               true

Consumer reads 9                        9               false

Consumer tries to read.
Buffer empty. Consumer waits.           9               false

Producer writes 10                      10              true

Consumer reads 10                       10              false


Producer done producing
Terminating Producer

Consumer read values totaling 55
Terminating Consumer
```

Producer/Consumer Relationship: Bounded Buffers

- The previous program may not perform optimally.
- Two threads operating at different speeds, one them will spend more (or most) of its time waiting.
- Even same relative speed cab ccasionally become “out of sync
- 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.

Producer/Consumer Relationship: Bounded Buffers (cont.)

- 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.*